THÈSE DE DOCTORAT DE L'UNIVERSITÉ PIERRE ET MARIE CURIE

Spécialité : **Informatique**
École doctorale Informatique, Télécommunications et Électronique (Paris)

Présentée par **Jean-Yves VET**
pour obtenir le grade de

DOCTEUR de l'UNIVERSITÉ PIERRE ET MARIE CURIE

Parallélisme de tâches et localité de données dans un contexte multi-modèle de programmation pour supercalculateurs hiérarchiques et hétérogènes

---

Organisme d'accueil
CEA, DAM, DIF
F-91297 Arpajon, France

Après avis de :

M. François BODIN	Professeur des Universités	Rapporteur
M. Raymond NAMYST	Professeur des Universités	Rapporteur

Soutenue le 25 novembre 2013 devant la commission d'examen composée de :

M. François BODIN	Professeur des Universités	Rapporteur
M. Patrick CARRIBAULT	Ingénieur-chercheur au CEA	Encadrant CEA
M. Albert COHEN	Directeur de recherche à l'INRIA	Directeur de thèse
M. Thierry GAUTIER	Chercheur à l'INRIA	Examinateur
M. Raymond NAMYST	Professeur des Universités	Rapporteur
M. Jean-Pierre PANZIERA	Directeur de l'ingénierie HPC chez BULL	Examinateur
M. Gaël THOMAS	Maître de Conférences	Président

Remerciements

On ne peut jamais tourner une page de sa vie sans que s'y accroche une certaine nostalgie a écrit Ève Belisle. Au terme de cette grande aventure que peut être le parcours sinueux d'une thèse, je souhaite remercier toutes celles et tous ceux qui m'ont permis de découvrir le monde de la recherche, de progresser dans cette voie et de rendre cette période la plus plaisante possible. Je suis persuadé que je serai vite rattrapé par ce sentiment de nostalgie que j'associerai à cette époque de ma vie, très riche en apprentissages et en rencontres formidables.

Je tiens tout d'abord à témoigner ma gratitude à M. François Bodin et M. Raymond Namyst, pour avoir bien voulu juger mes travaux en tant que rapporteurs et qu'examinateurs. Merci également à M. Thierry Gautier, M. Jean-Pierre Panziera et M. Gaël Thomas pour avoir accepté de se joindre au jury de ma thèse.

Ma reconnaissance va à Albert Cohen. Il fut un directeur de thèse précieux, doué de capacités de raisonnement et de travail qui m'ont fortement impressionné et durablement marqué. Merci à toi Albert pour ton aménité, ta patience maintes fois éprouvée et ton soutien sans faille durant ces trois années. Je remercie chaleureusement Patrick Carribault qui a accepté de prendre en charge la responsabilité de mon encadrement au CEA. Patrick, tu as été un encadrant à l'écoute et qui a su faire preuve d'une grande pédagogie à mon égard. Tel un maître d'armes tu as su attirer mon attention sur la nécessité de prendre régulièrement du recul pour mieux faire mouche (et éviter de recourir à des outils *redondants*). Je suis navré de t'avoir fait subir, à répétition, mon humour noir et mon caractère peu optimiste en apparence. Je suis néanmoins ravi d'avoir pu récupérer la location de ton ancien logement (m'épargnant ainsi d'avoir à affronter les terribles ragondins d'Arpajon).

Amical clin d'œil à tous les membres de l'équipe MPC. J'ai grandement apprécié votre sympathie, votre soutien et votre partage de savoir-faire. En particulier, merci à Marc pour avoir passé, à l'époque, le flambeau à Patrick afin qu'il récupère le logement qu'il m'a laissé par la suite. Peut-être, la légende voudra-t-elle un jour que cette histoire tire ses origines d'une tradition plus ancienne. Ainsi, dois-je comprendre que je suis désormais investi d'une mission quasi ancestrale : ce logement restera au sein de l'équipe MPC ! Merci à Jean-Baptiste, collègue de bureau, co-renifleur en blouse blanche et amplificateur de folie (je t'avais prévenu que nous finirions en *HP*). Merci à Jérôme d'avoir su trouver le courage nécessaire pour nous supporter dans notre bureau exigüe, ne fut-ce que pour quelques mois ; pardon de t'avoir communiqué notre démence (*écureuil !*). Merci à Sylvain d'avoir été toujours si avenant et disponible, à Emmanuelle pour nous avoir communiqué à tous ta bonne humeur habituelle (et tes excellentes adresses de restaurants), à Antoine pour avoir tenté à maintes reprises de me motiver à courir. Merci à François pour tes interventions efficaces sur les machines, à Camille aka *DJ Vector paddé*, à Augustin, à Julien, à Aurèle dont le patronyme me fera toujours penser à un boys band moldave auteurs de paroles très appropriées (*NUMA NUMA*). Merci à Sébastien, maître *Yoda* des allocations mémoires, pour ses conseils techniques et pour nos discussions passionnantes. J'espère que tu continueras à m'épauler dans la même palanquée pour quelques bulles en tant que parfait binôme de plongée sous-marine. Pourrions-nous enfin trancher si le père d'Ondine a finalement bien pris la barque à congres au lieu de la barque à bars sachant que le congre est barivore ?

En évoquant ce questionnement de barque qui nous tourmente tant, un grand merci à mon parfait associé Mickaël pour avoir dirigé la nôtre de barque, seul à la barre pendant cette longue période. Tu m'as laissé l'opportunité de réaliser cette thèse sereinement et grâce à toi, nôtre entreprise ressemble désormais à un véritable petit navire.

Merci à tous les autres collègues que j'ai pu rencontrer à l'INRIA, à l'ENS ou au CEA, pour nos échanges enrichissants. Mes pensées vont tout particulièrement à Feng, Michael, Riyadh, Sébastien, Jordan *le philosophe des caches*, Betrand, Manu, Nicolas, Alex *Flash Gordon*, Thomas, Xavier.

Merci Lucie, pour avoir enduré mes variations d'humeur pendant mes périodes de stress et de nuits blanches. Tu as su m'apporter l'écoute, la tendresse et le réconfort dont j'avais besoin pour avancer et surmonter l'épreuve de la rédaction.

Enfin, mes plus profonds remerciements vont à mes parents qui sont extraordinaires. Merci pour votre soutien inconditionnel pendant toutes ces années. Malgré la distance, vous avez été à mes côtés à chaque instant pour me donner confiance en moi. C'est grâce à cette présence et vos encouragements que j'ai pu saisir la chance d'approfondir mes études dans un domaine qui me passionne et cela dans les meilleures conditions que l'on puisse souhaiter. Merci.

Résumé

Des besoins toujours plus conséquents en puissance de calcul, notamment pour faire face à de nouveaux défis dans le domaine de la simulation numérique, ont entraîné la complexification des moyens de traitement. Nous avons étudié des nœuds de calcul issus de supercalculateurs de classe pétaflopique tels que Tera-100 détenu par le CEA ainsi que la machine Curie appartenant à GENCI. Au cours de l'histoire des supercalculateurs, les architectures matérielles ont capitalisé de nombreuses innovations technologiques dont certaines ont permis de contourner des obstacles qui auraient pu considérablement retarder la progression des capacités de calcul. La consommation électrique est désormais devenue le principal frein à la conception de nouvelles machines plus performantes. Afin de progresser dans cette quête de puissance de traitement, des grappes de calcul intégrant des processeurs de différentes natures ont vu le jour. Ces machines hétérogènes combinent les capacités d'exécution de processeurs traditionnels à celles de composants spécialisés appelés accélérateurs. Ceux-ci présentent de meilleures caractéristiques de calcul pour des opérations massivement parallèles. En particulier, ces composants spécifiques offrent un rapport performance/consommation réduit du fait de l'adjonction de multiples unités d'exécution matériellement simplifiées. Avec la multiplication du nombre d'unités de calcul se pose la question de l'efficacité d'exploitation de ces machines sophistiquées. Les programmeurs d'applications sont dès à présent contraints de coordonner toutes ces ressources de calcul hétérogènes, lesquelles peuvent recourir à des modèles de programmation distincts pour mieux exprimer le degré de parallélisme d'une application, ce qui en intensifie la complexité du fait de leur exploitation conjointe. Toutes ces unités de traitement sont amenées à échanger des résultats de calcul intermédiaires pour produire, *in fine*, une solution correcte. Pourtant, certaines mémoires déportées dans les accélérateurs n'étant pas maintenues matériellement cohérentes entre elles et avec la mémoire centrale, cela impose aux développeurs de gérer eux-mêmes les transferts de données aux moments les plus appropriés, et introduit par conséquent une difficulté de programmation supplémentaire. Par ailleurs, les performances des programmes sont fortement conditionnées par la façon dont toutes les ressources de calcul accèdent aux données. Pour répondre à une problématique de pression accrue sur la mémoire centrale et pour être en mesure de maximiser le degré d'occupation des unités de traitement, les machines s'articulent autour d'une imbrication de plus en plus marquée de diverses mémoires. Favoriser une utilisation performante de la hiérarchie des mémoires passe par des schémas d'accès spécifiques qui respectent des principes de localité des données. Ces contraintes d'accès permettent de tirer un meilleur parti des machines tout en limitant les dépenses énergétiques induites par des accès distants, mais rendent en contrepartie l'élaboration des programmes plus délicate. Ainsi, il s'agit dès lors d'exploiter simultanément des architectures hétérogènes et hiérarchiques dans un contexte de grappes de calcul, tout en portant un intérêt particulier à l'accès aux données.

Cette thèse présente des contributions permettant d'accentuer la localité des données dans des nœuds de calcul hétérogènes. Elles s'appuient sur un modèle de programmation par tâches, dont l'originalité réside dans l'ajustement de la quantité de calcul en fonction de l'unité d'exécution ciblée. Ce modèle de programmation est particulièrement adapté à un équilibrage de charge dynamique entre des ressources de calcul hétérogènes. Il favorise une meilleure exploitation des unités de traitement en offrant une meilleure réactivité en présence de variations des temps d'exécution, lesquelles peuvent être générées par des codes de calcul irréguliers ou des mécanismes matériels difficilement prévisibles. De plus, grâce à une sémantique adaptée, la déclaration des tâches de calcul facilite le recours à des mécanismes de gestion automatisée des opérations de cohérence des mémoires déportées et décharge les développeurs de cette tâche fastidieuse et source d'erreurs. Nous avons développé la plateforme d'exécution H3LMS afin d' agréger les propositions de cette thèse. Cette plateforme est intégrée à l'environnement de programmation MPC, développé conjointement par le CEA et le laboratoire *Exascale Computing Research* afin de faciliter la cohabitation de plusieurs modèles de programmation pour une meilleure exploitation des grappes de calcul. H3LMS permet, entre autres, de mieux aiguiller les tâches vers les unités de traitement appropriées en réduisant la quantité de coûteux accès distants au sein d'un nœud de calcul. Ces travaux s'intéressent également à l'adaptation de codes de simulation existants, conçus à l'origine pour exploiter exclusivement des processeurs traditionnels et pouvant comporter plusieurs dizaines voire plusieurs centaines de milliers de lignes de code. En particulier, des applications transparentes à des codes de simulation existants sont présentées à travers l'instrumentation de bibliothèques standards telles que les BLAS. Les performances de la solution développée sont évaluées sur la bibliothèque Linpack et par une application numérique réaliste du CEA.

Introduction

“ Si six scies scient six cyprès, combien de cyprès scient six cent six scies ? ”

Anonyme, virelangue français

Les capacités de calcul de la machine d'IBM *Deep Blue* [1] marquèrent les esprits en 1997, puisqu'elles permirent de battre officiellement le champion du monde d'échecs *Gary Kasparov*, considéré comme l'un des meilleurs joueurs de l'histoire. Plus récemment, près de deux cent mille milliards de composants électroniques élémentaires[1], appelés *transistors*, ont été agrégés pour fournir à *Titan*[2] la capacité de calcul qui l'a élevée, en novembre 2012, au rang de machine la plus puissante au monde. À titre de comparaison, cela correspondrait à la quantité de neurones totalisée par près de deux-mille cerveaux humains. Ce nombre colossal n'est qu'un arrêt sur image dans l'histoire des superordinateurs et va vraisemblablement continuer à s'élever de manière exponentielle avec les prochaines générations de machine. En effet, la loi de Gordon E. Moore [3], réévaluée en 1975, n'a jamais failli en prédisant un doublement de la densité des puces électroniques tous les deux ans.

L'époque des premiers calculateurs semble bien lointaine lorsque nous comparons leurs performances à celles de nos machines contemporaines. En 2013, elles disposaient d'une capacité de calcul d'un million de milliards de fois plus conséquente[3]. Leur évolution a en réalité progressé à une vitesse effrénée. La préhistoire de ces machines remonte à 1642 lorsque Blaise Pascal inventa la *machine arithmétique* surnommée la *Pascaline* [4], première machine à calculer purement mécanique. En 1834, le précurseur britannique Charles Babbage imagina la *machine analytique* [5]. Bien que jamais construite, elle ouvrit la voie aux machines programmables en étant dotée, sur le papier, de mémoires, d'imprimantes, d'une unité centrale et de lecteurs de cartes inspirés du métier à tisser *Jacquard* pour y insérer les instructions et les données sur lesquelles elles opèrent. Puis, en moins d'un siècle, les avancées ont permis de passer des grandes théories, telle celle exposée dans l'article fondateur de la science informatique par Alan Turing en 1936 [6], à l'automatisation concrète du calcul par de puissantes machines. Peu avant la Seconde Guerre mondiale, les premiers calculateurs électromécaniques sont apparus selon l'idée de Turing. Quelques années plus tard, en 1941, le calculateur Z3 est achevé et est alors considéré comme la première machine programmable fonctionnelle. Ce système, conçu par Konrad Zuse, fut employé par la Wehrmacht pour le guidage de missiles. La série-Z [7] inaugura bon nombres d'avancées technologiques telle celle de l'arithmétique binaire et celle des nombres à virgule flottante en tant qu'approximations de nombres réels [8]. Après la Seconde Guerre mondiale, ces calculateurs ont évolué et ont donné naissance à des machines au fonctionnement purement électronique que nous appelons depuis des *ordinateurs*. Initiées par la découverte du transistor en 1947, les performances de ces machines électroniques ont été considérablement décuplées par la suite.

[1] 18 688 *AMD Opteron 6274* de $2{,}4 \times 10^9$ transistors et 18 688 *Nvidia Tesla K20X* de $7{,}1 \times 10^9$ transistors.

[2] Classement *TOP500*, novembre 2012 [2].

[3] *Titan* (2012) : 17 PFLOP/s || *Z3* (1941) : 20 FLOP/s.

Ces ordinateurs ont été rapidement employés pour reproduire, par le calcul, des phénomènes physiques. Ces *simulations informatiques* ont ainsi permis de décrire des phénomènes sans qu'ils ne se soient réellement produits. La première simulation a été élaborée lors de la Seconde Guerre mondiale pour répondre aux besoins de conception et de modélisation du *projet Manhattan*[4], porté notamment par le mathématicien et physicien américain d'origine hongroise John von Neumann [9]. En 1953 eu lieu la première simulation dans le domaine civil en physique théorique, connue sous le nom d'*expérience de Fermi-Pasta-Ulam* [10]. Depuis, les simulations numériques se sont répandues dans divers domaines. Elles permettent, par exemple, d'anticiper des phénomènes météorologiques ou sont encore employées en tant qu'atout financier dans l'industrie en accélérant des cycles de conception. Certains secteurs nécessitent une capacité de traitement de plus en plus conséquente pour simuler des systèmes toujours plus complexes, donnant lieu à une course perpétuelle à la puissance de calcul.

Le terme de *supercalculateur* (ou *superordinateur*) apparaît dans les années soixante. Il s'agit d'un ordinateur conçu pour atteindre les plus hautes performances de calcul possibles avec les technologies connues lors de sa conception. La science des supercalculateurs est appelée *calcul haute performance* (HPC[5]). Les premières machines enchaînaient successivement les opérations qui étaient traitées par un unique système exécutif appelé *processeur* (ou CPU[6]). Dans les années soixante-dix, l'équivalent de la puissance de calcul de plusieurs processeurs est combinée. Afin de tirer le meilleur parti de ces machines, toutes ces unités doivent travailler de concert. C'est alors qu'a émergé le *calcul parallèle* permettant l'exécution simultanée de plusieurs opérations afin de les traiter plus rapidement. Ainsi, ces travaux informatiques peuvent être accélérés par rapport à des opérations successives, dites *séquentielles*. L'élaboration des codes calcul s'en trouve complexifiée, puisqu'il s'agit désormais de coordonner les différents processeurs qui sont amenés à échanger des résultats de calcul intermédiaires pour produire, *in fine*, une solution correcte. Avec la multiplication du nombre d'unités de calcul se pose la question de l'efficacité d'exploitation de ces machines sophistiquées. Une accélération linéaire n'est quasiment jamais atteinte. En d'autres termes, et en faisant écho au virelangue introductif, si des opérations sont traitées par 6 processeurs, il faudra bien souvent plus de 606 processeurs pour effectuer une quantité de calcul 101 fois plus importante pendant le même laps de temps. Optimiser l'efficacité des calculs effectués par une machine moderne est un des challenges de l'*informatique parallèle*.

Contexte, objectifs et contributions

Outre un degré de parallélisme insuffisant, différents facteurs peuvent constituer un frein à l'obtention de hautes performances de calcul. L'élément certainement le plus décisif concerne l'accès aux données sur lesquelles les calculs sont réalisés. Pour les machines contemporaines, ces données sont stockées dans une imbrication de mémoires plus ou moins rapides. Lorsque les unités de traitement sont en attente de récupération de ces données, elles peuvent se trouver sous-exploitées, ce qui rallonge inutilement le temps d'exécution des programmes. Favoriser une exploitation performante de la hiérarchie des mémoires passe par des schémas d'accès spécifiques qui respectent des principes de *localité des données*. Ces contraintes d'accès permettant de tirer un meilleur parti des machines, rendent l'élaboration des programmes plus délicate. Par ailleurs, des processeurs composés de centaines, voire de milliers d'unités de traitement ont vu le jour afin de progresser dans la quête de puissance de calcul. Ceux-ci présentent de meilleures caractéristiques de calcul pour des opérations *massivement parallèles*. Ils offrent, entre autres, un rapport performance/consommation réduit du fait de l'adjonction de multiples unités de calcul simplifiées. Ils sont ainsi appelés *accélérateurs* lorsqu'ils sont combinés à des processeurs plus traditionnels. Ces

[4] Nom de code du projet de recherche de la première tête nucléaire.

[5] **HPC :** *High Performance Computing*.

[6] **CPU :** *Central Processing Unit*.

nouvelles machines sont alors qualifiées d'hétérogènes puisqu'elles agrègent des processeurs de différentes natures et des mémoires qui peuvent être disjointes. Les programmeurs sont dès à présent contraints de coordonner ces mémoires et ces diverses ressources de calcul. Ils peuvent recourir à des modèles de programmation variés pour exprimer le degré de parallélisme le mieux adapté à chaque *microarchitecture* de processeur. Les développeurs d'applications assistent ainsi à une complexification de l'exploitation à haute performance des machines, d'une part en s'efforçant de limiter la baisse d'activité des architectures causée par la manipulation des données stockées dans la hiérarchie mémoire, et d'autre part en essayant de répartir efficacement les calculs entre toutes les ressources de traitement disponibles.

La programmation par *tâches de calcul* peut faciliter la gestion de ces machines hétérogènes. Elle organise les instructions par lots de traitements appelés *tâches* et participe à renforcer les principes de localité de données. Ce modèle repose sur un logiciel appelé *plate-forme d'exécution* qui répartit dynamiquement, entre les différentes ressources de traitement, les tâches décrites dans un code de calcul. Les opérations embarquées dans une tâche peuvent être exprimées à l'aide d'un ou de plusieurs autres modèles afin d'offrir une compatibilité avec les différents processeurs qui peuvent être ciblés. La programmation par tâches de calcul permet aussi d'établir des mécanismes capables de décharger le programmeur de certaines contraintes d'exploitation liées à la gestion de la mémoire. L'objectif de cette thèse est l'élaboration d'une telle plate-forme permettant de recourir dynamiquement à des ressources de calcul hétérogènes, tout en portant un intérêt particulier à l'accès aux données pour atteindre de hautes performances. Ces travaux s'intéressent également au portage progressif de codes de simulation existants, conçus à l'origine pour des processeurs traditionnels. Les contributions suivantes sont ainsi proposées :

1. Des **tâches de calcul décomposables** permettant d'adapter dynamiquement la charge de travail en fonction des unités de calcul ciblées tout en amplifiant la localité spatiale des données (chapitre 4).
2. Une **coopération hiérarchique** des unités exécutantes en s'appuyant sur les contraintes architecturales des machines actuelles (chapitre 4).
3. Une interface de programmation permettant de **répartir des données** en mémoire centrale afin de réduire la quantité d'accès distants lors de l'ordonnancement des tâches de calcul (chapitre 4).
4. Une **gestion automatisée** des données en prenant en charge leur transfert entre les différentes mémoires et en assurant leur cohérence (chapitre 5).
5. Une **rétention accrue des données** dans les mémoires déportées, couplée à l'ordonnancement des opérations afin de diminuer la quantité des données véhiculées (chapitre 5).
6. Une proposition d'interface permettant la **réutilisation de données préchargées** entre des appels à une bibliothèque de calculs optimisés et d'autres traitements (chapitre 5).
7. La **conception d'un support d'exécution** agrégeant les fonctionnalités précédentes et intégré à un environnement de programmation facilitant les interactions avec d'autres modèles de programmation (chapitre 6).

Certaines des ces contributions ont été publiées et présentées à MUTIPROG[7] 2012 [11].

[7] **MULTIPROG :** *Programmability Issues for Heterogeneous Multicores*.

De manière générale, ce document décrit le couplage entre l'ordonnancement de tâches de calcul à granularités multiples avec des politiques de remplacement de caches logiciels, tout en amplifiant la localité spatiale des données. En particulier, cette solution est évaluée sur une application numérique réaliste dont les performances sont limitées par la bande passante mémoire. Grâce aux contributions qui seront détaillées tout au long du manuscrit, la plate-forme *H3LMS* que nous avons développée permet d'accélérer cette application d'un facteur $2{,}64\times$ en exploitant deux processeurs graphiques et deux processeurs centraux par rapport à une exécution parallèle reposant uniquement sur les cœurs de calcul généralistes.

Organisation du document

Ce manuscrit est organisé en trois parties. La première précise le contexte. Elle met en exergue, dans un premier chapitre, les besoins en puissance de calcul des codes de simulation tout en retraçant l'évolution des architectures matérielles. Les avancées technologiques y sont introduites en insistant sur les aspects limitant induits par l'accès aux données. Le chapitre 2 explore plus en détails les différentes contraintes associées à la récupération des données dans les mémoires d'une machine parallèle moderne. Le chapitre 3 dépeint ensuite les modèles de programmation à disposition des développeurs de codes de simulation ainsi que l'état de l'art en matière d'équilibrage de charge dynamique et hétérogène.

La deuxième partie détaille les contributions en évaluant au fur et à mesure ces propositions sur des codes de calcul d'algèbre linéaire. Le chapitre 4 décrit les travaux associés à un renforcement de la localité spatiale grâce à une décomposition à la volée des tâches de calcul, à une organisation hiérarchique de leur exécution et à un placement initial soigneusement étudié des données en mémoire. Le chapitre 5 expose les travaux entrepris pour automatiser la gestion de mémoires disjointes et amplifier la localité temporelle grâce à des mécanismes de rétention de données. Le chapitre 6 présente les mécanismes d'ordonnancement permettant d'agréger les contributions sur la localité des données dans une unique plate-forme d'exécution. Cette dernière est intégrée à un environnement facilitant une cohabitation avec d'autres modèles de programmation. Le chapitre 7 évalue les performances de cette solution en exposant les résultats obtenus à partir de la bibliothèque de performance *Linpack* et d'une mini application du CEA.

Enfin, une troisième et dernière partie synthétise les travaux accomplis, puis aborde les améliorations et les perspectives envisagées à court et à plus long termes.

Table des matières

Première partie

Contexte

Chapitre 1

Les supercalculateurs au service des codes de simulation

> *“ Depuis plus d'une décennie, des visionnaires ont soutenu que l'organisation d'un unique ordinateur a atteint ses limites et que des progrès significatifs ne peuvent être réalisés que par l'interconnexion d'une multitude d'ordinateurs. ”*
>
> Gene Amdahl, 1967

Après la Seconde Guerre mondiale, des méthodes de simulation se sont développées et ont été adoptées dans divers domaines tels que la physique, la chimie, la biologie, le génie civil, etc. Ces programmes de calcul se sont complexifiés en s'appuyant d'abord sur des représentations tridimensionnelles de plus en plus réalistes, puis après les années 90, en combinant plusieurs phénomènes physiques. Cette discipline requiert bien souvent des capacités de traitement de plus en plus importantes pour aboutir à des résultats dans un délai raisonnable et pour entreprendre des simulations plus précises et plus complètes. La conséquence de ces besoins en *calcul intensif* se traduit par une compétition effrénée pour détenir une puissance de traitement toujours plus imposante. Des supercalculateurs sont ainsi conçus en profitant des avancées technologiques pour offrir des capacités de calcul et de stockage accrues chaque année. Ils représentent un atout indéniable pour des laboratoires de recherche, pour les entreprises afin d'accélérer des cycles de conception à coûts réduits ou comme outil permettant d'asseoir la crédibilité des grandes puissances dans la conception et l'amélioration des moyens de défense et de dissuasion.

Le classement *TOP500* [12] des 500 machines les plus puissantes au monde témoigne de cette course aux capacités de traitements (cf. **figure 1.1**). Le rang de chaque machine est établi à partir d'un test de performance[8] issu de la bibliothèque *Linpack* [13] qui résout un système de n équations à n inconnues par une factorisation de Gauss avec pivot partiel [14]. L'indice de performance est évalué en FLOP/s[9] représentant la quantité moyenne d'opérations à virgules flottantes effectuées par seconde sur l'ensemble du test. Ce classement est réactualisé deux fois par an pour mieux refléter les avancées dans le domaine. Le test Linpack est un programme extrêmement optimisé. Lorsqu'il est correctement paramétré, il est principalement limité par les capacités de calcul de la machine. Ce test de performance est alors peu représentatif du comportement des codes de simulation, mais permet tout de même d'évaluer une borne supérieure des capacités de calcul brutes atteignables par une machine.

[8] *Benchmark* en anglais.

[9] **FLOP/s :** *FLoating point Operations Per Second*.

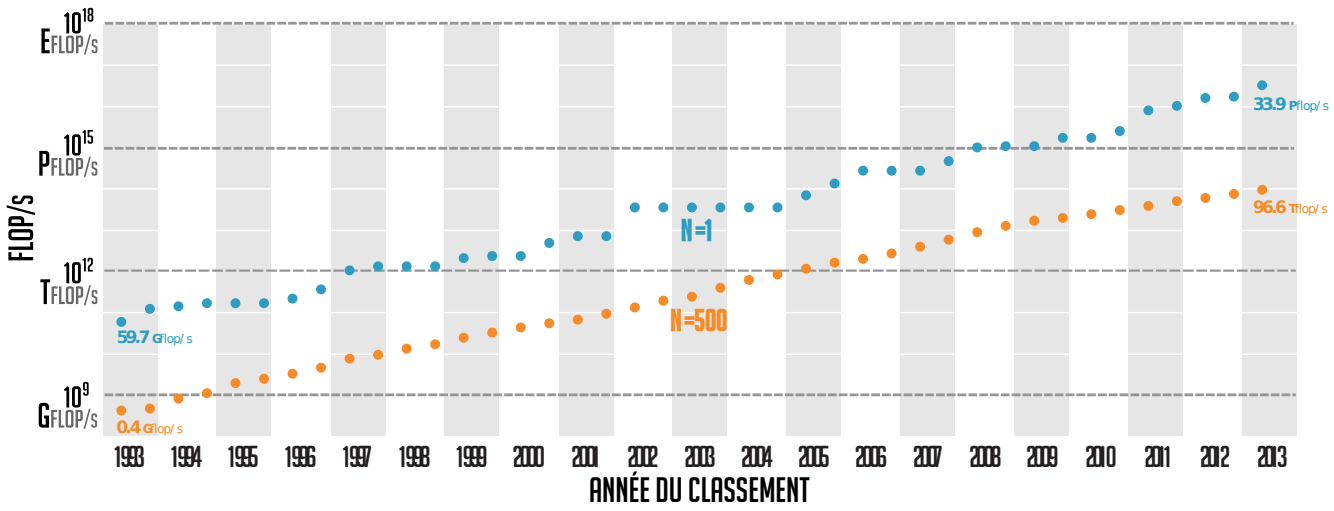


FIGURE 1.1 – Évolution des performances de la première ($N = 1$) et de la dernière machine ($N = 500$) du classement *TOP500* *(source : top500.org)*.

Outre le développement des supercalculateurs, les micro-ordinateurs se sont démocratisés et sont progressivement apparus dans les entreprises, dans les laboratoires de recherche et chez les particuliers. Cette adoption massive a contribué à leur amélioration et à la diminution de leur coût de fabrication grâce à une commercialisation à grande échelle. Certains organismes ne disposaient pas des moyens financiers suffisants pour acquérir un supercalculateur de l'époque. Une parade, anticipée par Gene Amdahl en 1967, consistait à combiner la puissance de calcul de plusieurs ordinateurs personnels [15]. Cette façon de procéder a été adoptée pour entreprendre le projet *Beowulf* [16] qui a abouti à l'assemblage d'une machine par la NASA[10] en 1994. Celle-ci offrait une meilleure flexibilité d'évolution en reliant, à titre d'exemple, un plus grand nombre d'ordinateurs basés sur des composants standards. L'amélioration des interconnexions à la fin des années 90 a amorcé un tournant dans la façon dont les superordinateurs étaient élaborés. De nouveaux supercalculateurs, composés de puissants ordinateurs, ou *nœuds de calcul*, physiquement reliés en *grappes*[11], apparaissent alors. En juin 2013, la majorité des machines classées au *TOP500* étaient basées sur une architecture de type *grappe de calcul*. Chaque nœud repose sur une agrégation de plusieurs technologies acquises et améliorées à différentes périodes de l'histoire de l'évolution des machines. Au fil du temps, les supercalculateurs deviennent de plus en plus puissants, mais, en contrepartie, ils se complexifient.

En premier lieu, ce chapitre esquissera les besoins en puissance des programmes de simulations. En second lieu, il abordera les grandes étapes de l'évolution des machines afin d'appréhender les contraintes imposées par les technologies associées. Enfin, en dernier lieu, seront évoquées certaines spécificités des machines contemporaines.

[10] **NASA :** *National Aeronautics and Space Administration*.

[11] *Cluster* en anglais.

1.1 La puissance de calcul pour la simulation numérique

La simulation numérique vise majoritairement à reproduire des phénomènes physiques. En s'appuyant sur les recherches fondamentales, de nouveaux enjeux sont apparus. Elle est rapidement devenue inévitable dans divers domaines pour résoudre des problèmes intenses en calcul. Cependant, la simulation n'est pas que du calcul, c'est aussi le fruit de l'association de diverses compétences qui ne se résument pas à l'informatique. Il faut dans un premier temps élaborer un modèle physique des phénomènes étudiés et analyser les équations associées. La conception d'algorithmes permet seulement ensuite d'exprimer la façon dont les machines vont résoudre ces équations via le calcul numérique. Il s'agit de décrire cette résolution (éventuellement réajuster la modélisation) pour exploiter efficacement les supercalculateurs qui sont devenus massivement parallèles. Enfin, la précision et la validité des solutions obtenues par la simulation doivent être évaluées pour s'assurer qu'elles sont bien représentatives du cadre d'étude considéré.

Dans l'industrie, les simulations interviennent souvent pour minimiser le recours à des moyens coûteux comme la production de maquettes et de prototypes. Il est moins onéreux de concevoir un modèle adéquat et de l'étudier sur un ordinateur que de mettre en place une série d'épreuves réelles. En aéronautique, le comportement d'un profil d'aile d'avion peut être étudié afin de maximiser sa portance, sans avoir à la construire. Dans les phases d'exploration de l'industrie pétrolière, elle sert à extrapoler les données sismiques en aiguillant les géologues afin de limiter les forages coûteux et difficilement accessibles pour se focaliser sur les sites les plus profitables. Les simulations peuvent aussi accélérer le temps et anticiper certains phénomènes. En astrophysique, elles permettent en particulier de recréer de longs phénomènes inaccessibles autrement. Le projet DEUS[12] [17] aspire à reconstituer numériquement la formation de l'univers pour comprendre la nature de l'énergie noire. Les prévisions météorologiques possèdent des contraintes de précision et également de rapidité d'exécution. Il s'agit de prévoir le temps qu'il fera dans un futur très proche en tenant compte d'une grande quantité de paramètres relevés par les stations. À un tout autre ordre de grandeur, les simulations ont conduit à l'élaboration de composants électroniques plus performants. En science des matériaux, il est possible d'agir sur les différents paramètres à l'échelle des molécules et des atomes. Des calculs qualifiés d'*ab initio* sont en plein essor et consistent à retrouver les propriétés des matériaux en se basant sur la physique atomique. Ils permettent de s'affranchir de données expérimentales qui pouvaient servir de postulats et de paramètres pour étudier des phénomènes plus conséquents. Par ailleurs, les simulation participent à réduire les risques d'accidents en étudiant des phénomènes dangereux. Des modélisations de tsunamis sont par exemple réalisées dans le cadre de prévention pour déterminer les zones potentiellement exposées.

Ainsi, la simulations numérique sert à étudier les propriétés de systèmes modélisés et d'anticiper leur évolution. Elle permet d'aller au-delà du couple classique constitué de la théorie et de l'expérimentation, en explorant une plus grande quantité de possibilités. "*La simulation est une extension ou une généralisation de l'expérience*" selon l'académicien Michel Serres [18]. Elle permet de grossir les espaces, d'accroître la vitesse du temps, d'augmenter ou de diminuer les interactions et les contraintes. Elle peut approcher des phénomènes de plus en plus complexes en s'appuyant sur des capacités de traitements toujours plus conséquentes. L'amélioration des superordinateurs comble progressivement ces besoins de puissance. Cependant, ces machines tendent à se complexifier. Étudier leurs évolutions nous aide à comprendre la technologie actuelle.

[12] **DEUS :** *Dark Energy Universe Simulation*.

1.2 Les architectures parallèles et l'évolution des supercalculateurs

En 1966, Michael J. Flynn propose une classification [19] des architectures en fonction des types d'opérations. Celles-ci sont caractérisées par le nombre d'instructions et la quantité de flots de données impactés par ces traitements. Les supercalculateurs actuels reposent sur plusieurs catégories via divers mécanismes, héritages des technologies développées au cours de l'évolution des machines. Cette section retrace les principales avancées en s'appuyant sur la taxonomie de Flynn.

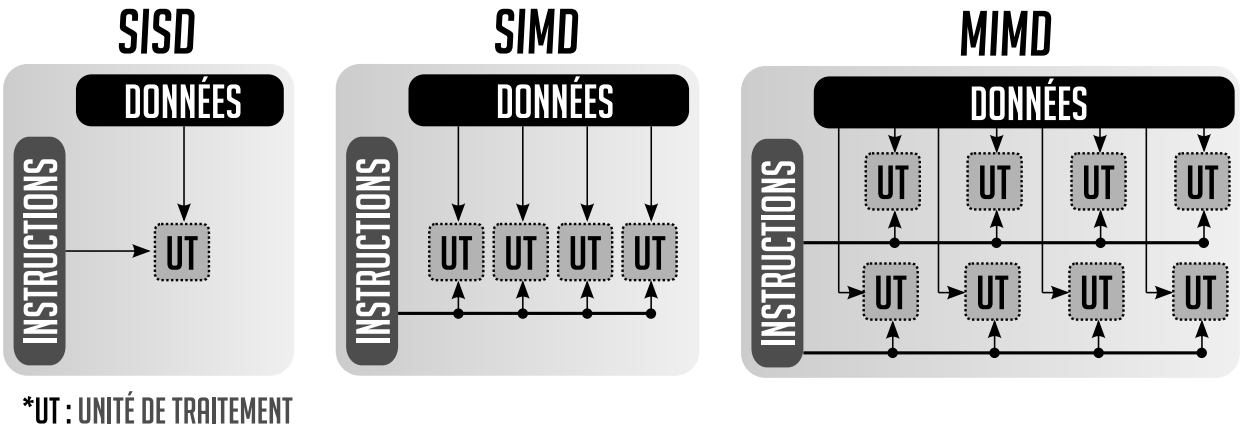


FIGURE 1.2 – Taxonomie de Flynn (Le modèle *MISD* n'est pas représenté, car peu employé).

1.2.1 Des machines séquentielles aux grappes de calcul

Les premières machines dites *scalaires* fonctionnaient séquentiellement, c'est-à-dire qu'une instruction modifiait un flux de données à la fois. Il s'agit de la catégorie la plus élémentaire appelée ***SISD***[13] de la classification de Flynn (cf. figure 1.2). En 1962, la machine ATLAS [20] de l'université de Manchester et Ferranti accélère les opérations *SISD* en ouvrant la voie au *parallélisme d'instructions*[14]. Ce fut la première machine à employer des techniques similaires à celles élaborées par Henry Ford dans le domaine de l'automobile. Un *pipeline d'exécution*, sorte de ligne de production, accroît le rendement des opérations en permettant une exécution simultanée de plusieurs sous étapes d'une instruction. Les travaux de Seymour Cray, Jim Thornton et Dean Roushde de l'entreprise CDC[15] ont permis d'améliorer davantage les performances [21]. En 1963, le *CDC-6600* [22] est considéré comme le premier supercalculateur. C'est la première machine *super-scalaire*. Elle rend possible l'exécution de plusieurs instructions simultanément grâce à diverses unités de traitement. Elle était plus de dix fois plus performante que ses concurrentes à sa sortie, et elle restera la machine la plus puissante au monde jusqu'en 1969. Depuis, les traitements *SISD* ont été améliorés en combinant des techniques permettant de gérer plus de parallélisme d'instructions.

Dans les années 70, des supercalculateurs sont capables de traiter plusieurs flots de données à partir d'une même instruction ; c'est l'hégémonie des machines *vectorielles* [23]. Ces architectures ***SIMD***[16] (cf. figure 1.2) émergent avec le *CDC STAR-100* [24] et le *TI-ASC* [25]. Celles-ci restent toutefois plus lentes que les machines contemporaines purement *super-scalaires*. En 1972, Seymour Cray quitte *CDC* et fonde *Cray Research* dont le premier produit en 1975 fut le *Cray-1* [26], un des supercalculateurs les plus renommés de l'histoire, combinant opérations vectorielles et

[13] **SISD :** *Single Instruction, Single Data*.
[14] **ILP :** *Instruction Level Parallelism*.
[15] **CDC :** *Control Data Corporation*.
[16] **SIMD :** *Single Instruction, Multiple Data*.

parallélisme d'instructions. Des emplacements mémoires plus rapides, appelés *registres*, sont positionnés physiquement plus proches des unités de calcul. Ils permettent de réduire les temps d'accès aux données réutilisées plusieurs fois d'affilée par rapport à des opérations directement effectuées en mémoire centrale. De nos jours, les processeurs modernes pour le marché grand public, de même que ceux dédiés au calcul, embarquent des registres et des unités de calcul vectorielles accessibles par des instructions de bas niveau.

Des machines constituées de plusieurs processeurs sont apparues dans les années 70. Des flots d'instructions différents pouvaient être réalisés simultanément sur plusieurs flots de données, permettant ainsi un traitement de type ***MIMD***[17] (cf. **figure 1.2**). Ces premières machines multi-processeurs sont pour la plupart dites *asymétriques* (AMP[18]) puisque chaque processeur possédait sa propre mémoire, ainsi que des propriétés d'accès aux périphériques distinctes. À titre d'exemple, le deuxième processeur que pouvait héberger la machine *VAX-11/782* n'était pas connecté aux périphériques d'entrées/sorties [27]. Ces processeurs supplémentaires étaient généralement considérés comme des extensions afin d'augmenter, à moindre frais, les capacités de calcul. Les systèmes d'exploitation de l'époque étaient cependant développés pour un unique processeur central et ne permettaient pas de contrôler efficacement plusieurs processeurs à la fois. Des machines *symétriques* (SMP[19]), plus aisées à programmer, ont progressivement remplacé les systèmes asymétriques. Les processeurs qui composaient ces nouvelles architectures étaient alors fortement couplés. Ils partageaient la même mémoire, possédaient un plein accès à toutes les entrées/sorties et étaient pilotés par un unique système d'exploitation. De 1982 à 1985, le *Cray X-MP* [28] fut la machine la plus puissante au monde. Elle intégrait deux processeurs basés sur des améliorations du Cray-1, lesquels étaient exploités de manière symétrique autour d'une mémoire commune. Dans les années 90, des machines *massivement parallèles*, constituées de centaines, voire de milliers d'unités de calcul indépendantes sont développées. Le *CM-5* [29], commercialisé en 1992 par *Thinking Machines Corporation*, en est un exemple avec ses 1024 processeurs. Inspiré par le projet *Beowulf*, à l'origine des architectures en *grappes* à base de composants standards, la machine *ASCI Red* [30] réunissait 4 510 nœuds de calcul, chacun constitué d'une mémoire distincte et de deux processeurs. Lors de sa construction en 1996, c'était le premier supercalculateur à dépasser le millier de milliards d'opérations par secondes (TFLOP/s) au test de performance *Linpack*.

Depuis, les assemblages par *grappes* se sont répandus, combinant la puissance de nombreux nœuds de calcul et embarquant plusieurs processeurs. Chaque nouvelle génération de machine pouvait à la fois augmenter sa quantité de processeurs interconnectés et profiter de l'amélioration de leurs performances. Dans les années 2000, plusieurs limitations technologiques ont cependant contraint les fondeurs à modifier la façon dont les processeurs étaient élaborés.

1.2.2 La révolution des processeurs multi-cœurs

Les fondeurs, tels qu'Intel et AMD, profitaient du doublement de la densité des puces prévu par la loi de Moore afin d'accroître les performances des processeurs. Deux leviers permettaient cela, grâce, d'une part à l'amélioration des microarchitectures pour un meilleur parallélisme d'instructions, et d'autre part à l'élévation de la *fréquence d'horloge*, ce qui permettait d'accélérer la cadence d'exécution des calculs. Depuis le début des années 2000, ces deux facteurs de performance ont rencontré des limitations majeures.

[17] **MIMD :** *Multiple Instruction, Multiple Data*.
[18] **AMP :** *Asymmetric Multi-Processing*.
[19] **SMP :** *Symmetric Multi-Processing*.

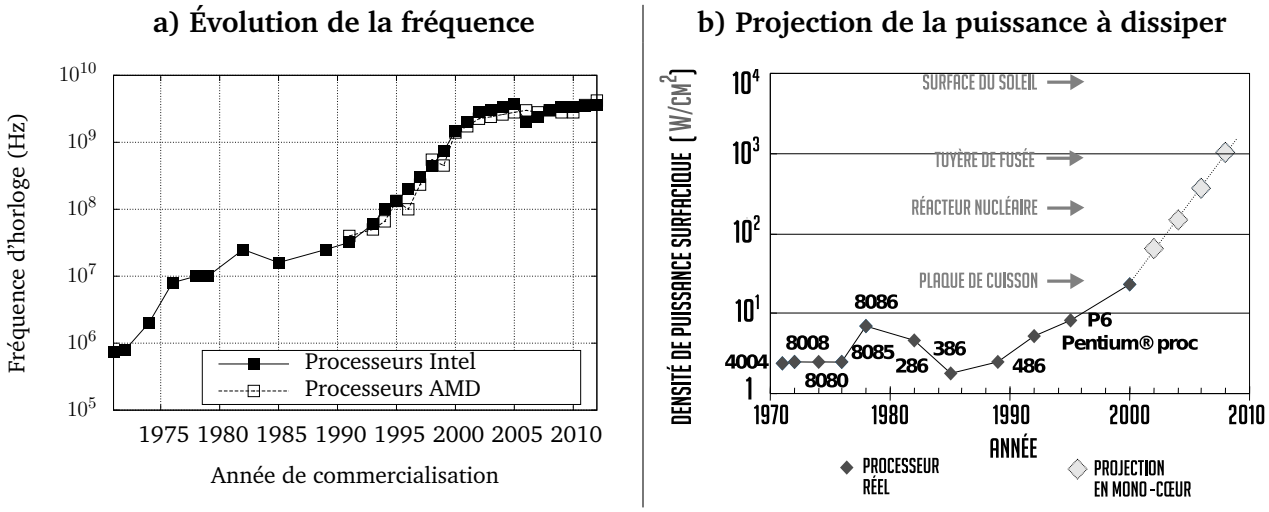


FIGURE 1.3 – **a)** Évolution de la fréquence des microprocesseurs. **b)** Projection de la puissance surfacique des microprocesseurs avant l'introduction des multi-cœurs *(source : S. Borkar, Intel)*.

La fréquence d'horloge des processeurs est passée de *740 kHz* (740 mille cycles par seconde) en 1971 à environ *3 GHz* (3 milliards de cycles par seconde) en 2004. Huit ans plus tard, cette fréquence n'avait globalement pas progressé comme l'illustre la **figure 1.3a**. Pour comprendre la cause de cette stagnation, il faut s'intéresser au fonctionnement physique des processeurs. Ceux-ci sont des composants très denses constitués de transistors. Un transistor est un élément électronique à deux états, ouvert ou fermé, pour représenter une information binaire. Le pic de consommation d'énergie est obtenu lorsqu'il change d'état. Or, dans un processeur, les transistors passent justement leur temps à changer d'état ; la vitesse de commutation étant cadencée par la fréquence d'horloge. Plus les transistors commutent rapidement, plus l'énergie consommée est importante. L'équation de la puissance dynamique d'un transistor peut être écrite de la manière suivante :

$$P = K \times (\text{capacité de charge}) \times (\text{tension})^2 \times (\text{fréquence de commutation})$$

La théorie du passage à l'échelle de Dennard [31] anticipait une meilleure efficacité des transistors lorsque ceux-ci étaient gravés plus finement. Il était ainsi possible de réduire la *tension* et la *capacité de charge*. Couplée à la loi de Moore, elle prévoyait un doublement du nombre de transistors tous les deux ans et une élévation de la fréquence de 40% en conservant une puissance électrique équivalant à la génération précédente. Depuis le début des années 2000, la théorie de Dennard n'est plus applicable. Une butée technologique a été atteinte, appelée *mur de la puissance*. Augmenter la fréquence des processeurs implique désormais inévitablement une élévation de la puissance consommée. Une grande partie de celle-ci est transformée en chaleur qu'il faut dissiper pour ne pas entraîner la destruction du composant. Pendant un temps, une parade a permis de continuer à augmenter la fréquence d'horloge en employant des dissipateurs et des ventilateurs de plus en plus conséquents. Cette solution a rapidement atteint ses limites, car la puissance à dissiper augmentait à une allure folle. La **figure 1.3b** est une projection pessimiste en 2000 de l'ampleur de la puissance thermique à évacuer en conservant la même vitesse d'évolution, sans rupture technologique.

Presque à la même période, le *parallélisme d'instructions* atteignait également ses limites. Il était de plus en plus compliqué d'extraire du parallélisme d'un flux d'instructions. En 2004, la version *Prescott* du processeur *Intel Pentium 4* intégrait un plus long pipeline, destiné à accroître le rendement des opérations effectuées. Ses 31 *étages* permettaient également d'atteindre de plus hautes fréquences d'horloge. Ce pipeline divisait les instructions en un plus grand nombre de sous-instructions élémentaires dont chacune pouvait fonctionner à une cadence plus élevée. Hélas les performances de nombreux codes de calcul étaient dégradées, notamment à cause de l'association des nombreux étages à d'autres fonctionnalités du *parallélisme d'instructions*, comme les exécutions spéculatives liées aux instructions de branchement. Lors d'une mauvaise prise de décision par anticipation, les efforts accumulés dans le pipeline sont abandonnés afin de réorienter les calculs vers une solution correcte. La longueur du pipeline rallonge ainsi le délai de complétude des premières instructions, entraînant finalement des performances mitigées. *Intel* prévoyait également une élévation de la fréquence de fonctionnement de cette microarchitecture jusqu'à 10 GHz. Les courants de fuite à l'origine de la fin de la théorie de Dennard n'ont permis d'atteindre qu'une fréquence de 3,8 GHz, aux prix d'une élévation importante de l'enveloppe thermique. Cette dernière avoisinait les 100 Watts à dissiper, contre moins de la moitié pour les processeurs *Pentium 3* de génération précédente. Le successeur du *Pentium 4*, lequel devait être commercialisé en 2005, a été abandonné à cause de son enveloppe thermique trop élevée de 150 Watts [32].

Une nouvelle façon d'améliorer les performances des processeurs était nécessaire. Il ne fallait ni compter sur une nette élévation de la fréquence d'horloge, afin de conserver une enveloppe thermique acceptable, ni attendre des progrès significatifs du *parallélisme d'instructions*. Le solution adoptée dès 2005 fut d'intégrer l'équivalent de plusieurs processeurs, appelés *cœurs de calcul*, sur un même circuit intégré[20]. En réduisant leur fréquence de fonctionnement, il était possible de combiner les capacités de calcul de plusieurs processeurs sur une puce, sans augmenter la puissance thermique à dissiper. Leur exploitation est ainsi comparable au recours à plusieurs processeurs dans un même nœud de calcul, en suivant un mode de fonctionnement de type *MIMD*. Ces *processeurs multi-cœurs* peuvent cependant comporter des circuits particuliers, afin de faciliter la communication entre les cœurs de calcul et participent à la complexification de la mémoire des machines, laquelle sera abordée dans le prochain chapitre.

1.2.3 Les accélérateurs massivement parallèles

Des matériels spécialisés sont parfois employés pour accélérer certains traitements. Basés sur des circuits intégrés dédiés, ces accélérateurs sont plus efficaces pour certaines fonctionnalités que les processeurs généralistes (CPU[21]). Ces coprocesseurs sont rattachés à une *carte fille* en tant qu'extension d'un nœud de calcul ou intégrés dans la même puce que le *CPU* en formant un processeur accéléré appelé ***APU***[22]. Autrefois, les coprocesseurs pouvaient être directement connectés à la carte mère, comme c'est toujours le cas pour le *CPU*. Le recours à ces puces spécialisées est relativement ancien puisque des coprocesseurs étaient déjà employés dans les années 1960 avec la machine *Solomon* [33] afin d'améliorer les capacités de traitement vectoriel.

De nos jours, les accélérateurs sont majoritairement dérivés de processeurs graphiques (***GPU***[23]). Des variantes peuvent être dénombrées, telles que les coprocesseurs possédant une microarchitecture plus proche des CPUs, mais intégrant beaucoup plus de cœurs de calcul que ceux-ci (***MIC***[24] ou *Intel Xeon Phi*). Les accélérateurs actuels détiennent essentiellement une

[20] *Die* en anglais.
[21] **CPU :** *Central Processing Unit*.
[22] **APU :** *Accelerated Processing Unit*.
[23] **GPU :** *Graphics Processing Unit*.
[24] **MIC :** *Many Integrated Core*.

microarchitecture massivement multi-cœurs, c'est à dire intégrant plusieurs dizaines ou centaines de cœurs de calcul[25] comme le démontre la **figure 1.4a**. Chacun de ces cœurs possède des capacités de calcul vectoriel accrues grâce à de larges unités ou à l'emploi de plusieurs cœurs légers fonctionnant en mode *SIMD*. Tout comme les processeurs généralistes multi-cœurs, ces accélérateurs combinent donc des mécanismes de types *MIMD* et *SIMD*, mais permettent d'atteindre des capacités de traitement bien plus avantageuses au regard de la consommation électrique (cf. **figure 1.4b**). Les principaux coprocesseurs sont présentés dans cette sous-section.

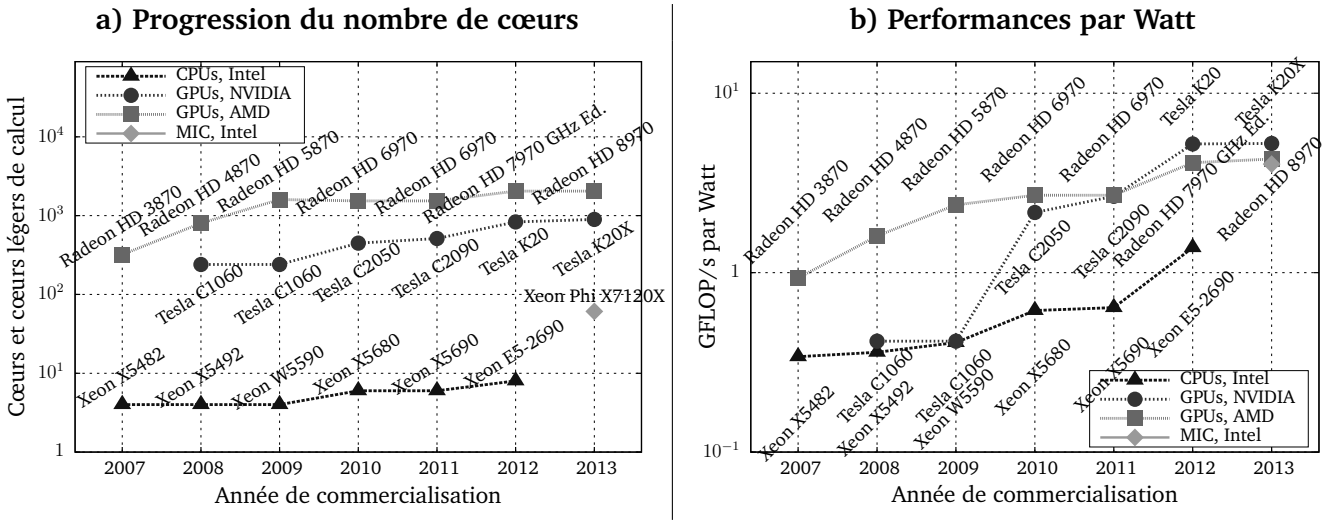


FIGURE 1.4 – Caractéristiques de calcul en double précision des CPUs, GPUs et MIC *(source [34])*.

Les générations antérieures d'accélérateurs et alternatives

Les ***FPGAs***[26] ont été conçus par l'entreprise Xilinx en 1985. Contrairement aux processeurs généralistes ou aux puces dédiées à des applications spécifiques (ASICs[27]), la structure de leur circuit n'est pas figée. Les FPGAs possèdent des blocs logiques qui peuvent être reconfigurés matériellement au moyen d'un langage de description d'agencement électronique tels que *VHDL*[28] ou *Verilog*. Le câblage de ces blocs peut être modifié à volonté, permettant ainsi d'adapter le circuit pour effectuer des traitements spécifiques. Ils sont généralement moins performant que les puces ASICs, mais du fait de leur capacité de reprogrammation, ils servent souvent à concevoir ces derniers. En 2004, des FPGAs ont été intégrées à la machine *Cray XD1* aux côtés de processeurs généralistes *AMD Opteron* et peuvent entre autres servir à accélérer des opérations d'algèbre linéaire [35, 36]. La machine *Maxwell* a été élaborée à l'université d'Édinburgh plus tard, en 2007, pour du calcul généraliste. Elle est quant à elle constituée d'un assemblage de 64 FPGAs [37]. Cependant, en plus d'être très onéreux, la programmation des FPGAs reste complexe. Peu de machines de production les ont adoptés pour du calcul intensif généraliste. Les progrès dans le domaine du *calcul à haute performance reconfigurable* (HPRC[29]) pourraient toutefois les amener sur le devant de la scène dans les années avenir.

[25] *Many-Cores* en anglais.
[26] **FPGA :** *The Field-Programmable Gate Arrays*.
[27] **ASIC :** *Application-Specific Integrated Circuit*.
[28] **VHDL :** *VHSIC Hardware Description Language*.
[29] **HPRC :** *High Performance Reconfigurable Computing*.

Les cartes accélératrices de l'entreprise *Clearspeed* [38] intègrent une puce **ASIC** pour accélérer des traitements dans le domaine du calcul à haute performance. En novembre 2006, le supercalculateur japonais *Tsubame*, alors 9^e au classement *TOP500*, embarque 360 de ces accélérateurs pour accroître les performances de cette machine de 24% en élevant la consommation électrique de seulement 1%. Depuis 2008, un processeur spécialisé *CSX700*, constitué de 192 cœurs légers, est capable de fournir une puissance de calcul théorique de 96 GFLOP/s pour 25 Watts de consommation lorsqu'il est associé à une capacité de 2 Go de mémoire sur une carte accélératrice. Ces processeurs ne sont cependant pas commercialisés en grand nombre ce qui a tendance à les rendre financièrement moins abordables que d'autres solutions. Depuis 2009, l'entreprise Clearspeed cherche à étendre son offre au secteur des systèmes embarqués, et plus particulièrement aux projets qui nécessitent une puissance de calcul substantielle.

En 2005, *Sony*, *Toshiba* et *IBM* dévoilent le processeur ***Cell*** [39]. Il s'agit d'une puce hétérogène embarquant des cœurs de calcul de différents types. Cette architecture peut être qualifiée d'*APU* puisqu'un cœur généraliste (PPE[30]) est épaulé par huit cœurs de calcul spécialisés (*SPE*[31]) comportant des jeux d'instructions vectorielles. Chaque *SPE* possède une mémoire locale de faible capacité pour permettre le déport des données et des instructions de calcul. Les programmeurs sont contraints à gérer explicitement ces mémoires, ce qui participe à rendre difficile l'exploitation de ce processeur. Le *Cell* équipe la *PlayStation 3*, une console de jeux vidéo commercialisée par Sony, ce qui le rend plus attractif financièrement pour le domaine du calcul scientifique. En juin 2008, *Roadrunner* [40] était la machine la plus puissante du monde en embarquant à la fois $6\,480$ processeurs généralistes multi-cœurs de marque *AMD* et $12\,960$ processeurs Cell. C'est le premier supercalculateur à dépasser le million de milliards d'opérations par seconde (PFLOP/s) au *TOP500*. Sa capacité de calcul était plus de deux fois supérieure à la deuxième machine et détenait une des meilleures efficacités énergétiques du classement. Le Cell cédera progressivement sa place aux processeurs graphiques dans le domaine du calcul à haute performance.

Les GPUs

Les premiers processeurs graphiques étaient employés pour piloter un écran et décharger le CPU de certaines opérations qui permettaient un affichage à deux dimensions. Dans les années 1990, des cartes d'extensions ont fait leur apparition pour accélérer cette fois-ci les rendus à trois dimensions. La série *Voodoo*, élaborée par la société *3dfx Interactive*, fut très populaire pour améliorer la qualité des rendus graphiques dans les jeux vidéos. A cette époque, les cartes graphiques 2D étaient vendues séparément de ces cartes accélératrices 3D, chacune d'entre elles embarquait ainsi un processeur spécialisé. En 1996, l'entreprise canadienne ATI (acquise par AMD en 2006) commercialise la *ATI Rage*, première carte intégrant les deux capacités 2D et 3D. En 2000, le concurrent Nvidia rachète *3dfx Interactive* après avoir dévoilé le processeur graphique *Geforce 256*. Le terme de *GPU* [41] est alors évoqué pour la première fois et désigne un composant complexe capable d'effectuer de nombreux traitements visuels. Ces premiers GPUs étaient peu exploités en dehors des jeux vidéo et de l'infographie. Ils intégraient des unités des traitements matériels très spécialisées (FPP[32]). Grâce à plusieurs étapes, ces unités spécifiques permettaient plusieurs modifications visuelles, avant de les combiner pour former l'image finale. Cette technique n'offrait cependant pas une souplesse suffisante pour s'adapter efficacement aux évolutions des effets graphiques.

[30] **PPE :** *PowerPC Processing Element*.
[31] **SPE :** *Synergistic Processing Element*.
[32] **FPP :** *Fixed Function Pipeline*.

En 2001, avec la carte *Nvidia Geforce 3*, les GPUs évoluent pour supporter des techniques de post-traitement plus sophistiquées, en s'adaptant aux programmes par *shaders* qui permettent de mieux paramétrer une partie des opérations de rendu. Une plus grande souplesse était ainsi permise en gagnant en généricité par l'emploi d'unités de traitement matérielles propres aux trois types de "*shaders*" usuels[33]. La puissance de calcul des GPUs devient de plus en plus conséquente grâce à une grande capacité de traitements parallèles. De plus, l'emploi massif des GPUs dans les consoles de jeu et dans les ordinateurs a contribué à fortement réduire leur coût. Les scientifiques commencent à s'y intéresser pour accélérer des traitements dans d'autres domaines. La pratique GPGPU[34] émerge [42]. Elle consiste a détourner l'utilisation spécialisée d'un GPU vers du calcul générique.

Depuis, les processeurs graphiques ont unifié les unités de traitement *shaders* en permettant une meilleure gestion des ressources matérielles et une plus grande souplesse d'exploitation. Les GPUs adoptent des architectures massivement parallèles en embarquant une capacité de traitements conséquente, regroupée dans des unités de calcul multi-cœurs. Chaque constructeur adopte des approches diverses. Les architectures d'ATI/AMD intègrent des cœurs à larges instructions vectorielles tandis que celles de Nvidia comportent de nombreux cœurs légers exploités de manière *SIMD*. Ces unités de traitement, d'élaborations distinctes, sont appelées *processeur de flux* (SP[35]). Des modèles de programmation plus accessibles ont permis d'accroître davantage le recours aux GPUs pour du calcul générique [43]. Ils intègrent des améliorations qui les rendent plus attractifs pour le calcul scientifique tels que le support du calcul à double précision, des mémoires à correction d'erreurs matérielles (ECC[36]) ou des communications complètement asynchrones avec la mémoire centrale. Depuis 2010, plusieurs noyaux de calcul peuvent s'exécuter de manière simultanée sur des unités multi-cœurs distinctes avec l'architecture *Fermi* [44, 45] de *Nvidia*. Les GPUs possèdent désormais un mode de fonctionnement mixte *MIMD* et *SIMD*. En novembre 2010, la machine chinoise *Tianhe-1A* devient le premier supercalculateur au classement *TOP500* en reposant sur $14\,336$ CPUs *Intel* et $7\,168$ GPUs *Nvidia*.

Les MICs

En 2009, *Intel* annonce le processeur *Larrabee* [46], une puce semblable à un GPU basée sur de nombreux cœurs de calcul similaires à ceux employés dans les CPUs généralistes *x86/64*. Cependant, la solution n'est pas viable commercialement en raison de ses performances décevantes. La microarchitecture a été améliorée en s'appuyant également sur d'autres projets[37] pour aboutir aux MICs. *Intel* ne vise désormais que le marché du calcul à haute performance avec son premier produit commercialisé en 2012, baptisé *Xeon Phi*. Ce dernier embarque un système d'exploitation, ce qui peut lui octroyer une plus grande autonomie vis-à-vis des CPUs. Il est également composé de 61 cœurs de calcul[38], connectés en anneau et comportant chacun de larges instructions vectorielles de 512 bits. Cette solution est censée offrir une compatibilité des logiciels développés pour des CPUs, là où les GPUs requièrent une réécriture complète du code. *Intel* mise ainsi sur la productivité et la programmabilité en permettant un portage fonctionnel rapide. En juin 2013, *Tianhe-2* est la machine la puissante au monde en agrégeant seize-mille nœuds de calculs, comportant chacun deux CPUs et trois Xeon Phi [12] d'*Intel*. Ce supercalculateur totalise ainsi plus de trois millions de cœurs de calcul.

[33] *Vertex shaders*, *Geometry shaders* et *Pixel shaders*.
[34] **GPGPU :** *General Purpose Graphics Processing Unit*.
[35] **SP :** *Stream Processors*.
[36] **ECC :** *Error Checking and Correcting*.
[37] *Teraflops Research Chip* [47] et *Intel Single-Chip Cloud*.
[38] Dont un cœur dédié à l'OS.

1.3 Les supercalculateurs contemporains

Les machines les plus puissantes, élaborées après 2008, sont pour la plupart de classe "*pétaflopique*" (10^{15} FLOP/s), c'est-à-dire qu'elles sont capables d'effectuer plus d'un million de milliards d'opérations par seconde [48]. La machine du *CEA-DAM Tera-100* (cf. **figure 1.5a**) est un exemple de ce genre de supercalculateur. Produite par la société française *BULL*, elle est mise en service en 2010 pour 1,254 PFLOP/s théorique. Elle se positionne sixième au *TOP500* en novembre 2010 en atteignant 1,05 PFLOP/s et était également la machine la plus puissante d'Europe. *Curie* (cf. **figure 1.5b**) a été financé par le *Grand Équipement National de Calcul Intensif* (GENCI) et est hébergé dans les locaux du *Très Grand Centre de Calcul* au CEA. C'est un supercalculateur ouvert aux scientifiques européens dans le cadre de la participation aux infrastructures PRACE[39]. Il a également été installé par *Bull* et accède à la neuvième place du *TOP500* avec 1,359 PFLOP/s en juin 2012.

a) Tera-100 **b) Curie**

FIGURE 1.5 – Exemples de deux supercalculateurs *pétaflopiques* français.

L'élaboration d'un supercalculateur petaflopique requiert un effort particulier d'intégration pour contenir une forte densité de composants, tout en assurant une dissipation thermique adéquate. Comme l'illustre la **figure 1.6**, Tera-100 est une machine organisée en une grappe de 4 370 nœuds de calcul, répartis dans 220 armoires. Chaque nœud contient 4 processeurs octo-cœurs. Tera-100 totalise ainsi 139 840 coeurs de calcul, 300 To de mémoire vive et possède un stockage d'une capacité de 20 Po.

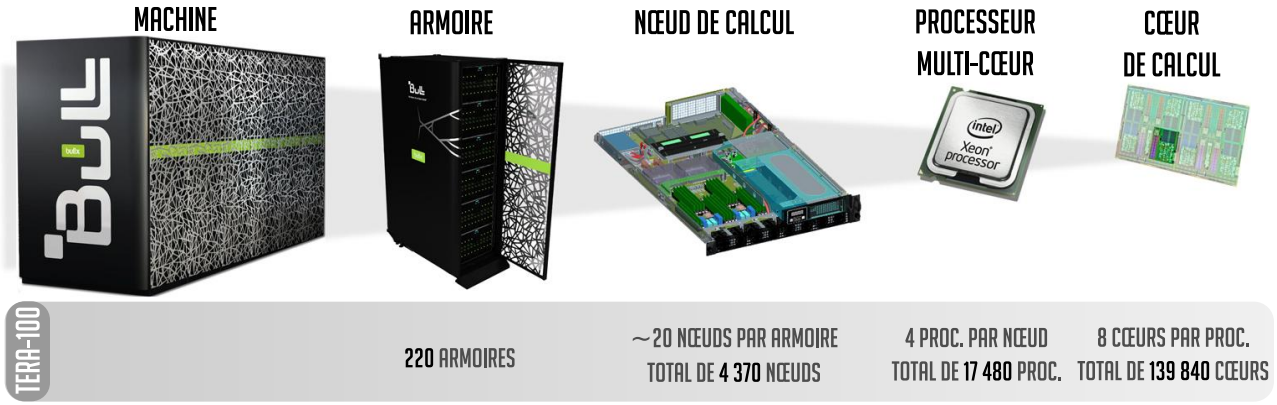


FIGURE 1.6 – Intégration des ressources de calcul en grappe. Exemple de *Tera-100*.

[39]**PRACE** : *Partnership for Advanced Computing in Europe*.

1.3.1 L'émergence des grappes de calcul hétérogènes

Des supercalculateurs intégrant des accélérateurs matériels apparaissent à partir de 2006 et constituent désormais une part non négligeable des machines classées au *TOP500* (cf. figure 1.7).

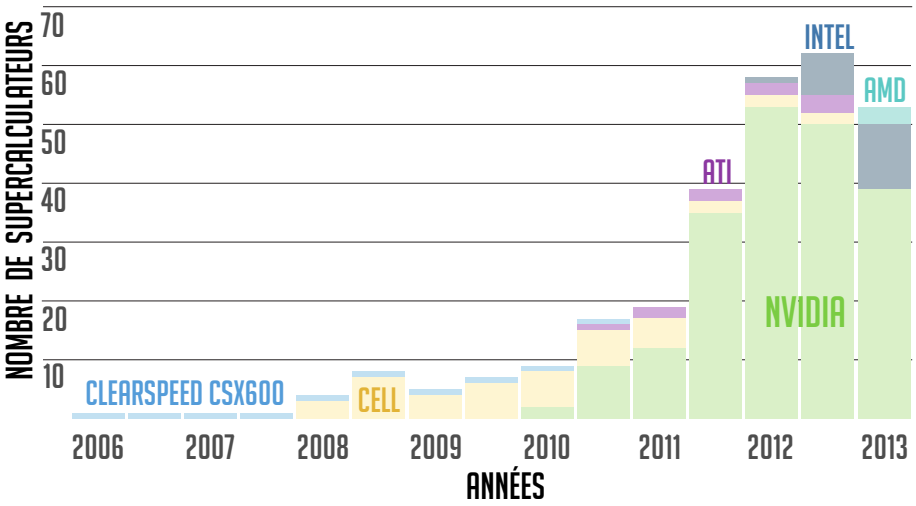


FIGURE 1.7 – Nombre de supercalculateurs du *TOP500* comportant des accélérateurs *(source : top500.org)*.

En 2011, une partition de 192 nœuds hétérogènes est rattachée à Tera-100. Chaque nœud est alors constitué de deux CPUs quadricœurs *Intel* et deux GPUs *Nvidia*. Les machines hétérogènes agrègent ainsi plusieurs technologies développées tout au long de l'histoire des superordinateurs. Les processeurs réalisent des opérations de type *SISD* et *SIMD*. Leurs assemblages dans un nœud, ainsi que la démultiplication du nombre de cœurs, permettent un fonctionnement *MIMD à mémoire partagée*, alors que l'ensemble de tous les nœuds repose sur un mécanisme *MIMD à mémoires distribuées*. Les accélérateurs matériels, quant à eux, combinent *MIMD à mémoire partagée* et *SIMD* avec leurs cœurs légers ou leurs instructions vectorielles. La figure 1.8 résume l'imbrication de ces diverses architectures dont chacune nécessite généralement un modèle de programmation distinct, plus adapté aux contraintes liées à leur exploitation.

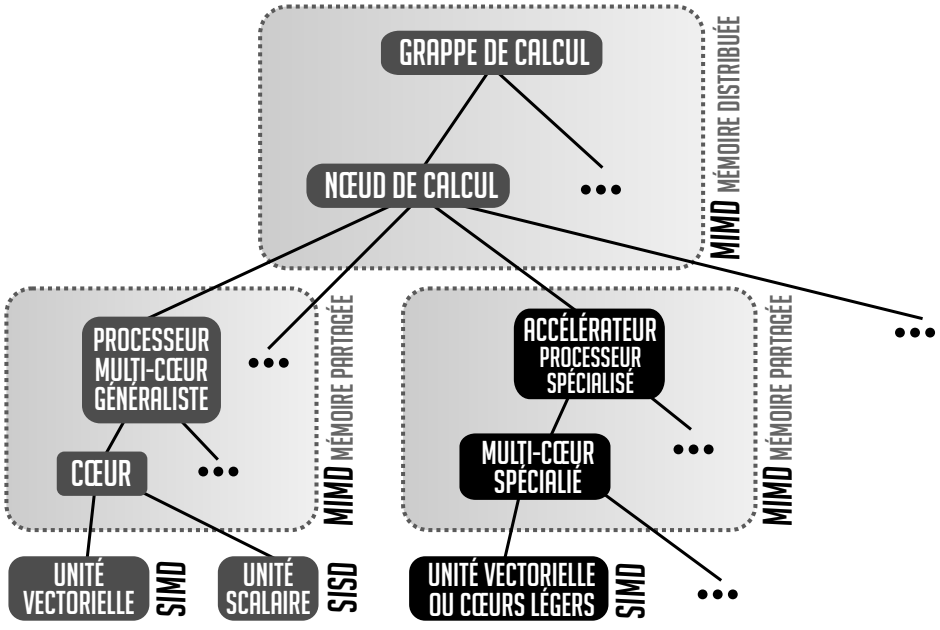


FIGURE 1.8 – Supercalculateur en grappes avec nœuds de calcul hétérogènes. Types de traitements de chaque assemblage selon la taxonomie de Flynn.

1.4 Les supercalculateurs de prochaine génération

Les prochaines générations évolueront, si les avancées technologies le permettent, vers des systèmes "*exaflopiques*" (10^{18} FLOP/s) d'ici l'horizon 2020. Il existe deux visions différentes de l'évolution des architectures. Un article paru en 2008 dans l'*Electronic Engineering Times* confronte les opinions de plusieurs architectes matériels sur l'avenir des processeurs multi-cœurs [49]. Chuck Moore, inventeur du langage *Forth* et détenteur de plusieurs brevets dans la conception de processeurs à basse consommation [50], suggère que les ordinateurs devraient être conçus comme des téléphones portables, c'est-à-dire employer une variété d'unités de calcul spécialisées. Atsushi Hasegawa, ingénieur en chef chez le fabricant de semi-conducteurs japonais *Renesas*, partage la même idée du besoin à recourir à des puces hétérogènes. Un point de vue radicalement différent est adopté par Anant Agarwal, architecte des premiers multi-processeurs à cohérence de caches à l'origine du projet *Beowulf* et fondateur de l'entreprise *Tilera* [51] spécialisée dans la conception de processeurs multi-cœurs pour le domaine des systèmes embarqués. Selon lui, les processeurs multi-cœurs devraient être constitués de cœurs généralistes et homogènes afin de conserver un modèle de programmation suffisamment simple.

Dans le domaine du calcul à haute performance, il est pour le moment difficile de prédire quelle approche permettra aux architectures massivement parallèles d'atteindre l'*Exascale* dans les meilleures conditions de consommation énergétique et de programmabilité. Les sous-sections suivantes étudient chacune de ces deux directions.

1.4.1 Piste des architectures hétérogènes

Les puces hétérogènes de type *APU* misent sur la spécialisation des opérations. Elles embarquent sur la même puce et un processeur capable de traiter efficacement les portions de code séquentielles et un autre processeur plus adapté à des opérations massivement parallèles. Par rapport à une utilisation de composants disjoints, cette association présente le principal avantage de réduire les surcoûts induits par les communications entre les différentes ressources de calcul. Le bus d'interconnexion est actuellement un des principaux goulots d'étranglement lorsqu'une grande quantité de données doit être déplacée depuis la mémoire centrale vers une mémoire embarquée dans un accélérateur déporté. Une association au sein d'une même puce permet de connecter un GPU et un CPU à la même mémoire et d'atténuer cette contrainte de communication. En 2014, plusieurs projets aboutiront à des *APUs* plus performants que ceux proposés dans le domaine des systèmes embarqués :

— La nouvelle console de jeux *PlayStation 4*, annoncée en février 2013, embarque un *APU AMD* de dernière génération. Ce dernier est composé d'un CPU multi-cœurs et d'un GPU dont la puissance théorique combinée atteint les $1{,}87$ TFLOP/s en simple précision. La technologie de la mémoire associée est en faveur d'une large bande passante avec de la mémoire graphique à 176 Go/s, mais n'offre en contrepartie qu'une capacité limitée de 8 Go [52].

— Le projet *Denver* d'*Nvidia* annoncé en 2011, a pour ambition de s'affranchir de l'architecture *x86* et de proposer des APUs se basant sur un CPU ARM et d'un puissant GPU. En plus du domaine des systèmes embarqués avec la puce *Tegra 5*, ces processeurs hétérogènes pourraient à terme viser le marché des ordinateurs personnels et des supercalculateurs [53]. La version *5.5* plate-forme de programmation *CUDA* supporte en particulier la compilation sur processeurs ARM pour les portions de code destinées aux CPUs.

1.4.2 Piste des architectures homogènes

L'approche homogène repose sur une juxtaposition d'unités ou cœurs de calcul similaires. Un cœur pris séparément possède une puissance suffisante pour traiter des sections de codes séquentielles. Le recours à la combinaison de tous les cœurs servirait à accélérer les portions massivement parallèles. Ces ressources de calcul uniformes permettent théoriquement de faciliter leur programmation. De nouvelles architectures massivement parallèles et homogènes sont dès à présent disponibles ou verront le jour dans un futur proche :

— Les MICs d'*Intel* seront déclinés sous deux formes différentes avec la prochaine architecture de nom de code *Knight Landing*. La première se basera sur le schéma classique d'un accélérateur déporté et le seconde sera employée en tant que principal constituant d'un nœud de calcul pour remplacer les *CPUs* traditionnels et se débarrasser de la contrainte de bande passante réduite imposée par le bus d'interconnexion [54].

— Les processeurs *Kalray MPPA*[40] 256 sont pour le moment destinés aux systèmes embarqués. Ils intègrent 256 cœurs de calcul homogènes interconnectés en 16 îlots de 16 cœurs, offrant une puissance théorique de 230 GFLOP/s en simple précision pour une consommation réduite à moins de 40 Watts [55]. Chaque îlot possède 4 groupes constitués 4 cœurs et d'une mémoire locale programmable. Le processeur peut être connecté à de la mémoire classique de type DDR3. Cette architecture s'articule donc autour de deux niveaux de mémoire accessibles par les développeurs.

— Les puces *Tilera TilePro64* [56] embarquent 64 cœurs généralistes identiques, interconnectés par un réseau maillé [41] et offrant une consommation réduite d'une 20^e de Watts. Le projet du MIT *Angstrom* [57, 58], financé par l'agence pour les projets de recherche avancée de défense aux États-Unis (DARPA[42]), vise à étendre cette architecture à plusieurs milliers de cœurs de calcul pour concevoir des machines exaflopiques.

1.5 Lien entre la puissance de calcul et la consommation électrique

Une meilleure efficacité énergétique des architecture permet d'assembler un plus grand nombre de composants pour un budget de consommation donné. En particulier, les FLOP/s par Watt (FLOP/s/Watt) est une métrique primordiale pour l'élaboration des architectures émergentes tout comme cela peut l'être pour le domaine des systèmes embarqués. La **figure 1.8** présente l'évolution de cette métrique entre 2008 et début 2013 pour la première machine et la moyenne des cinq machines les plus performantes du classement *TOP500*.

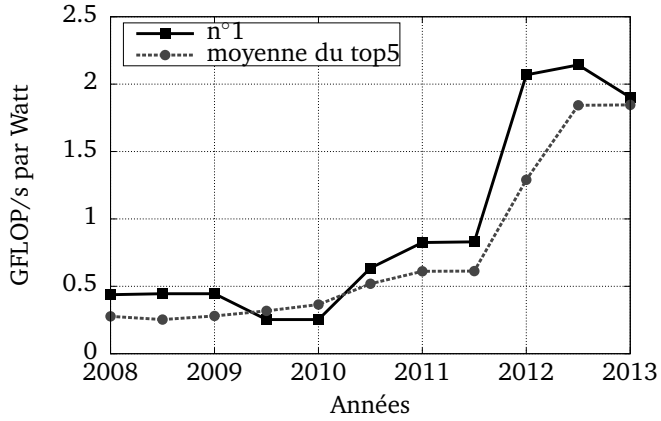


FIGURE 1.9 – Efficacité énergétique de la première machine et de la moyenne des cinq premières machines du classement *TOP500*.

[40] **MPPA :** *Multi-Purpose Processor Array*.
[41] *Mesh* en anglais.
[42] **DARPA :** *Defense Advanced Research Projects Agency*.

1.5.1 Similitudes avec le domaine de l'embarqué

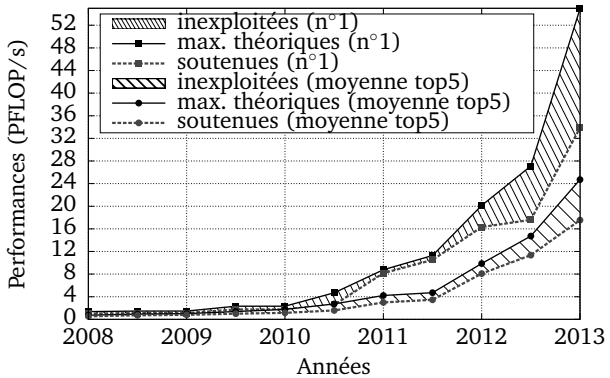
Une convergence semble engagée entre les architectures du domaine de l'embarqué et les machines à hautes performances. Dans le premier cas, l'autonomie est une priorité. Il s'agit d'élaborer des solutions matérielles qui conservent des niveaux de consommation suffisamment bas imposés par les capacités d'alimentation des batteries, tout en améliorant les performances de traitement. Dans le domaine des supercalculateurs, l'objectif est d'obtenir les meilleures performances possibles. La consommation électrique est également un forte contrainte, puisque les machines sont conçues pour une enveloppe énergétique donnée. Cette dernière est déterminée par le budget alloué au coût de fonctionnement et les capacités d'alimentation du bâtiment hébergeant la machine.

Depuis 2008, *Nvidia* commercialise des processeurs pour le domaine des systèmes embarqués appelés *Tegra*. En 2011, la puce hétérogène *Tegra 3* possède trois types de cœurs différents : des cœurs issus d'un processeur généraliste *ARM*, des processeurs de flux associés à un *GPU* allégé et un cœur spécialisé surnommé "*compagnon*" pour la gestion des tâches à basse consommation. L'architecture *Logan* sera en 2014 la cinquième version du processeur *Tegra* et intégrera une architecture de GPU employée dans le *HPC*. *Tegra 5* embarquera alors un processeur quadri-cœurs *ARM*, un GPU basé sur l'architecture *Kepler* et un cœur compagnon. Il en est globalement de même avec les *APUs AMD Fusion* qui intègrent des processeurs généralistes et des GPUs *AMD Radeon*. *Intel* a emprunté le chemin inverse en s'appuyant sur l'architecture des processeurs à basse consommation *Intel Atom* afin de concevoir les cœurs de calcul qui composent les accélérateurs *MICs*. Une convergence des technologies peut être aussi constatée, dirigée par les performances par Watt.

1.5.2 Puissance de calcul effective

La **figure 1.10a** révèle la part des performances inexploitées en comparant la puissance théorique et la puissance soutenue sur le test *Linpack*. En juin 2013, la machine la plus puissante du *TOP500 Tianhe-2* atteint $61{,}67\%$ des performances maximales théoriques ($21{,}04$ PFLOP/s inexploitables) contre 71% pour la moyenne des cinq premières ($7{,}15$ PFLOP/s inexploitables).

a) Puissances de calcul théoriques et soutenues



b) Consommation électrique

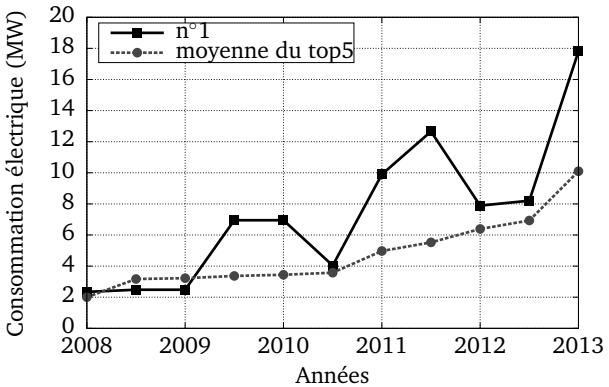


FIGURE 1.10 – *TOP500* entre juin 2008 et juin 2013. **a)** Puissances de calcul théoriques et puissances atteintes moyennées sur les cinq premières machines et avec la première. **b)** Consommation électrique en MW des mêmes catégories.

Entre 2008 et 2013, l'efficacité moyenne des cinq supercalculateurs les plus puissants au monde varie entre 58% et 82%. Dans l'absolu, cela représente une perte conséquente de la capacité de traitement. À titre de comparaison, les 21 PFLOP/s inexploités de *Tianhe-2* sont supérieurs à la puissance de calcul soutenue par la machine américaine *Titan* ($17{,}59$ PFLOP/s) classée première sept mois auparavant, en novembre 2012.

La **figure 1.10b** démontre que les machines ont tendance à consommer plus d'électricité en passant d'une moyenne de 2 MW en 2008 pour les cinq premières machines à $10{,}10$ MW en 2013. En rapprochant les deux **figures 1.10**, la part de la puissance électrique consommée qui est employée pour alimenter la proportion inexploitable des machines croît exponentiellement. Le rapport $\frac{\textit{performances effectives}}{\textit{consommation électrique}}$ est devenu capital pour concevoir de puissants supercalculateurs en limitant l'augmentation des dépenses énergétiques.

Discussion

En résumé, la maîtrise des machines nécessite d'exploiter toutes les ressources de traitement intégrées à la hiérarchie des grappes de calcul. Celles-ci peuvent être issues de diverses technologies. Elles nécessitent d'extraire du parallélisme en s'appuyant en partie sur divers modèles de programmation qui seront étudiés dans le **chapitre 3** (page 55). Cependant, même avec un code hautement optimisé comme le test de performance *Linpack*, une part de la puissance de calcul des machines reste inexploitable. La cause réside principalement dans les coûts d'accès aux données dans les diverses mémoires de la grappe de calcul. Les unités de traitement ne sont pas maintenues suffisamment occupées pour atteindre des performances plus proches de la puissance de calcul théorique maximale. Avec des programmes moins optimisés, les déplacements de données deviennent généralement prépondérants. Ainsi, en plus de l'expression du parallélisme, la façon dont on accède aux données est un facteur primordial pour atteindre de hautes performances. Ce degré de complexité supplémentaire est étudié dans le chapitre suivant.

Chapitre 2

Les performances assujetties au chargement des données

> *“ … de diviser chacune des difficultés que j’examinerois, en autant de parcelles qu’il se pourroit, et qu’il seroit requis pour les mieux résoudre. ”*
>
> René Descartes, Le *Discours de la Méthode*, 1637

Par définition, les processeurs effectuent des opérations qui modifient des données. L’état de ces dernières est physiquement stocké dans des mémoires. À l’heure actuelle, le facteur matériel certainement le plus décisif permettant d’atteindre de hautes performances de calcul réside dans la rapidité d’accès à ces données par les unités de traitement des processeurs. Les cycles de chaque processeur passés à attendre les données sont des opportunités de calcul perdues, menant rapidement à une sous exploitation des unités de traitement, qui se traduit par une durée totale d’exécution des programmes plus longue que nécessaire. Hennessy et Patterson évaluent de 20 à 40% les instructions faisant référence à la mémoire dans la plupart des programmes [59]. Ces proportions, bien qu’approximatives, révèlent l’importance de ces accès et la nécessité pour les constructeurs d’élaborer des stratégies pour minimiser les temps d’attente liés au rapatriement des données. Les mécanismes qui régissent la mémoire des machines actuelles ont évolué et se sont complexifiés, afin de contourner ou de s’adapter à des contraintes de natures historiques, technologiques ou liées à la difficulté de programmation.

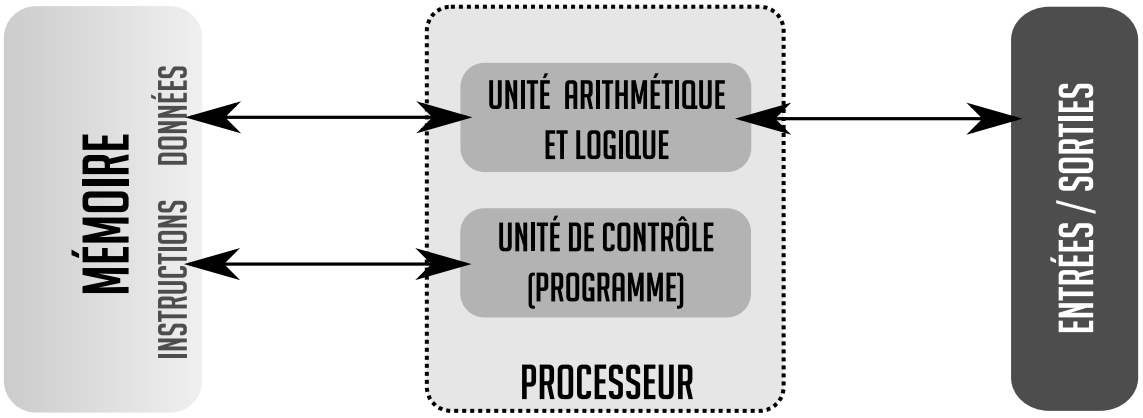


FIGURE 2.1 – Modèle de *Von Neumann* : les données et les instructions sont stockées dans la même mémoire.

— D'un point de vue historique tout d'abord, les machines que nous exploitons reposent sur le modèle de *Von Neumann* dont la principale caractéristique est le stockage dans la mémoire des instructions et des données traitées par le programme (cf. **figure 2.1**). La sollicitation de la mémoire est ainsi accentuée, puisqu'elle doit fournir des informations de deux natures différentes pour permettre les calculs.

— De plus, en matière de facilité d'exploitation, les machines asymétriques étaient relativement complexes à programmer du fait des mémoires séparées que devaient gérer les programmeurs. Elles ont été supplantées par les machines symétriques, lesquelles s'articulent autour d'une mémoire partagée, accessible par tous les processeurs du nœud de calcul (cf. **chapitre 1.2.1**, page 20). La mémoire est ainsi devenue une ressource centrale dont chaque donnée peut être accessible et modifiable de manière concurrente. Cela implique de fortes contraintes afin d'assurer la cohérence de ces données [60] ; des choix architecturaux pouvant se transformer en véritables goulots d'étranglement.

— En outre, des limitations technologiques ont freiné l'évolution des performances des mémoires. Les fondeurs et les constructeurs de machines sont confrontés à une barrière technologique baptisée *mur mémoire* [61] faisant référence à la différence d'évolution de la rapidité des mémoires face à celle des processeurs. La puissance de calcul des processeurs continue à augmenter de manière exponentielle, grâce notamment à la multiplication des cœurs de calcul (cf. **chapitre 1.2.2**, page 21). Les temps d'accès de la mémoire s'améliorent nettement moins rapidement. Pourtant, cette dernière doit globalement être en mesure de fournir des quantités de données de plus en plus conséquentes [62].

— Plus récemment, en termes d'évolution des ressources de calcul, des accélérateurs matériels ont été rajoutés dans les machines (cf. **chapitre 1.2.3**, page 23). Ces matériels spécialisés embarquent leur propre mémoire et nécessitent de communiquer avec la mémoire centrale. Les échanges de données entre la mémoire du nœud de calcul et celle des accélérateurs sont explicitement déclenchés par l'utilisateur, ce qui réintroduit un aspect asymétrique aux nouvelles architectures. Ces accélérateurs accroissent ainsi la demande en quantité de données que doit pouvoir fournir la mémoire centrale, tout en introduisant de nouvelles difficultés que les programmeurs sont contraints de surmonter.

Ainsi, des mécanismes ont été conçus pour alléger la pression sur la mémoire centrale. Certains d'entre eux, bien que très efficaces lorsque quelques conditions sont respectées, peuvent introduire des dégradations importantes de performance dans d'autres circonstances. Pris dans leur globalité, ces mécanismes mènent à une complexité croissante pour les programmeurs qui visent à concevoir des applications capables d'atteindre de hautes performances. En suivant le second précepte du *Discours de la Méthode* de R. Descartes, afin de mieux appréhender les problèmes inhérents aux mécanismes matériels, les principales sources de difficultés seront une à une décomposées dans les sections suivantes.

Ce chapitre étudie certains aspects matériels et leurs limites en s'appuyant sur l'architecture des machines actuelles. Il convient de mettre en lumière les causes des principaux problèmes et leurs impacts sur une gestion adéquate des données par le programmeur. Seules les mémoires volatiles, directement impliquées dans les calculs, sont considérées. Par conséquent, les mémoires persistantes permettant d'archiver les données, telles que les disques durs, bandes magnétiques et mémoires flash ne sont pas abordées. La première section fournit un tour d'horizon du système mémoire organisé d'imbrications d'éléments. Les parties suivantes se focalisent sur les grandes catégories qui composent la hiérarchie mémoire d'un nœud de calcul, à savoir la mémoire centrale, les mémoires locales et les mémoires déportées exploitées par les accélérateurs matériels.

2.1 L'organisation hiérarchique de la mémoire

Les constructeurs mettent au point des machines pouvant gérer une quantité de mémoire de plus en plus conséquente. Cette mémoire doit être en mesure de maintenir occupées les unités de calcul, en leur fournissant un flux de données suffisant. Or, la capacité de la mémoire, ainsi que ses performances ont un coût et contraignent les choix technologiques. Ainsi, l'élaboration du système mémoire est communément associée à un compromis entre plusieurs facteurs interdépendants : le prix, la capacité de stockage et les performances d'accès aux données. Les machines actuelles répondent à ces contraintes en imbriquant de manière hiérarchique des technologies de mémoires distinctes comme l'illustre la **figure 2.2**.

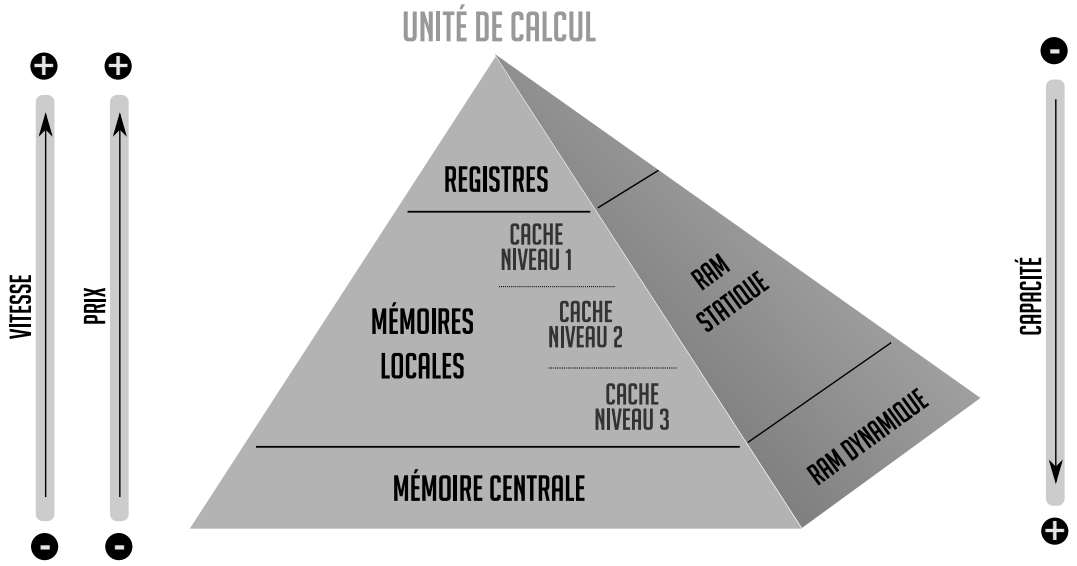


FIGURE 2.2 – Hiérarchie de la mémoire.

D'un point de vue matériel, deux types de mémoires volatiles sont essentiellement employés dans la conception des nœuds de calcul [63]. Schématiquement, la première catégorie, qualifiée de mémoire dynamique, nécessite un rafraîchissement électronique périodique afin de conserver les informations sur le support physique. Cette technologie possède une bonne densité de stockage, mais en contrepartie, des temps d'accès aux données (*latences*) plus longs. La seconde catégorie est associée aux mémoires dites statiques, où le rafraîchissement n'est pas utile, ce qui permet d'atteindre des latences plus courtes. Davantage de transistors sont nécessaires pour réaliser ce type de mémoire. Par conséquent, à finesse de gravure identique des transistors, moins de données peuvent être matériellement stockées qu'avec une mémoire dynamique.

Typiquement, la mémoire centrale d'une machine est basée sur la technologie dynamique en raison de sa plus grande capacité de stockage. Elle ne suffit pas à elle seule à maintenir un état d'occupation suffisant des unités de calcul à cause de ses temps d'accès relativement élevés. Des mémoires qualifiées de locales, implantées physiquement plus près des ressources de calculs, sont employées. Elles reposent quant à elles sur la technologie statique pour profiter de temps d'accès plus avantageux. Dans les années 90, des machines ont été conçues uniquement à partir de mémoires locales (architectures COMA[43]). Les données sont répliquées et déplacées au plus près des unités de calcul, au sein d'un réseau de mémoires locales. L'architecture est complexe, en particulier celle des mécanismes matériels permettant de maintenir un état cohérent des données

[43] **COMA :** *Cache Only Memory Access*.

répliquées [64]. La capacité de stockage étant relativement limitée pour maintenir des coûts acceptables, l'architecture n'est pour le moment plus employée. Désormais, les machines reposent plutôt sur une hiérarchie de mémoires dont les niveaux sont de plus en plus rapides au fur et à mesure que les données se rapprochent des unités de calcul.

La mémoire centrale, dont les temps d'accès sont relativement élevés dans cette imbrication en cascade, sert en quelque sorte de dépôt pour les données de calcul. Ses performances sont cruciales puisque l'augmentation du nombre de cœurs, et donc des copies locales, implique un besoin en données de plus en plus conséquent. Des mécanismes permettant de masquer une partie de ces latences élevées sont déployés. Par exemple, le préchargement de données permet de rapatrier par anticipation des données dans les mémoires locales. Des astuces de recouvrement permettent d'effectuer, au même instant, des calculs pendant que sont transférées des données qui seront prochainement utiles. Certains cœurs de calcul peuvent également gérer plusieurs flots d'instructions simultanément (SMT[44]), permettant entre autres, de mieux occuper les unités de calculs pendant que les données d'un autre flot sont transférées. Enfin, les concepteurs de machines ont amélioré la quantité de données (*bande passante*) pouvant être simultanément déplacée. Les données sont transférées en parallèle pour rapatrier de plus grosses quantités de données dans les mémoires locales.

Pour résumer, les architectures actuelles combinent les avantages des deux types de mémoires. Les données sont ainsi répliquées dans une imbrication de mémoires locales de plus en plus rapides. Les machines tendent ainsi à se complexifier et à intensifier la hiérarchisation des différents éléments, permettant une amélioration progressive des performances et un accroissement de la quantité des éléments à un coût acceptable. Malgré l'utilisation de mécanismes visant à masquer une partie des temps d'accès à la mémoire centrale, ces latences élevées restent pénalisantes. De hautes performances ne peuvent être atteintes qu'en réduisant les sollicitations de la mémoire centrale. Ainsi, les programmes de calcul doivent exploiter au maximum les mémoires les plus rapides, en s'appuyant sur les principes de localité, lesquels sont abordés dans les sections suivantes.

2.2 Imbrication et cohérence des mémoires locales

Les mémoires locales, telles que les *mémoires caches* (aussi appelées *mémoires tampons* ou *antémémoires*), ont pour objectif de fournir plus rapidement les données aux unités de calcul. Pour assurer une bonne efficacité, leur capacité est limitée. Ces mémoires effectuent des opérations à l'échelle de blocs dont les données sont stockées physiquement de manière contiguë. Dans les mémoires caches, ces blocs sont appelés *lignes de caches*. Lorsqu'une opération d'écriture ou de lecture est réalisée, une ligne de cache complète est rapatriée depuis la mémoire centrale. Lorsque la mémoire cache n'a plus assez d'espace disponible, une ligne doit être choisie et évincée pour être évacuée vers la mémoire centrale. La place ainsi libérée peut être occupée par une autre ligne.

Les mémoires caches misent sur les principes de localité des données. Lorsqu'une opération sur une donnée est réalisée, il y a de fortes chances pour qu'une autre donnée, physiquement proche, soit également modifiée (*localité spatiale*) et se trouve dans la ligne déjà rapatriée. De même, ces données chargées en mémoires caches peuvent être à nouveau modifiées dans le programme. Si ces opérations sont suffisamment rapprochées dans le temps, il est probable que ces données soient encore présentes en mémoire cache (*localité temporelle*), évitant ainsi d'aller les chercher à nouveau en mémoire centrale. Telles des poupées russes, différents niveaux de mémoires caches sont imbriqués comme l'illustre la **figure 2.3**.

[44] **SMT :** *Simultaneous Multi-Threading*.

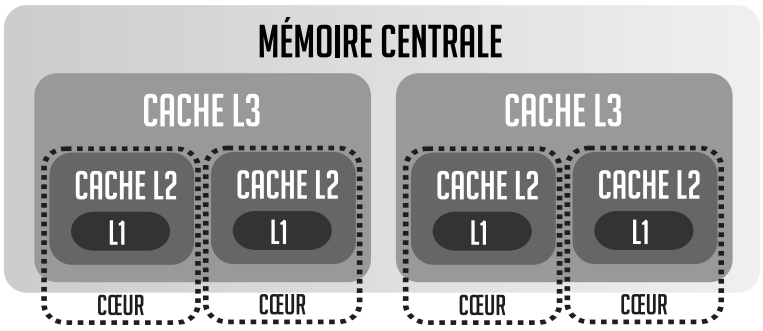


FIGURE 2.3 – Imbrication des mémoires.

Cette section présente quelques optimisations et phénomènes pénalisants propres à chaque niveau d'emboîtement. Pour simplifier les explications, seules des mémoires caches inclusives sont considérées. En d'autres termes, si une donnée est présente à un niveau, elle l'est aussi dans les niveaux plus lents.

2.2.1 Mémoires caches dédiées à un cœur de calcul

Il s'agit des niveaux les plus proches des unités de calcul et donc les plus rapides (cf. figure 2.4). Ces mémoires assurent le lien avec les unités de calcul via les registres. Afin de maximiser les performances, les principes de localité doivent être exploités au maximum.

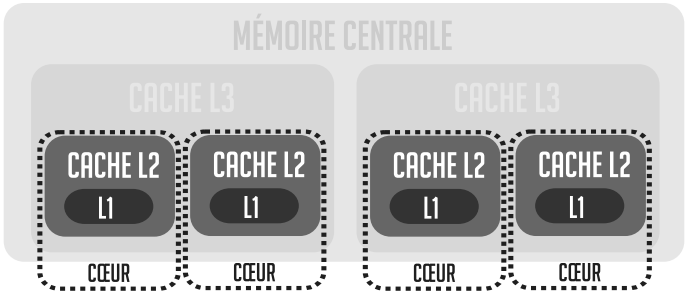


FIGURE 2.4 – Imbrication des mémoires caches dédiées à un cœur de calcul.

Morcellement du calcul pour une meilleure localité

Selon la quantité de données et les schémas d'accès, des lignes de caches peuvent être évacuées avant d'être réutilisées. Cela se traduit par des aller-retours avec la mémoire centrale, ralentissant le calcul. Pour limiter ce phénomène, une technique de morcellement des calculs[45] est fréquemment employée [65, 66]. Elle consiste à décomposer et regrouper les opérations effectuées sur des données consécutives, afin de mieux exploiter les mémoires caches. Elle complexifie considérablement le code. Le morcellement peut s'opérer à tous les niveaux des mémoires caches et peut dépendre de l'architecture matérielle lorsque les dimensions des blocs morcelés sont calculées à partir de la capacité des mémoires. Dans ce cas, des techniques de modélisation peuvent être employées afin de prédire le comportement des mémoires caches et de sélectionner de meilleurs paramètres de morcellement pour une application donnée [67, 68]. Les améliorations dépendent alors des caractéristiques matérielles de la machine ciblée. D'autres algorithmes sont

[45] *Blocking* en anglais.

au contraire élaborés pour fonctionner sans aucune connaissance a priori des caractéristiques des mémoires locales [69]. Ils sont conçus pour atteindre de bonnes performances de manière générale et ne nécessitent pas d'ajuster des paramètres de morcellement en fonction du matériel.

Des bibliothèques optimisées emploient largement ces techniques selon le type de traitement à accomplir. Elles permettent d'abstraire la complexité aux concepteurs d'applications, tout en assurant une certaine probabilité. C'est par exemple le cas de la bibliothèque mathématiques d'*Intel* (MKL[46] [70]), laquelle fournit des routines de calcul optimisées telles que des opérations d'algèbre linéaire (multiplication de matrices, résolution de systèmes, etc.) et de traitement du signal (transformées de Fourier rapides, etc.). La bibliothèque mathématiques d'AMD (ACML[47] [71]) optimisée pour les processeurs de la gamme *Opteron* offre des fonctionnalités similaires. La **figure 2.5** démontre l'intérêt de faire reposer les programmes de calcul sur ces bibliothèques optimisées. Elle présente la différence des performances séquentielles de diverses multiplications de matrices carrées. Un code naïf est comparé à une implémentation complexifiée pour effectuer des traitements par bloc, ainsi qu'à la fonction *dgemm* de la bibliothèque *Intel MKL*. Cette dernière fait appel à de nombreuses autres optimisations.

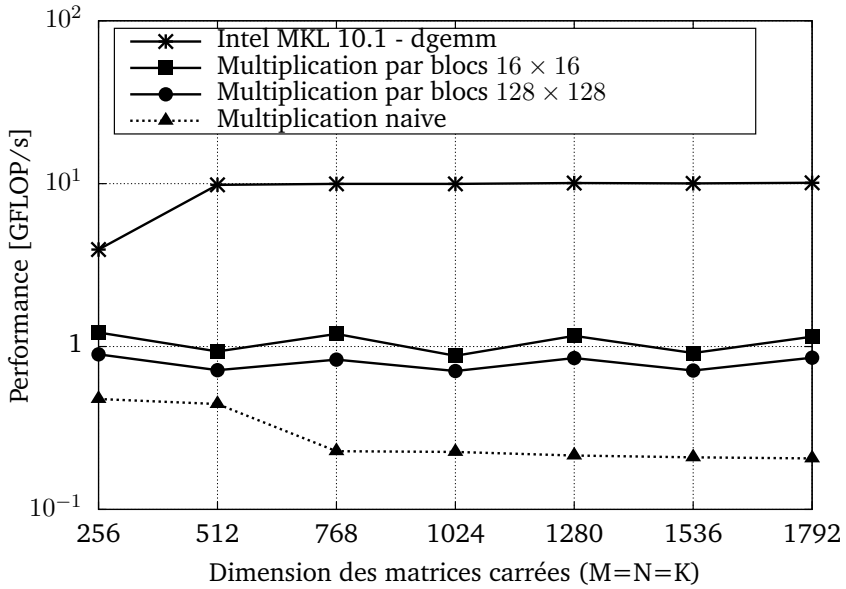


FIGURE 2.5 – Multiplications de matrices carrées en double précision. Comparaison d'un code séquentiel naïf d'une multiplication morcelée par bloc et d'une bibliothèque optimisée sur un cœur de calcul de processeur *Intel Xeon Nehalem E5640*.

2.2.2 La cohérence des données dans les mémoires caches

Les architectures SMP, associées à l'exploitation de plusieurs mémoires caches, impliquent d'assurer la cohérence des données lorsqu'elles sont dupliquées. La mémoire de chaque nœud de calcul est visible et rendue cohérente pour tous les cœurs. Par conséquent, lorsque plusieurs cœurs effectuent des opérations sur les mêmes données, toutes les autres unités de calcul nécessitent un accès à la dernière version à jour. Afin de réduire la quantité de données véhiculées, la cohérence des mémoires caches est typiquement maintenue par un protocole de cohérence par *invalidations*. Au lieu de propager directement une ligne de cache dans toutes les mémoires qui la contiennent, elle est tout d'abord invalidée là où elle est périmée. La cohérence s'opère de manière tardive

[46] **MKL :** *Math Kernel Library*.

[47] **ACML :** *AMD Core Math Library*.

lorsque la ressource de calcul a besoin à nouveau d'une donnée contenue dans la ligne de cache en question.

Le plus simple des protocoles par invalidation est nommé *MSI*[48]. Il attribue un des trois états *modifié* (M), *partagé* (S), *invalidé* (I) à chaque ligne de cache contenue dans chacune des mémoires locales. L'état *partagé* est attribué aux lignes qui sont cohérentes dans plusieurs mémoires caches à la fois. L'état *modifié* correspond à une ligne de cache qui a été écrite localement. Les copies dans les autres caches sont alors périmées (état *invalidé*). Ainsi, lorsqu'une ligne est invalide, elle doit être réactualisée avant que le cœur de calcul puisse en exploiter les données. Cette ligne est passée à l'état *partagé* tant qu'elle n'est pas modifiée. Ainsi, c'est seulement lorsqu'un cœur cherche à lire une donnée invalidée qu'une mise à jour est déclenchée. La figure 2.6 illustre ce mécanisme du point de vue d'une mémoire cache associée à un cœur de calcul.

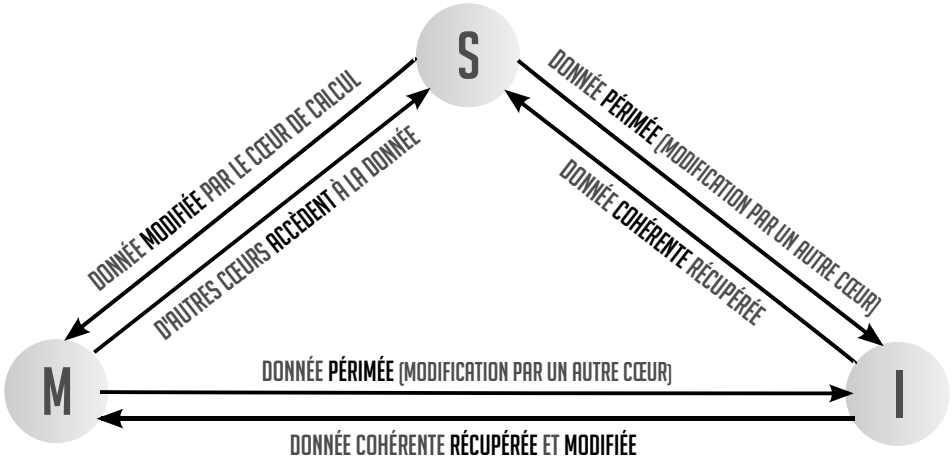


FIGURE 2.6 – Protocole *MSI* pour la cohérence des mémoires caches.

Les effets du faux partage

La propagation des modifications des données est complètement prise en charge par le matériel. Dans certains cas de figure, trop de pression sur les mécanismes de cohérence peut entraîner une perte importante des performances de calcul. Lorsque plusieurs cœurs de calcul modifient la même donnée, la ligne de cache associée est invalidée et rechargée tour à tour dans leurs mémoires caches. Pire, il peut arriver que des données différentes, placées sur la même ligne de cache, soient modifiées simultanément par plusieurs cœurs de calcul. D'un point de vue algorithmique, chaque donnée n'a besoin d'être modifiée que par un seul cœur de calcul à la fois. Pourtant, du fait de la granularité du système de cohérence, les données se trouvent partagées, puisque physiquement situées sur la même ligne de cache. Ce phénomène est appelé *faux partage*[49].

Le *faux partage* peut être évité en éloignant les données sur différentes lignes de caches. Des données non exploitées par le programme peuvent être insérées pour permettre cette séparation[50]. Il est également possible de contrôler l'affectation des calculs en éloignant dans le temps des opérations sur la même ligne de cache accédée par plusieurs cœurs de calcul. Des résultats de *faux partages* sont présentés dans la figure 2.7. Des compteurs de quatre octets sont incrémentés pour chaque cœur, entraînant des dégradations de performances lorsqu'ils ne sont pas éloignés spatialement.

[48] **MSI :** ***M****odified,* ***S****hared,* ***I****nvalid*.
[49] *False sharing* en anglais.
[50] *Padding* en anglais.

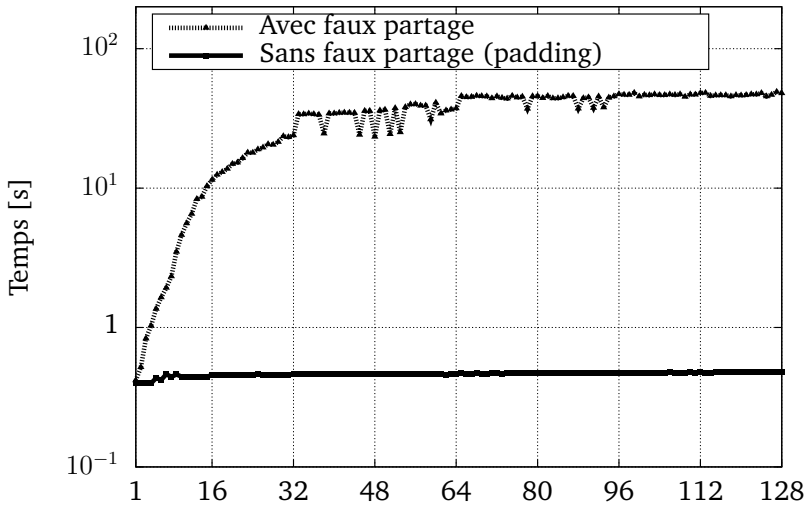


FIGURE 2.7 – Compteurs de 4 octets incrémentés localement par chaque cœur de calcul. Comparaison du temps d'exécution avec *faux partage* et sans *faux partage* en y introduisant du *padding* sur un nœud de calcul *Curie large* (cf. **annexe A.4**).

2.2.3 Mémoires caches partagées entre plusieurs cœurs de calcul

Certains processeurs possèdent des mémoires locales partagées entre plusieurs cœurs de calcul. La **figure 2.8** illustre l'emploi de deux mémoires caches de niveau 3, chacune associée à deux cœurs. Il s'agit généralement de la mémoire cache de dernier niveau, dernière mémoire locale avant les accès à la mémoire centrale. La conséquence de ce partage est potentiellement la réduction de la partie exploitable de ce cache pour chaque ressource de calcul. Une exploitation simultanée de tous les cœurs partageant le cache est donc pénalisée en comparaison d'une utilisation purement séquentielle.

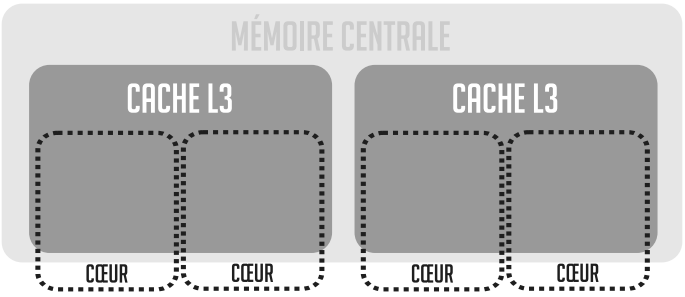


FIGURE 2.8 – Caches partagés entre plusieurs cœurs de calcul.

Exploitation collaborative d'un cache partagé

En répartissant les calculs qui ont recours aux mêmes données sur des cœurs partageant une même mémoire cache, il est possible de réduire le nombre d'accès à la mémoire centrale. Cela passe par une décomposition adéquate des calculs et à un ordonnancement tenant compte du placement des données[51] [72].

Ainsi, en se basant sur les principes de localité, les mémoires locales permettent de réduire les accès à la mémoire centrale. Les premiers niveaux sont employés pour fournir rapidement des données aux unités de calcul. En mettant en place des techniques pour favoriser l'exploitation

[51] *Data aware* en anglais.

de ces mémoires caches, et en évitant les effets de faux partage, de meilleures performances de calcul peuvent être obtenues. La tendance va à l'accroissement du nombre de cœurs dans les nœuds de calcul, augmentant par la même occasion le nombre de mémoires locales. Pour être en mesure de suivre la demande en quantité de données transférées vers les mémoires locales, les performances de la mémoire centrale doivent également être améliorées. Pour répondre à ces besoins, les architectures des nœuds de calcul ont subi des modifications. De nouveaux problèmes ont été introduits et sont abordés dans la section suivante.

2.3 Les accès à la mémoire centrale par les processeurs

La mémoire centrale contient (idéalement) toutes les données d'un programme informatique. Les architectures *SMP* permettent à tous les processeurs de disposer de cette mémoire. Des résultats intermédiaires peuvent, par exemple, être directement accessibles par n'importe quel cœur du nœud de calcul. L'enjeu est de conserver cette caractéristique, tout en minimisant les points de contention sur la mémoire. En effet, cette dernière doit répondre à un besoin croissant en bande passante pour alimenter les mémoires locales qu'embarquent les cœurs de calculs de plus en plus nombreux.

Les premières architectures *à mémoire partagée* agrégeaient des processeurs autour d'un système de communication ou bus (FSB[52]) afin d'accéder à la mémoire centrale. Tous les processeurs partageaient ce lien et ainsi la même pénalité d'accès à la mémoire pouvait être observée. Il s'agissait de machines dites *à accès uniformes à la mémoire* (*UMA*[53]) puisque chaque accès impliquait en moyenne la même latence, indépendamment du processeur qui en émettait la requête. La **figure 2.9** schématise ce genre d'architecture en regroupant quatre processeurs autour d'une mémoire. Sur le bus, transitaient à la fois des informations permettant la cohérence des mémoires locales et des échanges de données avec la mémoire centrale. Les requêtes à destination de cette dernière étaient interceptées par un contrôleur qui se chargeait de centraliser les opérations d'écriture et de lecture. A titre d'exemple, les processeurs de chaque nœud de calcul de la machine *ASCI Red* (cf. **chapitre 1.2.1**, page 20) étaient connectés par un bus à accès uniformes à la mémoire. Les performances croissantes des microprocesseurs multi-cœurs ont conduit à un besoin en mouvement de données de plus en plus conséquent. Les bus et le contrôleur mémoire, tous deux centralisés, sont devenus des goulots d'étranglement importants. Dans les années 90, W. Stallings évoquait déjà une limite située entre 16 et 64 processeurs pour ce modèle à accès uniformes et centralisés [73].

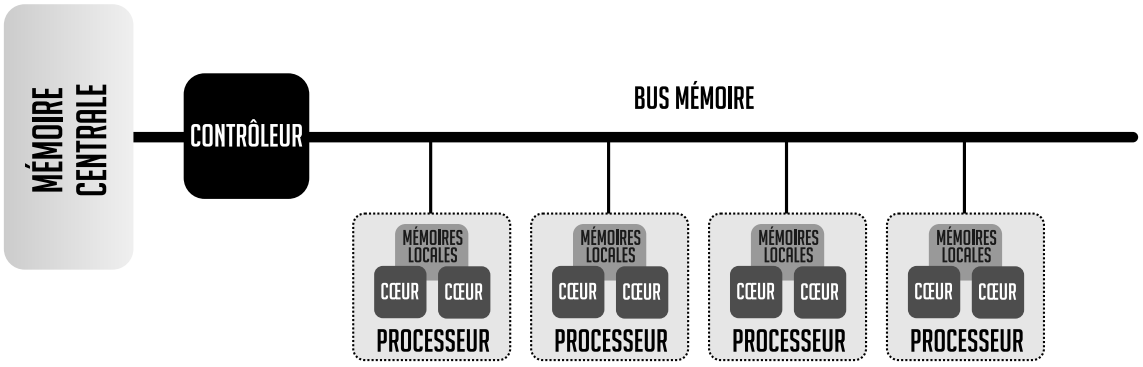


FIGURE 2.9 – Architecture à accès uniformes à la mémoire centrale (*UMA*).

[52] **FSB :** *Front Side Bus*.

[53] **UMA :** *Uniform Memory Access*.

Cette section présente des technologies adoptées par les fondeurs et constructeurs de machines pour pallier les pénalités engendrées par des accès multiples à la mémoire. Les architectures qui en découlent introduisent de nouvelles contraintes de programmation dont les effets sont traités ci-après.

2.3.1 Des accès non uniformes à la mémoire centrale

L'architecture *SMP* a subi d'importantes modifications, notamment pour contourner le point de contention qu'imposait le bus de la mémoire centrale. Une réponse à cette limitation a consisté en un retour partiel vers un type d'accès particulier employé dans certains supercalculateurs des années 80 et 90 (*BBN Butterfly*, *Stanford DASH* [74], *Cray T3D* [75], *SGI Origin*[76]) avant le succès des grappes de calcul. Cette solution combine les avantages des architectures *SMP* et *AMP*, permettant un meilleur passage à l'échelle (*SSMP*[54] [77]) tout en offrant l'illusion d'une mémoire centralisée. En réalité, elle est composée de plusieurs mémoires distribuées et partagées (*DSM*[55]) dans le nœud de calcul.

Une des conséquences de cette architecture est l'introduction de variations des temps d'accès en fonction de la zone mémoire sollicitée. Ces machines possèdent donc une architecture à *accès non uniformes à la mémoire* (*NUMA*[56]). La **figure 2.10** schématise ce type d'architecture en représentant quatre processeurs octo-cœurs, chacun étant directement connecté à une mémoire. C'est ce type d'assemblage qui a été adopté pour les nœuds de calcul standards des supercalculateurs *Tera-100* et *Curie* (cf. **chapitre 1.3**, page 27). Bien qu'un nœud de calcul combine maintenant plusieurs dépôts distribués, le terme de mémoire centrale est conservé dans la suite de ce manuscrit. Ce terme désigne alors l'union des emplacements distribués puisque cet ensemble peut toujours être considéré comme centralisé par des programmeurs non avertis.

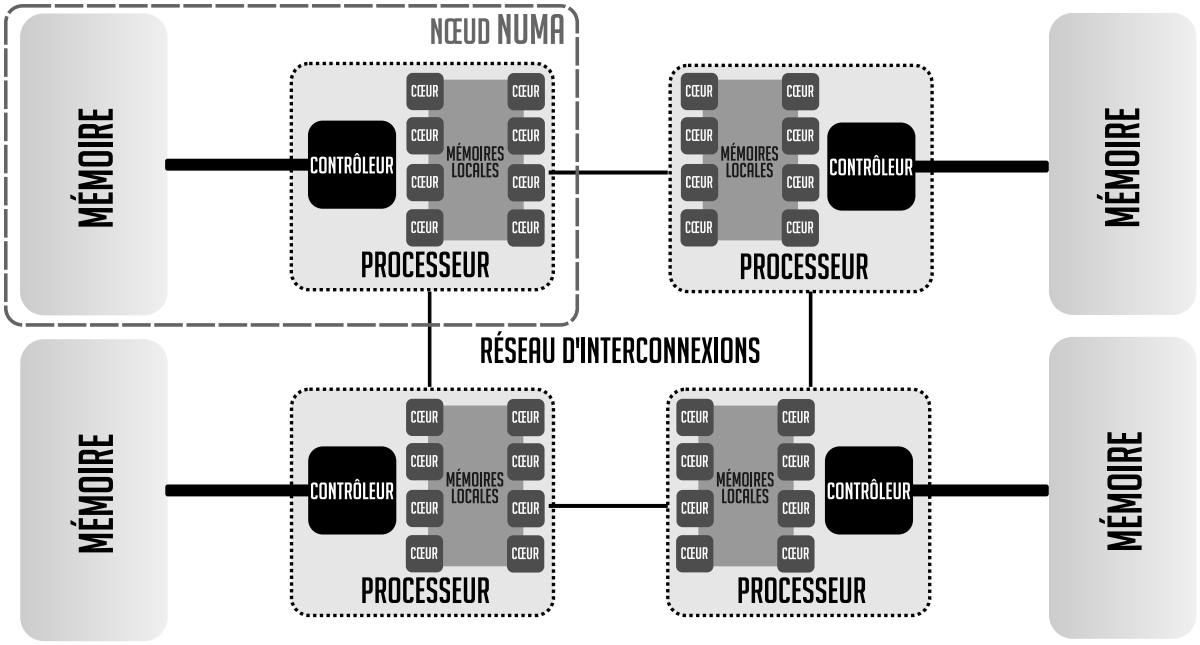


FIGURE 2.10 – Architecture à accès non uniformes à la mémoire centrale (*NUMA*).

[54] **SSMP :** *Scalable Shared-memory MultiProcessors*.
[55] **DSM :** *Distributed Shared Memory*.
[56] **NUMA :** *Non-Uniform Memory Access*.

Plusieurs contrôleurs sont alors employés pour piloter la mémoire. La plupart des processeurs embarquent désormais un de ces contrôleurs (à partir de 2003 pour *AMD*, 2008 pour *Intel*), permettant d'y associer directement une mémoire, tout en améliorant les temps d'accès à celle-ci. De surcroît, les processeurs ne sont plus connectés à un unique bus, mais reliés via un réseau d'interconnexion point à point. À titre d'exemples, les processeurs *X86* d'*AMD* adoptent le format d'interconnexion *HyperTransport* [78, 79], tandis que ceux d'*Intel* emploient le réseau *QPI*[57] [80].

Cette nouvelle architecture permet de se débarrasser de la contention d'un unique bus mémoire. En contrepartie, des variations des temps d'accès aux mémoires apparaissent. Un processeur accède plus rapidement à la mémoire directement connectée au contrôleur embarqué [81]. Pour tenir compte de cette affinité, le terme de *nœud NUMA* est employé pour désigner l'association d'un contrôleur, d'une mémoire et d'un lot de cœurs de calcul possédant un accès rapide à cette mémoire. La *distance*, ou le *facteur NUMA*, est un paramètre qui peut servir à anticiper les pénalités d'accès distants, c'est-à-dire des accès effectués par un cœur de calcul à de la mémoire située dans un autre *nœud NUMA*. Ce facteur est défini par le rapport entre le temps d'accès à la mémoire distante sur le temps d'accès à la mémoire locale. Plus le facteur est élevé, plus les accès distants sont pénalisants lors des calculs [82, 83].

La cohérence des mémoires locales est toujours maintenue (*ccNUMA*[58]) à l'échelle de ligne de caches. À la place d'une cohérence maintenue par une écoute sur un unique bus [84], elle est gérée de manière répartie. Des circuits spéciaux appelés *répertoires* [85] sont distribués dans chaque nœud NUMA et servent à conserver, localement, la liste des cœurs de calculs qui accèdent à des données situées dans la mémoire associée. Lorsque les données sont modifiées, uniquement les cœurs qui détiennent une copie en mémoire cache reçoivent une requête d'invalidation. Seule une partie du réseau d'interconnexion est occupée, minimisant ainsi le trafic induit par le maintien de la cohérence des données.

À l'image des mémoires caches, le placement des données dans les mémoires des nœuds NUMA est géré par des blocs de mémoire physiquement contiguës. Ces blocs sont appelés pages mémoires et contiennent au minimum l'équivalent de plusieurs dizaines de lignes de cache (4 096 octets en général - des grosses pages de l'ordre de plusieurs Mo peuvent plus rarement être employées). Les données d'une page mémoire ne peuvent donc se situer que dans la mémoire d'un nœud NUMA à la fois (hors réplication explicite). L'évolution des nœuds de calcul permet un passage à l'échelle, mais introduit des effets *NUMA*. Ces effets peuvent être constatés dans différents cas de figure, lesquels sont décrits dans les sous-sections suivantes.

Les effets NUMA au sein d'un processeur

Des processeurs sont conçus en combinant l'équivalent de plusieurs processeurs distincts sur la même puce (*Intel Core 2 Quad*, *AMD Opteron [86] Magny-Cours* [87]). Ces processeurs peuvent également contenir plusieurs contrôleurs mémoires, impliquant des effets NUMA. Dans ce cas précis, il est fréquent qu'un seul contrôleur soit connecté à la mémoire. Les groupes des cœurs de calcul ne possédant pas de mémoire directement rattachée n'accèdent alors qu'à de la mémoire distante tel que l'illustre le schéma d'interconnexion (cf. **figure 2.11a**) de la mémoire et d'un processeur *AMD Opteron 6164 HE* embarqués dans le nœud de calcul *ENS-Fermi* (cf. **annexe A.6**). La **figure 2.11b** présente l'impact de ces accès non uniformes sur la bande passante d'une copie de 32 Mo à partir de ce processeur embarquant un total de douze cœurs de calcul. Les cœurs numérotés de 7 à 12 ne possèdent pas de mémoire directement rattachée à leur contrôleur. Ainsi,

[57] **QPI :** *QuickPath Interconnect*.

[58] **ccNUMA :** *Cache Coherent NUMA*.

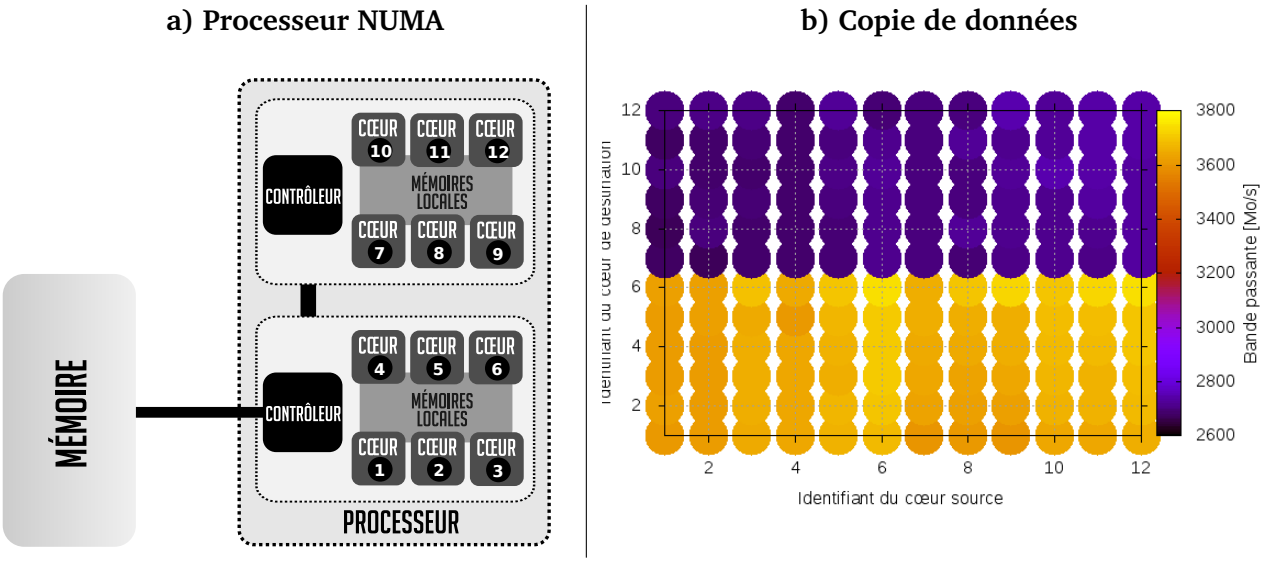


FIGURE 2.11 – Effets NUMA d'un processeur 12 cœurs *AMD Opteron 6164 HE* employé dans le nœud de calcul *ENS-Fermi* (cf. annexe A.6). **a)** Architecture *NUMA* du processeur. **b)** Efficacité d'une copie de données (32 Mo contigus) en Mo/s, en fonction de l'affinité mémoire de chaque cœur de calcul.

lorsque des données sont allouées par ceux-ci, elles résident dans la mémoire connectée aux cœurs 1 à 6. La bande passante est systématiquement plus faible lorsque les cœurs 7 à 12 accèdent aux données (cœurs de destination).

Les effets NUMA hiérarchiques entre processeurs

L'interconnexion de plusieurs processeurs sur un même support ou *carte mère* mène à des effets NUMA. Il existe typiquement des cartes mères pouvant accueillir deux ou quatre processeurs. Certains constructeurs de machines, désireux d'accroître sensiblement la quantité de la mémoire et du nombre de cœurs dans un nœud de calcul, ont eu l'idée d'étendre l'interconnexion des processeurs à plusieurs cartes mères (*Bull Coherence System*, *SGI NUMAflex*). Ainsi, plusieurs nœuds de calcul peuvent être agrégés en un seul. Il en résulte une hiérarchie d'effets NUMA avec différents types d'accès distants, à savoir les accès NUMA liés à une même carte mère et ceux induits par des cartes mères distinctes. La figure 2.12 illustre ces connexions étendues sur une machine *Bull Novascale Bullion* (*Curie large*, cf. annexe A.4) composée de seize nœuds NUMA, lesquels sont organisés en deux niveaux. Il s'agit schématiquement d'un assemblage de quatre cartes mères, chacune embarquant quatre processeurs *Intel Xeon Nehalem* [88, 89] de huit cœurs. La figure 2.13 révèle les conséquences de ces accès non uniformes hiérarchiques sur ce nœud de calcul en relevant la bande passante d'une copie de 32 Mo. Les copies opérées à partir d'un cœur depuis la même mémoire permettent d'atteindre environ 2 800 Mo/s. Les bandes passantes entre mémoires issues d'une même carte mère s'élèvent quant à elles à environ 2 400 Mo/s, alors que celles entre assemblages distants ne parviennent pas à la moitié (1 100 Mo/s).

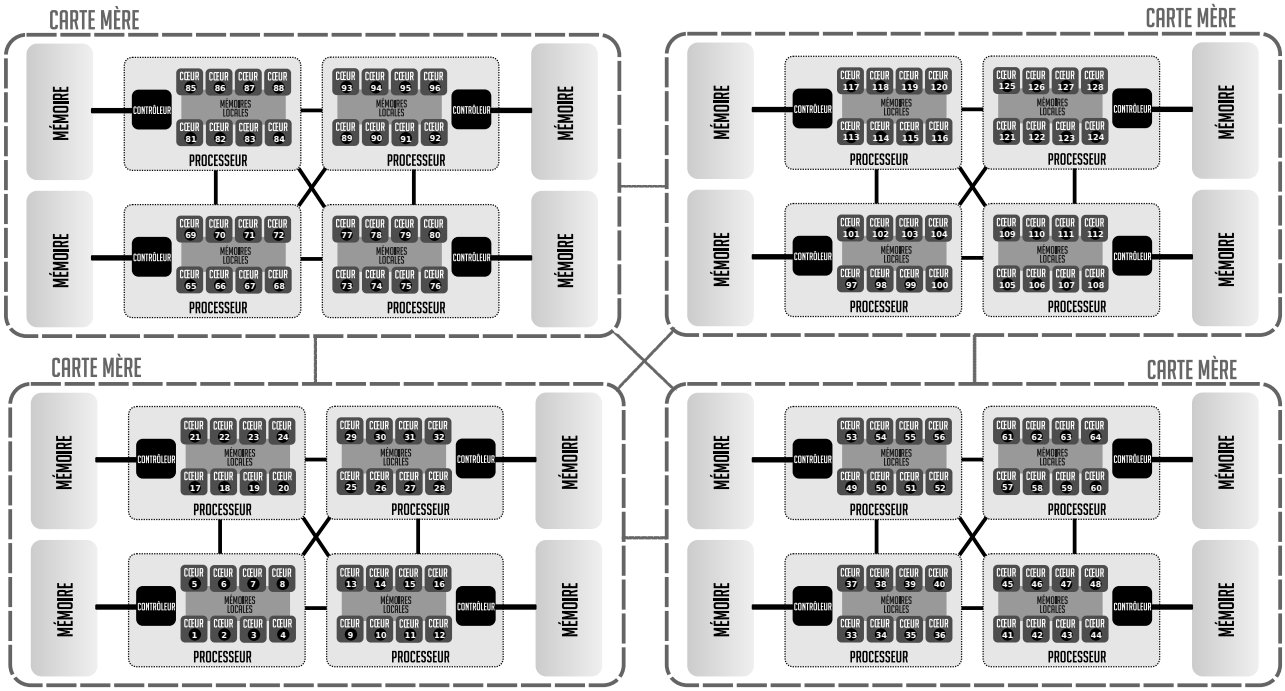


FIGURE 2.12 – Interconnexions entre processeurs et mémoires sur un assemblage *Bull Novascale Bullion*, totalisant 128 cœurs de calcul. Architecture à 16 nœuds NUMA hiérarchiques (*Curie large*, cf. annexe A.4).

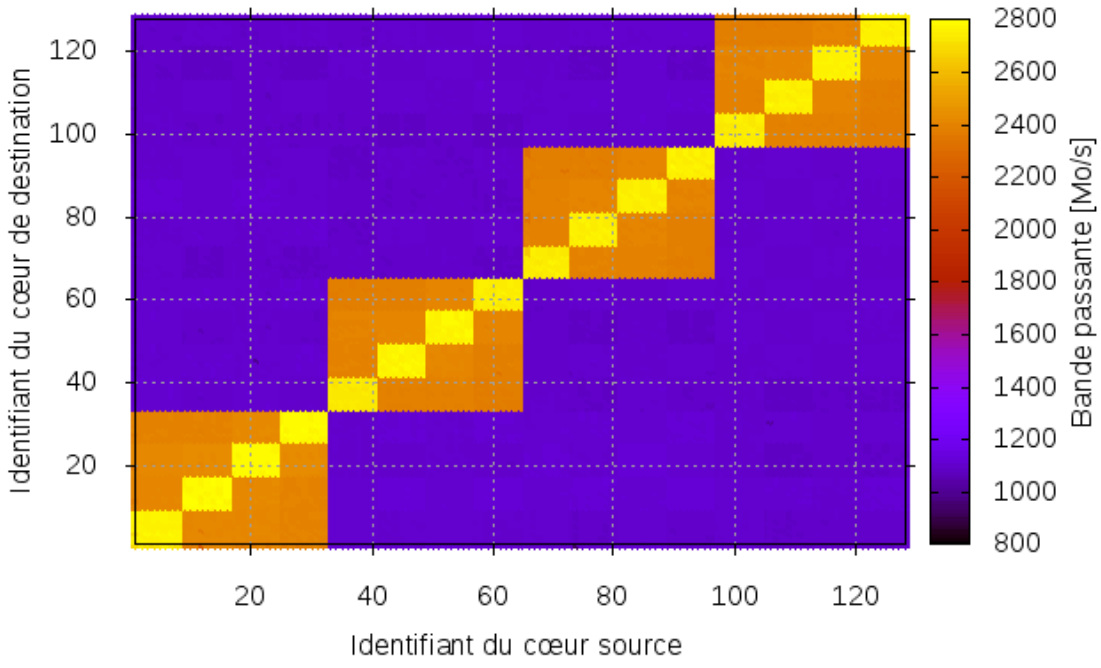


FIGURE 2.13 – Effets NUMA entre processeurs sur un assemblage *Bull Novascale Bullion*. Efficacité d'une copie de données (32 Mo contigus) en Mo/s, en fonction de l'affinité mémoire de chaque cœur de calcul (*Curie large*, cf. annexe A.4).

Les effets NUMA dus à l'absence de certaines interconnexions

Des interconnexions peuvent être manquantes, soit à cause d'une version bridée du processeur, soit du fait d'une connexion omise au sein ou entre les cartes mères pour des raisons de coûts de fabrication. Les processeurs ne sont alors pas tous directement interconnectés. Des *sauts* via d'autres processeurs peuvent être nécessaires avant d'atteindre un nœud NUMA distant, ce qui se traduit par des temps d'accès rallongés.

Les effets NUMA peuvent être contenus. Les accès distants doivent être minimisés en s'appuyant sur la connaissance de la topologie de la machine, notamment l'affinité mémoire des processeurs, ainsi que les facteurs NUMA. Cela passe par un placement initial des pages dans la mémoire des nœuds NUMA adéquats, combiné à la prise en compte de ces placements lors de l'exécution du programme.

2.3.2 Bande passante limitée par cœur de calcul

La bande passante des cœurs de calcul d'un nœud NUMA est généralement restreinte. Chaque cœur de calcul n'a accès qu'à une fraction de cette bande passante. Afin d'atteindre la meilleure bande passante possible, tous les cœurs doivent travailler de concert.

En résumé, les architectures des machines actuelles requièrent une utilisation conjointe des cœurs de calcul pour atteindre la bande passante maximale. Elles induisent également des accès non uniformes à la mémoire. Une connaissance de l'interconnexion des processeurs, ainsi qu'un placement stratégique des pages mémoires permettent de limiter les pénalités engendrées par les accès distants. La prochaine section met en exergue les effets de ces phénomènes sur les performances des périphériques tels que les accélérateurs, ainsi que les limitations associées à la gestion de la mémoire de ces matériels spécialisés.

2.4 Les accès à la mémoire centrale par les accélérateurs matériels

Les accélérateurs matériels se présentent majoritairement sous la forme de cartes d'extension (cf. **chapitre 1.2.3**, page 23) et embarquent dans cas une mémoire dédiée. Ces cartes spécialisées sont rajoutées dans un nœud de calcul qui joue le rôle d'*hôte* comme le schématise la **figure 2.14** qui correspond à l'architecture des nœuds de type *Curie hétérogène* (cf. **annexe A.3**) ou *Inti hétérogène* (cf. **annexe A.1**). La technologie des mémoires de ces nouvelles ressources de calcul déportées est légèrement différente de celle employée pour la mémoire centrale. Elle permet une utilisation plus adaptée au besoin des matériels spécialisés, en fournissant une bande passante plus conséquente, imposée par une puissance de calcul plus élevée que celle délivrée par les processeurs généralistes. Par exemple, les cartes accélératrices *Nvidia* de génération *Kepler 2* ont recours à une mémoire graphique capable d'atteindre théoriquement 280 Go/s contre environ 50 Go/s pour les processeurs contemporains *Intel* de microarchitecture *Ivy Bridge*. La **figure 2.15** met en lumière ces disparités en bandes passantes.

Certains processeurs généralistes intègrent des processeurs graphiques sur la même puce (*AMD APU* [90], *Intel Sandy Bridge*, *Intel Ivy Bridge* etc.). Ce choix est principalement opéré par souci de consommation électrique. Ces processeurs graphiques intégrés partagent directement la mémoire centrale avec les processeurs généralistes. Ils ne permettent pas encore d'apporter des performances aussi élevées que les versions reliées à une mémoire dédiée.

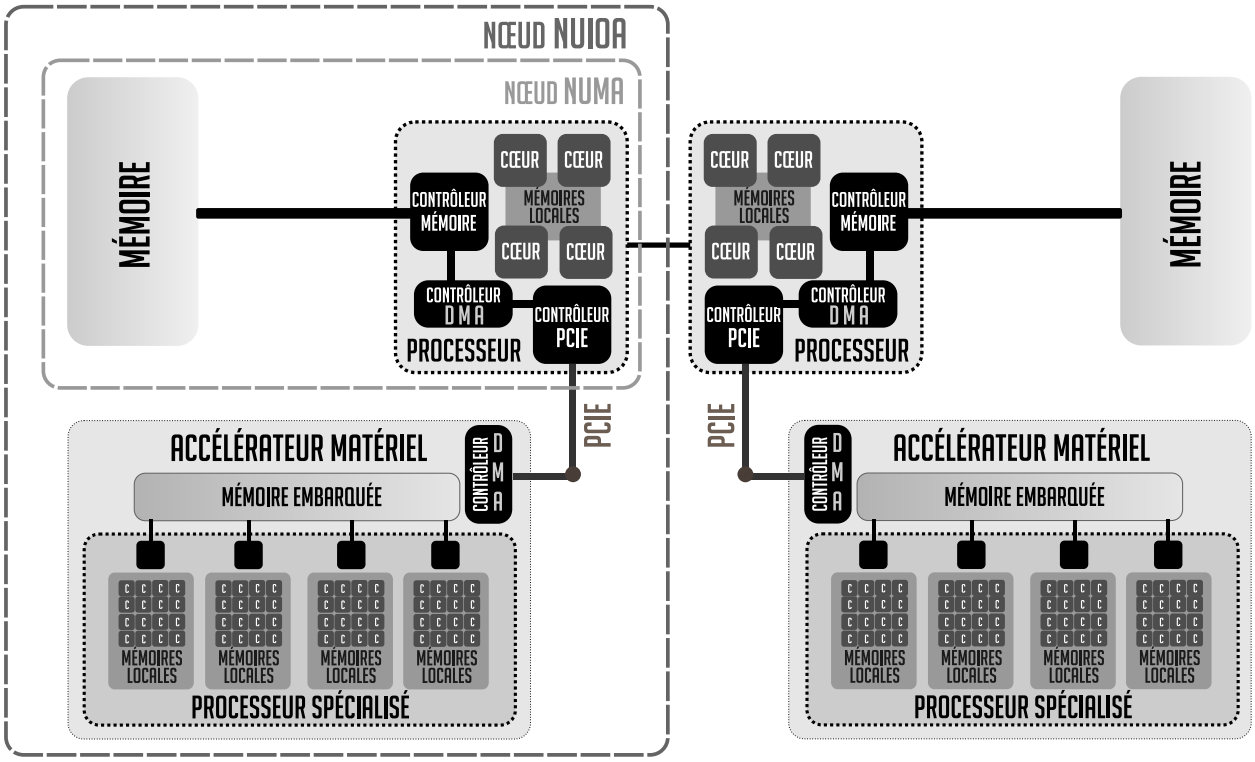


FIGURE 2.14 – Architecture comportant deux nœuds *NUMA*, chacun relié via une communication *PCIe* à un accélérateur matériel déporté constituant ainsi des nœuds internes à accès non-uniformes à la mémoire (*NUMA*) et aux entrées/sorties (*NUIOA*). Correspond à l'architecture des nœuds de type *Curie hétérogène* (cf. **annexe A.3**) ou *Inti hétérogène* (cf. **annexe A.1**).

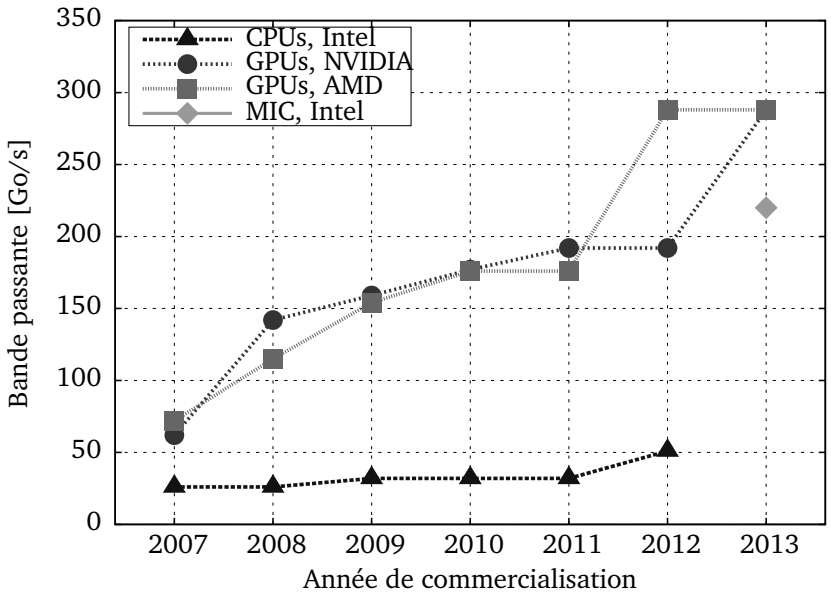


FIGURE 2.15 – Bandes passantes maximales (théoriques) de CPUs, GPUs et MIC *(source [34])*.

Les matériels déportés s'articulent autour d'une hiérarchie mémoire comparable à celle rencontrée dans un nœud de calcul. Ils communiquent ensuite avec l'hôte via un bus d'entrées/sorties baptisé *PCI Express* (*PCIe*), lien qui peut également être employé pour communiquer avec d'autres nœuds de calcul. Dans le cas des accélérateurs matériels, ce bus permet les échanges de données entre la mémoire centrale et la mémoire embarquée. Cependant, la bande passante est relativement restreinte comparée à celle que peut soutenir l'ensemble de la mémoire embarquée. Par exemple, la dernière norme *PCIe*, seulement exploitée par les accélérateurs les plus récents (PCIe 3.0 16X), fournit une bande passante théorique d'environ 16 Go/s par direction contre 280 Go/s pour la mémoire embarquée dans les derniers accélérateurs. Le chargement des données est ainsi fortement limité par le bus PCIe[59]. Le rapport de la quantité de calculs sur la quantité de données à transférer est important à étudier pour déterminer si le déport d'un traitement sur un accélérateur en vaut la peine.

Les échanges de données sont réalisés par accès directs à la mémoire centrale (*DMA*[60]). Cela permet d'effectuer des opérations de transfert, sans monopoliser un cœur de calcul, en faisant appel à une puce spécialisée appelée *contrôleur DMA*. Ainsi, le contrôleur DMA permet des copies asynchrones, c'est-à-dire qu'il peut se charger de transférer des informations pendant que les processeurs sont libres d'effectuer des calculs sur d'autres données. Si les calculs sont majoritaires, les temps de transferts peuvent être théoriquement "masqués", puisqu'ils sont effectués en parallèle. Néanmoins, les communications ne sont pas automatiquement déclenchées par le matériel. La cohérence des données, entre la mémoire centrale et les mémoires embarquées, doit être maintenue via des copies explicites par le logiciel. Ces copies sont souvent renseignées par les programmeurs de code de simulation. Tout comme le contrôleur mémoire, les contrôleurs *PCIe* sont désormais intégrés aux processeurs. Chaque accélérateur est directement connecté à un processeur central. Cette architecture étendue induit des accès non uniformes d'entrées/sorties (*NUIOA*[61] [91]), dont les effets seront introduits dans les prochaines sections.

2.4.1 Les contraintes d'accès à la mémoire centrale en mode DMA

Afin d'exploiter un accélérateur déporté, les données nécessaires aux calculs doivent être préalablement transférées dans la mémoire embarquée, laquelle pourra fournir une bande passante plus adéquate aux cœurs de calculs spécialisés. La copie par mode DMA permet d'effectuer des transferts asynchrones sans intervention d'un cœur de calcul. Les données transférées par le contrôleur DMA doivent être gérées de manière particulière par le système d'exploitation. Ce dernier peut migrer certaines pages mémoires pour les déplacer physiquement sur un autre support de stockage. Les pages mémoire doivent être marquées comme "*verrouillées*" (ou *punaisées*), ce qui permet d'assurer au contrôleur DMA que les données ne seront pas déplacées. Or, le programmeur peut employer de la mémoire non verrouillée dans son programme. Dans ce cas, les pilotes des matériels spécialisés ont recours à des espaces tampons en mémoire centrale dont les pages mémoires associées sont verrouillées. Les données y sont préalablement copiées[62] pour permettre un fonctionnement en mode DMA. La **figure 2.16** montre que ces copies intermédiaires peuvent fortement impacter les bandes passantes utiles d'une copie via le *PCIe*.

[59] Les processeurs graphiques évalués dans ce manuscrit sont connectés à un bus *PCIe 2.0 16X*, pouvant théoriquement transférer des données à 8 Go/s.

[60] **DMA :** *Direct Memory Access*.

[61] **NUIOA :** *Non-Uniform Inputs/Outputs Access*.

[62] Technique du ***double buffering*** en anglais.

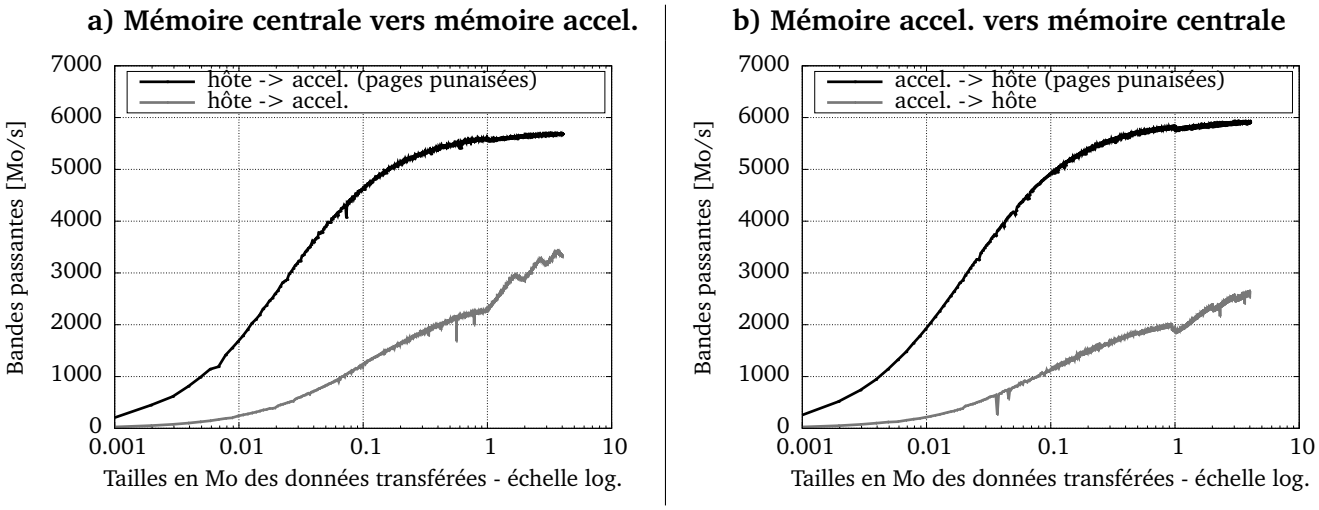


FIGURE 2.16 – Bandes passantes avec et sans verrouillage des pages associées en mémoire centrale (*Curie hétérogène*, cf. **annexe A.3**).

Chaque déclenchement de transfert apporte également un surcoût induit par son initiation. Ainsi, pour une même quantité de données transférées, plusieurs sous-transferts introduisent plus de latences qu'un unique transfert. La **figure 2.17** révèle l'intérêt de les regrouper afin d'éviter plusieurs petits transferts. Une copie d'1 Mo est décomposée en plusieurs sous-transferts. La dégradation de performance est amplifiée avec l'augmentation du nombre de transferts générés.

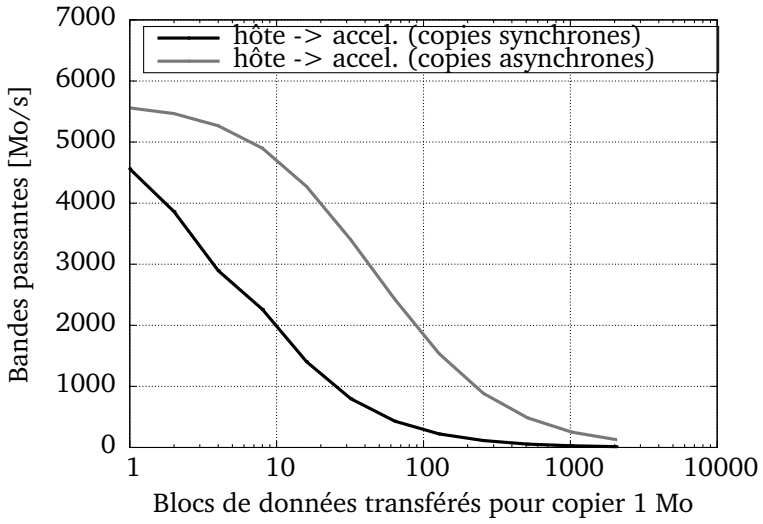


FIGURE 2.17 – Impact de la décomposition d'une copie d'1 Mo (sens mémoire centrale vers mémoire embarquée avec pages verrouillées) en plusieurs transferts sur un nœud de calcul *Inti hétérogène* (cf. **annexe A.1**).

2.4.2 Des accès asymétriques et non-uniformes à la mémoire centrale

Des bandes passantes asymétriques peuvent être constatées en fonction du sens de la copie sur le bus *PCIe*. La **figure 2.18** révèle ces phénomènes qui s'expriment de manières différentes en fonction de la machine ciblée. Peu d'actions peuvent être réalisées au niveau de l'application pour tenir compte de ces disparités. Le sens hôte vers accélérateur est sans doute le plus pénalisant dans la plupart des cas de figure, surtout lorsqu'une plus grande quantité de données en lecture seule est nécessaire pour produire un résultat qui sera rapatrié en mémoire centrale. Aucune action particulière n'a été entreprise dans cette thèse pour tenir compte de ces phénomènes

asymétriques, à part accroître la localité des données déportées afin de réduire globalement la quantité de transferts.

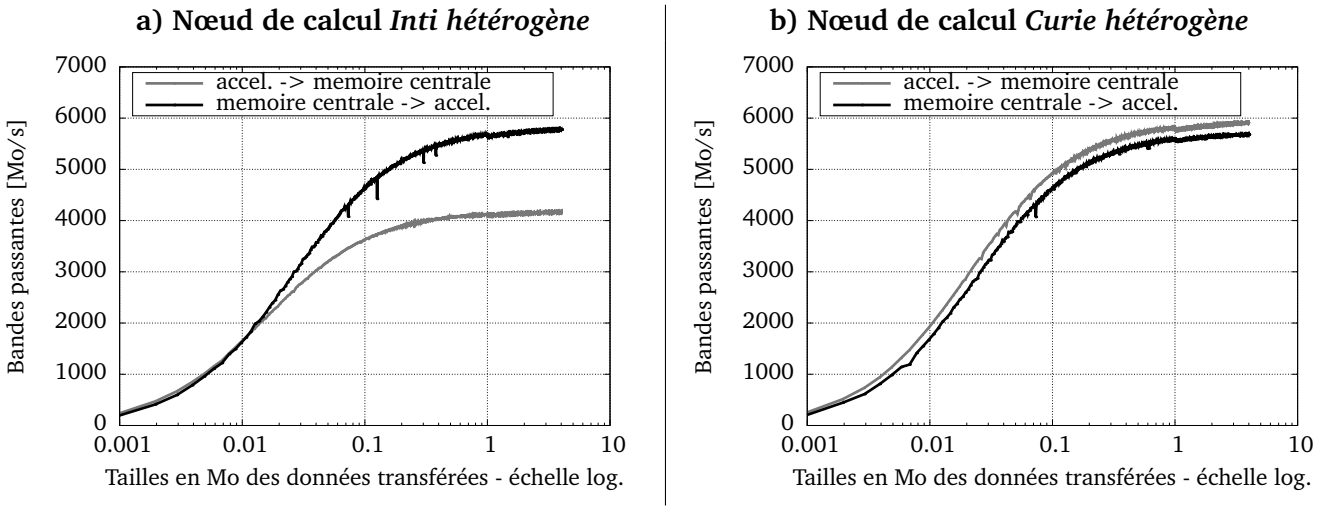


FIGURE 2.18 – Bandes passantes asymétriques à la mémoire centrale, dépendant du sens d'accès (pages verrouillées). **a)** Le sens mémoire centrale vers mémoire embarquée offre une meilleure bande passante sur le nœud de calcul *Inti hétérogène* (cf. annexe A.1). **b)** Le sens mémoire embarquée vers mémoire centrale fournit une meilleure bande passante sur le nœud de calcul *Curie hétérogène* (cf. annexe A.3).

L'interconnexion de matériels spécialisés à des nœuds de calcul NUMA introduit d'autres effets d'accès non uniformes (*NUIOA*). Un *nœud NUIOA* désigne la réunion d'un *nœud NUMA* et d'un ou plusieurs accélérateurs qui y sont directement connectés, comme l'illustre la figure 2.14. La figure 2.19 montre les effets sur les bandes passantes du *PCIe* en fonction de la mémoire du nœud NUMA accédée. Ces effets doivent être pris en compte lors de la conception d'applications en identifiant l'affinité d'un accélérateur à un *nœud NUMA*.

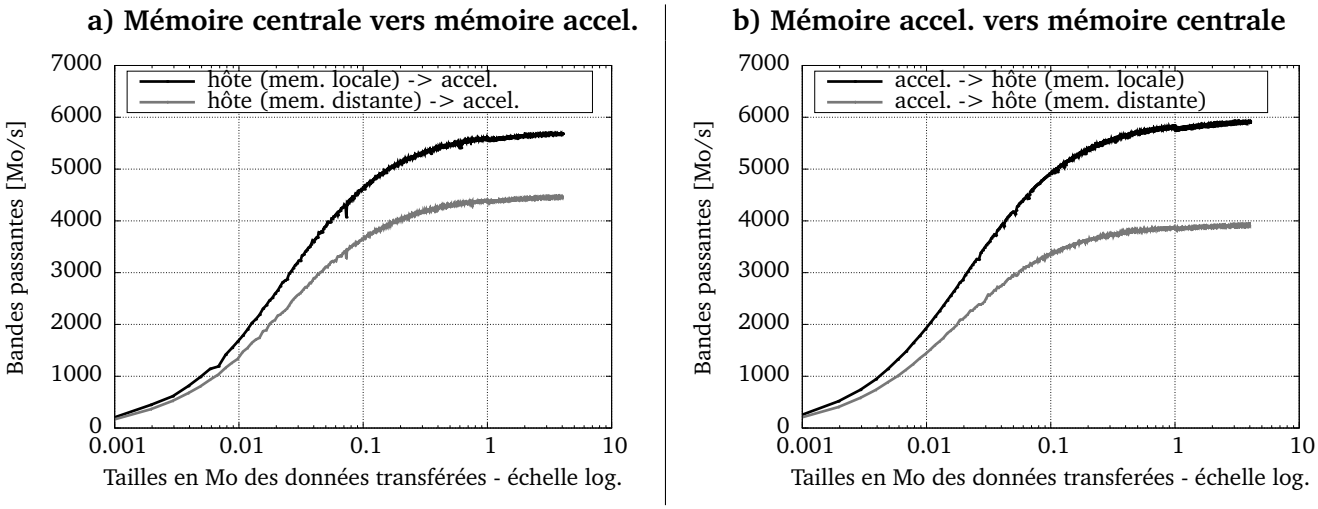


FIGURE 2.19 – Effets *NUIOA* sur un nœud de calcul *Curie hétérogène* (cf. annexe A.3). Bandes passantes des accès à la mémoire du *nœud NUIOA* contenant l'accélérateur, et des accès à de la mémoire distante. **a)** Sens mémoire centrale vers mémoire embarquée. **b)** Sens mémoire embarquée vers mémoire centrale.

2.4.3 L'exploitation des mémoires embarquées

Les sous-sections précédentes se sont focalisées sur des contraintes de transferts de données entre l'hôte et un accélérateur matériel via l'interconnexion PCIe. Lorsque ces chargements sont réalisés de manière à contenir la plupart des effets indésirables, une attention particulière peut être portée aux accès de la mémoire embarquée par les unités de calcul de l'accélérateur. Des pressions relativement importantes sont imposées à ces mémoires sollicitées par une grande quantité d'unités d'exécution. Les problèmes d'accès aux mémoires sont alors exacerbés par rapport à ceux rencontrés avec un processeur généraliste composé de quelques cœurs de calcul. Il devient alors crucial de respecter certains schémas d'accès à la mémoire afin d'atteindre des bandes passantes satisfaisantes. Des processeurs spécifiques peuvent aussi embarquer des mémoires locales spécifiques afin de répondre plus efficacement à des besoins en localité temporelle propres à chaque application. Des problématiques propres à l' exploitation de la hiérarchie mémoire de certains accélérateurs massivement parallèles sont abordées ci-après.

Une capacité mémoire limitée

Chaque accélérateur ne dispose en général que d'une quantité de mémoire bien maigre comparé à la capacité totale de la mémoire centrale. À titre d'exemple, le *GPU Nvidia Tesla K20X*, constitué de $2\,688$ processeurs de flux, ne peut compter que sur 6 Go de mémoire embarquée. En répartissant équitablement cette capacité, chaque processeur de flux ne dispose finalement que d'une mémoire très réduite d'environ $2{,}3$ Mo. Le *MIC Intel Xeon Phi 7100* quant à lui intègre 61 cœurs physiques et 16 Go, ce qui représente une quantité de 268 Mo par cœur ou 33 Mo par voie vectorielle (double précision). De plus, ces calculs ne tiennent pas compte de la nécessité de recourir à plusieurs processus par cœur (*SMT*), afin de mieux exploiter le matériel en recouvrant une partie des latences mémoires par du calcul. Habituellement, la quantité de mémoire par cœur se voit encore divisée par un facteur au moins égal à 4. Les capacités mémoire réduites se traduisent par de fortes contraintes de programmation pour les développeurs qui doivent concevoir des noyaux de calcul adaptés à ces capacités limitées. Certains jeux de données ne peuvent pas être directement contenus dans ces mémoires déportées. Généralement, des transferts sont déjà nécessaires pour échanger des données entre toutes les mémoires contenues dans plusieurs nœuds de calcul. Une quantité de mémoire réduite impose au programmeur d'initier, fur et à mesure, des opérations et des échanges de données entre les différentes mémoires, ce qui dégrade davantage les performances et alourdit potentiellement le code de calcul.

Des schémas performants d'accès aux données

Les mémoires embarquées peuvent fournir une bande passante relativement élevée. Cependant, la pression exercée par les processeurs de flux est gigantesque. Pour obtenir une bande passante capable d'alimenter en données ces centaines, voire ces milliers d'unités de calcul, la localité spatiale des accès doit être intensifiée. Il faut alors que les unités d'exécution soient coordonnées pour récupérer des données qui sont relativement groupées spatialement afin de mutualiser les accès à la mémoire et profiter de caches communs. Ces récupérations contiguës, communément appelées *accès coalescés*, doivent être explicitement établis par le programmeur en stockant à la fois les données de manière adéquate et en organisant leur récupération par les processeurs de flux.

Des mémoires locales programmables

Certains accélérateurs embarquent des mémoires locales programmables[63]. Celles-ci disposent d'une capacité aussi limitée que celles des caches et sont également employées pour réduire les sollicitations de la mémoire centrale. Elles offrent un plus grand contrôle aux développeurs d'applications mais au prix d'une complexité de programmation accrue.

[63] *Scratchpad memory* en anglais.

Discussion

En résumé, l'évolution des architectures a été accélérée par les besoins grandissants en bande passante des nœuds de calcul, afin de permettre un passage à l'échelle des performances. Les machines deviennent bel et bien de plus en plus puissantes, au prix d'une nette complexification. Celle-ci semble inévitable pour atteindre une vitesse de chargement des données suffisante vers des unités d'exécution de plus en plus nombreuses. Ainsi, des procédés visant à réduire les accès à la mémoire centrale sont largement déployés. La plupart des contraintes associées aux performances d'accès aux données sont résumées dans les prochaines sections. L'évolution de ces contraintes y est également abordée pour les prochaines générations supposées de machines. La figure 2.20 fournit une vue d'ensemble des contraintes d'accès.

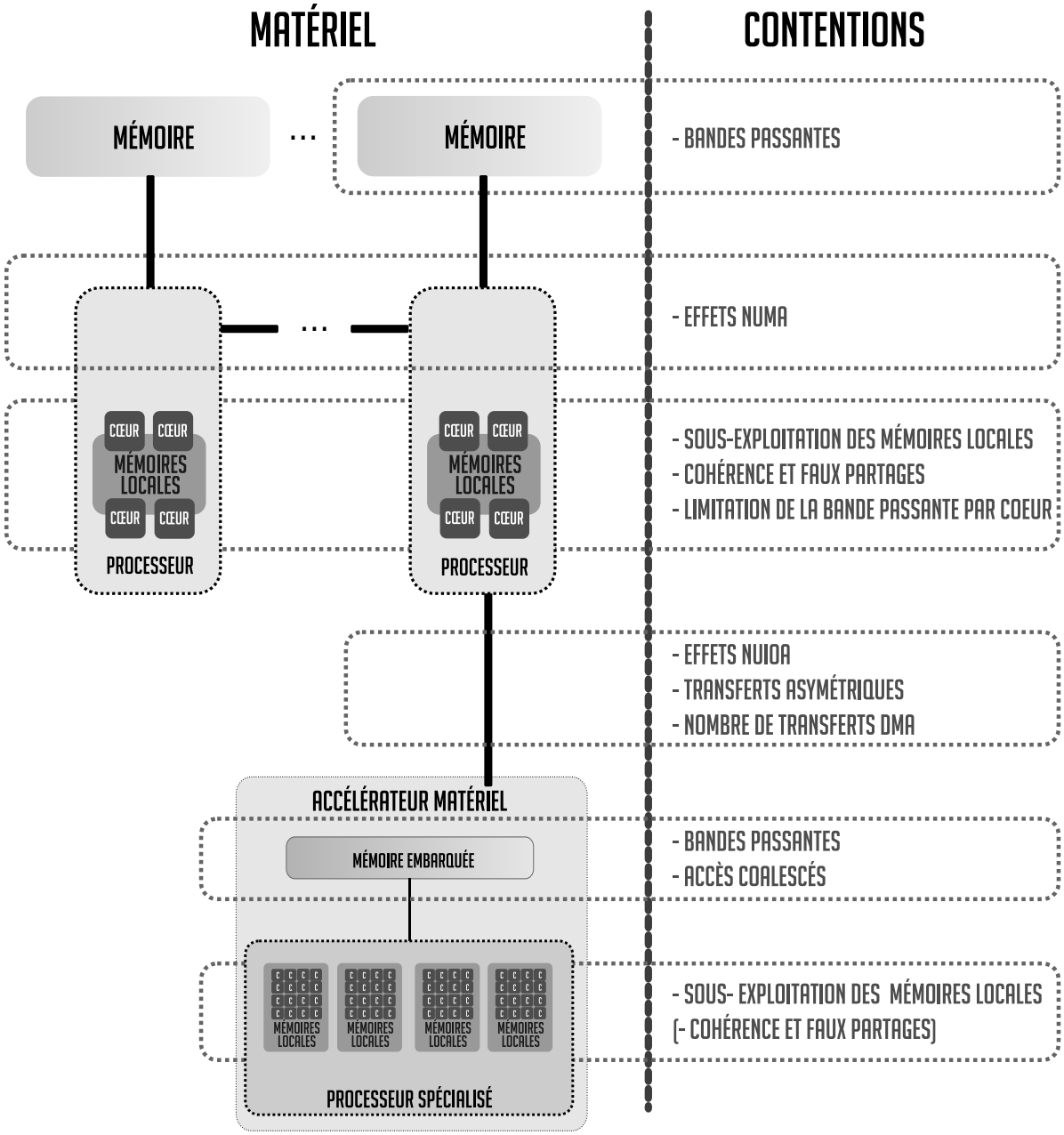


FIGURE 2.20 – Vue d'ensemble des contraintes matérielles d'accès aux données.

Accentuation de la hiérarchie mémoire

Les machines contemporaines misent plus que jamais sur les principes de localité spatiale et temporelle afin d'atteindre de hautes performances de calcul. Ces techniques conduisent à la fois à une accentuation de la hiérarchisation des machines et à la répartition d'éléments, autrefois sources de contentions. Il en résulte, entre autres, des performances d'accès disparates aux différentes parties qui composent la mémoire centrale.

Les deux principales limites qu'il faudra contourner pour accéder à l'*Exascale* sont la consommation électrique et les problèmes associés à la mémoire [92]. D'ici 2018, les mécanismes liés à la mémoire devront être améliorés afin de fournir une consommation énergétique par bit de l'ordre de trois à quatre fois inférieure aux machines pétaflopiques actuelles [93]. Pour atteindre les performances nécessaires en termes de latences, une intensification de la hiérarchisation de la mémoire sera vraisemblablement nécessaire. Ces architectures ne pourront être véritablement exploitées sans accentuer la localité des données dans les programmes. Des modèles de programmation associés à des moyens logiciels sont alors nécessaires pour accroître ces principes de localité.

Mémoires disjointes non cohérentes

La cohérence des mémoires caches sera vraisemblablement conservée puisqu'elle facilite la programmation des microarchitectures et ne représenterait qu'environ 4% de la consommation totale des cœurs de calcul [93]. En revanche, des accélérateurs bâtis autour de processeurs renfermant des centaines, voire des milliers de cœurs de calcul, ont fait leur apparition dans les machines sous la forme de cartes d'extension et viennent se greffer à une organisation déjà bien complexe. Ces matériels spécialisés embarquent leur propre hiérarchie mémoire qui n'est pas rendue cohérente avec celle du système qui les héberge. Désormais, des processeurs de natures différentes se côtoient dans ces machines devenues à la fois hétérogènes et asymétriques quant à la gestion des différentes mémoires.

Pour les prochaines générations de machines, l'emploi d'un processeur homogène ou d'un APU permettrait théoriquement d'accéder qu'à la mémoire centrale. Les mécanismes de cohérences de données seraient alors assurés matériellement. Le choix du type de mémoire directement connectée à ces nouvelles puces est néanmoins délicat. Recourir à une mémoire graphique de dernière génération (GDDR5) est une solution qui bénéficierait d'une large bande passante, plus en adéquation avec les architectures massivement parallèles. Cette technologie onéreuse ne permettrait sans doute pas d'y associer une aussi grande capacité qu'avec une mémoire plus classique (DDR3). Une hiérarchisation des mémoires centrales pourrait éventuellement voir le jour entre ces deux technologies afin d'apporter un compromis entre bande passante et capacité tout en favorisant un assemblage plus performant de plusieurs processeurs hétérogènes ou homogènes. De plus, des niveaux non cohérents peuvent toujours subsister entre la mémoire centrale et des mémoires locales programmables de type *scratchpad* (cf. **chapitre 2.4.3**, page 51).

À la lumière de ces différentes difficultés analysées, le troisième précepte du *Discours de la Méthode* pourrait être suivi, en établissant maintenant un ordre de pensées méthodiques. C'est exactement le rôle du programmeur, lequel, en s'appuyant sur des langages de programmation permettant une simplification et une certaine abstraction de la machine, va concevoir le programme de calcul souhaité. Il doit concilier, un à un, les différents aspects conduisant à de hautes performances, tels qu'un placement de données adapté, des accès aux données rapides par les unités d'exécution, une description des calculs adéquate et une répartition de la charge de travail performante. Le chapitre suivant aborde les problématiques d'expression des opérations en se focalisant sur la matérialisation d'un ordre de pensées pour résoudre un problème ; c'est-à-dire l'élaboration d'un logiciel de calcul à haute performance. Un écart important se creuse entre les performances que peuvent atteindre des programmeurs experts en optimisation face à d'autres développeurs. Des modèles de programmation permettant d'effacer une partie de ces différences, appelées parfois *"fossé du ninja"* [94], y sont présentés.

Chapitre 3

Les modèles de programmation parallèles

> *“ Le grand secret est qu'un orchestre peut en réalité jouer sans aucun chef d'orchestre. Bien sûr, un grand chef d'orchestre aura ce quelque chose qui les aidera à jouer ensemble et les unifiera. ”*
>
> Joshua Bell, violoniste américain

Le chapitre précédent s'est focalisé sur des mécanismes liés à la gestion de la mémoire. Bien qu'une exécution efficace d'un programme doit tenir compte de ces contraintes d'exploitation pour atteindre de hautes performances, les développeurs interagissent rarement directement avec le matériel. Des couches successives d'abstraction ont été élaborées pour minimiser les complexités sous-jacentes. Cette approche se base sur un triptyque composé d'un langage de programmation, un compilateur et d'un environnement d'exécution. La conception d'un code de calcul démarre par la spécification des opérations et la description de leurs enchaînements en s'appuyant sur un langage intelligible par un être humain. Ce code élaboré peut être ensuite *compilé* en un langage machine (ou langage intermédiaire), permettant une interprétation par les unités de calcul en s'appuyant sur l'environnement d'exécution. Pour des raisons historiques et parce que la majorité des processeurs est conçue pour exécuter des instructions élémentaires, le domaine du calcul à haute performance fait largement appel à la programmation impérative grâce notamment aux langages C, C++ ou Fortran. Ceux-ci, sont basés sur une déclaration d'instructions successives, propices à une exécution séquentielle des opérations.

Dans le cadre de la programmation parallèle, il s'agit de surcroît de morceler les traitements afin de permettre une exécution simultanée sur différentes unités de calcul. Cette description s'appuie sur des modèles de programmation qui permettent de gagner davantage en abstraction en se souciant uniquement de la décomposition et éventuellement de l'attribution du travail. À partir d'un langage de programmation donné, plusieurs modèles peuvent être combinés pour faire appel à différents types de parallélisme, lesquels peuvent être associés à des architectures spécifiques. L'expression globale du parallélisme est complexe car c'est à la fois la vitesse d'exécution et le bon déroulement de l'application qui sont en jeu. Les portions de codes non-parallélisées peuvent considérablement impacter les performances voire même annihiler toute chance de passage à l'échelle. Certains codes existants doivent être repensés pour intégrer, lorsque c'est possible, un degré de parallélisme suffisant. Le programmeur joue en quelque sorte le rôle d'un compositeur de musique en écrivant des partitions pour les unités de calcul. Chaque architecture interprète les instructions parallèles issues d'un modèle de programmation qui lui est plus adapté. Une coordination particulière est nécessaire pour une machine parallèle, comme cela peut être le cas lors du passage du jeu de solistes à la représentation d'un orchestre complet. L'objectif

est d'aboutir à un tout harmonieux et cohérent, c'est à dire en respectant des contraintes de performances d'exécution tout en obtenant un résultat correct. Cette conception possède un aspect créatif puisque, pour un même algorithme, chaque programmeur écrira les instructions et leurs enchaînements d'une manière différente pour aboutir à la même solution.

Lorsque toutes les sections parallèles sont créées, l'exécution par les unités de calcul peut s'opérer de manière *statique* ou *dynamique*, c'est à dire schématiquement avec ou sans chef d'orchestre. Lors d'une prise en charge statique, les instructions sont rigoureusement réparties de la façon dont le programme à été élaboré. Le premier inconvénient est que ce type de description dépend en général des caractéristiques de la machine ciblée. Une exécution sur une machine différente nécessite l'adaptation de certains paramètres. Ainsi, cette répartition offre en général peu de flexibilité et peut poser des problèmes de pérennité des performances. Deuxièmement, si un phénomène imprévu survient, les performances peuvent s'en trouver dégradées. Certains codes de simulation, caractérisés d'*irréguliers*, comportent des phases dont la quantité de calculs est variable et dont l'évolution est difficilement prévisible. Les machines employées deviennent également si complexes qu'il est parfois difficile d'anticiper certains phénomènes sources de ralentissement. Une prise en charge dynamique du traitement autorise une interprétation différente afin d'adapter les stratégies d'attribution des opérations en fonction du programme et de l'architecture employée. Une plate-forme d'exécution, ou support exécutif, permet cet équilibrage dynamique de charge tout en s'occupant d'autres mécanismes tout au long de l'exécution. Déléguer certaines opérations à une telle plate-forme permet de gagner en souplesse et de faciliter le travail du programmeur déjà bien complexe.

Ce chapitre introduit les contraintes associées à la programmation parallèle et hétérogène. La première section insiste sur la nécessité de concevoir des codes maximisant le degré le parallélisme. Les modèles de programmation parallèle à exécutions statiques ou ne permettant pas un équilibrage de charge élaboré sont abordés. Un exemple de code d'algèbre linéaire irrégulier est présenté en guise de motivation à l'emploi de modèles fondamentalement dynamiques, lesquels seront détaillés dans une troisième partie.

3.1 Des codes séquentiels aux codes parallèles

Depuis l'apparition des processeur multi-cœurs qui ont permis de contourner la contrainte imposée par le mur de la fréquence, l'accélération des codes séquentiels n'est plus automatiquement acquise. Les performances de chaque cœur n'augmentent plus aussi rapidement, et sont même quelques fois revues à la baisse pour tolérer une plus grande juxtaposition de cœurs sur un même support physique comme l'illustre la **figure 3.1** pour les accélérateurs Nvidia. Ce tournant technologique est particulièrement difficile pour les programmeurs. Comme le souligne métaphoriquement en 2005 Herb Sutter, architecte logiciel chez Microsoft et ex-secrétaire du comité de standardisation du langage C++, "*le repas gratuit est terminé*[64]" [95]. Cela signifie qu'un certain nombre d'efforts doit être désormais fournis pour espérer accélérer les codes de calcul. Les programmes requièrent un degré de parallélisme toujours plus important pour profiter des accélérations que peuvent théoriquement apporter les architectures récentes. En particulier, les portions de code qui ne sont pas parallélisées peuvent considérablement limiter les performances atteignables.

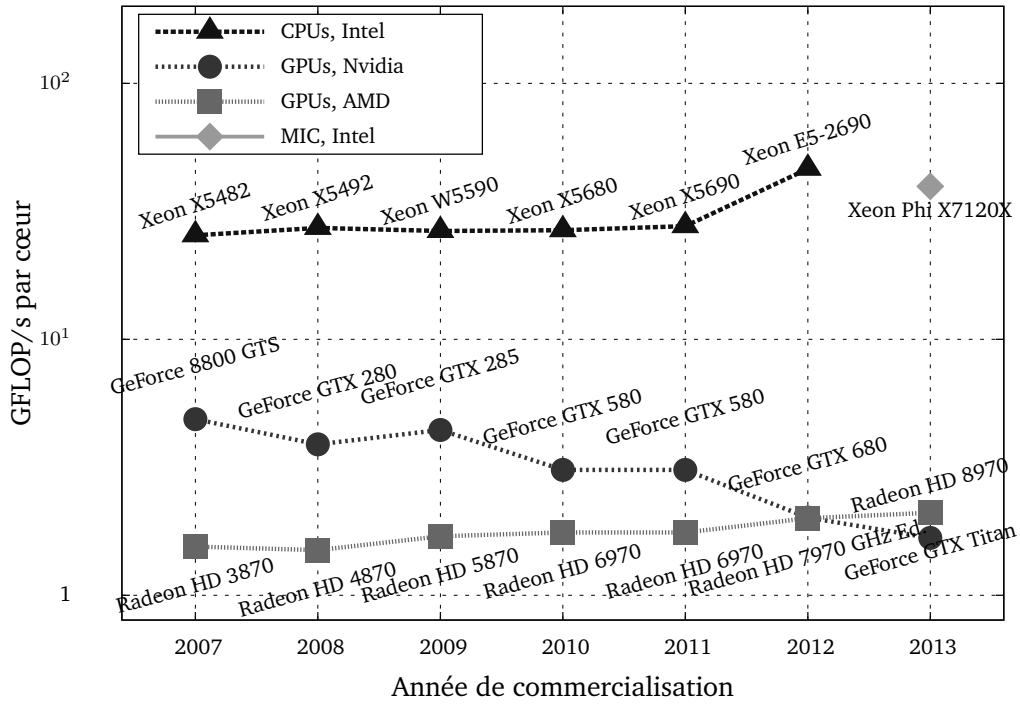


FIGURE 3.1 – Évolution de la puissance de calcul en GFLOP/s (simple précision) par cœur de calcul *(source [34])*.

[64]"*The free lunch is over*".

3.1.1 Impact d'une portion séquentielle dans un code parallèle

Concevoir un code parallèle n'est pas une tâche aisée. Obtenir de hautes performances à partir d'une telle expression l'est encore moins. Le chapitre précédent s'est focalisé sur les contraintes matérielles d'accès aux données par plusieurs ressources de calcul (cf. **chapitre 2**, page 33). À ces contraintes d'ordre matériel s'ajoutent des limitations d'expression du degré de parallélisme d'une application. En effet, les traitements séquentiels peuvent considérablement impacter les performances globales. Le syllogisme arithmétique énoncé par Ambrose Bierce peut aider à nous en convaincre. Il pose la question suivante :

> *"Si un homme peut creuser un trou avec une tarière en soixante secondes, en combien de temps soixante hommes peuvent ils creuser ce trou ?"*
>
> Ambrose Bierce, Le Dictionnaire du Diable [96], 1911

La réponse *soixante hommes peuvent creuser un trou avec une tarière en une seconde*, combinant à la fois logique mathématique et syllogisme, est naturellement incorrecte. Certaines opérations ne sont par nature pas divisibles et c'est précisément ce qui peut conduire à une baisse d'efficacité d'un code de calcul.

Passage à l'échelle fort

La loi d'Amdahl [97] (cf. **figure 3.2a**) permet de calculer l'accélération théorique maximale en tenant compte de la proportion d'opérations séquentielles dans un programme associé à une taille de problème donnée. Cette formule fournit une borne maximale des performances qui peuvent être espérées en fonction du nombre de processeurs employés. Il s'agit de l'étude d'un passage à l'échelle dit *fort*[65]. Cette loi peut être poussée à l'extrême en considérant une portion d'un code séquentiel occupant 10% du temps de calcul total et d'une quantité illimitée de processeurs ($P \to \infty$). Si cette infinité d'unités de calcul est mise à profit pour résoudre les 90% du travail restant, l'accélération maximale atteignable n'est que de 10 par rapport à une exécution séquentielle. La **figure 3.2b** illustre ces comportements asymptotiques.

a) Loi d'Amdahl

$$A = \frac{1}{\alpha + \frac{(1-\alpha)}{P}}$$

A : accélération
α : proportion séquentielle du programme
P : nombre de processeurs

b) Passage à l'échelle fort

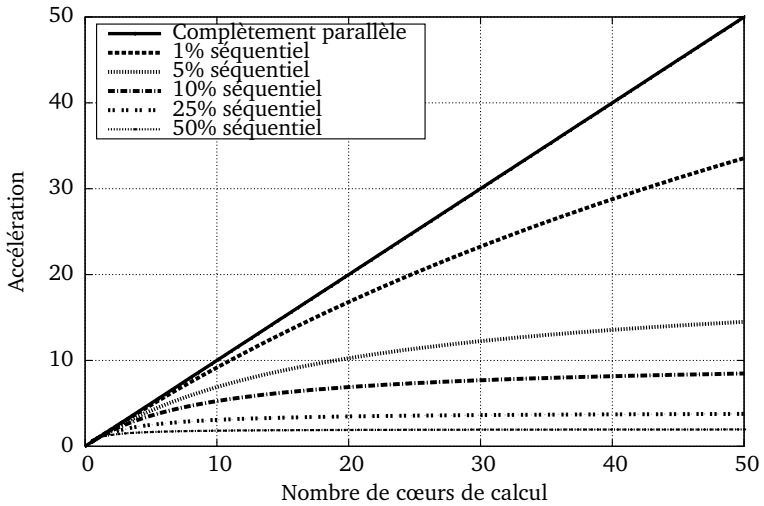


FIGURE 3.2 – Calcul de l'accélération théorique en *passage à l'échelle fort* à partir de la loi d'Amdahl.

[65] *Strong scalability* en anglais.

Passage à l'échelle faible

Le passage à l'échelle est ainsi fortement limité par les portions séquentielles. En réalité, la loi d'Amdahl ne tient pas compte du besoin de meilleures résolutions de calcul en élargissant la taille des problèmes. La loi de Gustafson [98] (cf. **figure 3.3a**) considère que l'accroissement du nombre de ressources de calcul implique également une augmentation des jeux de données. Il s'agit de l'étude d'un passage à l'échelle dit *faible*[66]. Cette loi révèle qu'un code comportant une portion séquentielle peut théoriquement bénéficier d'une accélération en multipliant les cœurs de calcul, à condition de faire croître la taille du problème (cf. **figure 3.3b**).

a) Loi de Gustafson

$$A = P - \alpha(P - 1)$$

A : accélération
α : proportion séquentielle du programme
P : nombre de processeurs

b) Passage à l'échelle faible

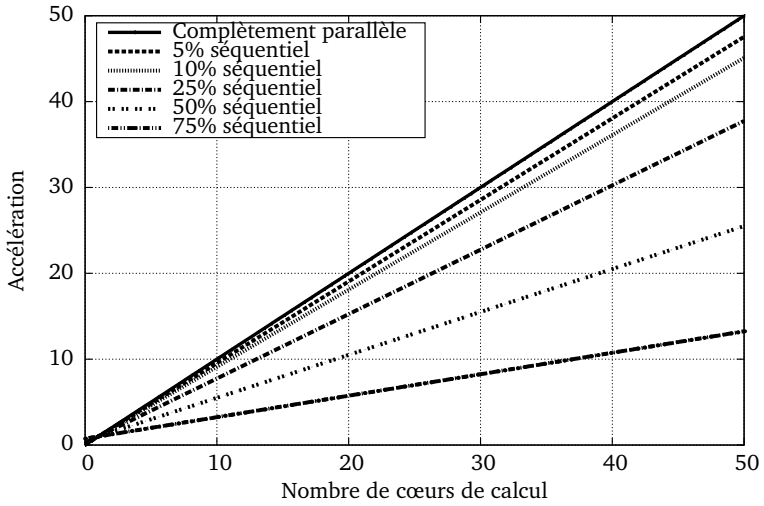


FIGURE 3.3 – Calcul de l'accélération théorique en *passage à l'échelle faible* à partir de la loi de Gustafson.

Ces formules ne considèrent bien entendu pas les pertes de performances liées aux synchronisations et aux accès aux données. Elles permettent tout de même de calculer une borne supérieure des performances potentiellement atteignables, en connaissant un certain nombre de paramètres. Dans tous les cas, la proportion de code séquentiel doit être réduite au maximum pour atteindre les meilleures performances possibles. Cela passe par une expression du parallélisme en s'appuyant sur des modèles de programmation. Il convient alors d'extraire, à différents niveaux, un maximum de parallélisme afin d'exploiter les divers mécanismes de calcul des machines contemporaines (cf. **chapitre 1.2**, page 20).

3.2 Les modèles de programmation parallèles statiques

Il existe plusieurs modèles de programmation qui permettent d'exprimer différentes formes de parallélisme, selon une sémantique plus ou moins complexe. Aucune adaptation de la répartition des opérations entre les unités de calcul n'est supportée nativement par les modèles abordés dans cette section. La répartition de la charge de travail est connue à l'avance, il s'agit d'une attribution *statique* des traitements.

[66] *Weak scalability* en anglais.

3.2.1 Les processus légers

Les premiers *processus* sont apparus suite aux réflexions menées dans les années 1960. L'objectif était de partager une ressource de calcul entre différents programmes. Il s'agit d'une programmation concurrente où des processus sont tour à tour activés et désactivés pendant un laps de temps donné. Cette concurrence peut également fournir l'illusion au programmeur que plusieurs traitements distincts s'opèrent de manière simultanée dans un environnement monoprocesseur, en changeant suffisamment rapidement de *contexte*. Par exemple, cela permet une prise en charge d'une interface graphique interactive conjointement à des traitements lourds. Le concept de *flot de contrôle* apparaît avec le *Berkeley Timesharing System* [99]. Au début des années 70, la notion de processus devient un fil de contrôle avec l'arrivée des systèmes d'exploitation UNIX. Chacune des instances peut échanger des informations via des tubes de communication[67] et des signaux. L'idée de *processus légers*[68] [100] apparaît par la suite et s'est développée avec l'expansion des systèmes parallèles. Chaque processus léger porte l'exécution d'un ensemble d'instructions sur un cœur de calcul. Plusieurs d'entre eux peuvent être contenus dans un même processus lourd et partager un espace mémoire virtuel commun, dans lequel ils ont la possibilité de s'échanger des informations. Ainsi, plusieurs cœurs de calcul peuvent être exploités simultanément. Une normalisation de l'interface a été proposée dans la fin des années 80 et porte le nom de *POSIX*[69] [101].Trois types de processus légers peuvent être distingués : les processus légers noyaux, les processus légers en espace utilisateur et les processus légers dits mixtes.

— Les processus légers s'appuyant sur des bibliothèques de niveau noyau impliquent l'intervention fréquente du système d'exploitation qui est chargé de gérer la durée de leur exécution respective avant rotation. La première implémentation de ce type apparaît en 1997, les *Linux Threads* [102] de Xavier Leroy. Cette implémentation fut remplacée par la Native POSIX Thread Library (NPTL [103]) qui encore utilisée dans Linux et qui permet l'utilisation des *PThreads* [104] par les programmeurs.

— Les bibliothèques de niveau utilisateur permettent, comme leur nom l'indique, une gestion des processus légers en espace utilisateur. Ceux-ci fonctionnent au dessus d'un unique processus noyau, aucune exécution parallèle n'est donc possible. L'avantage réside dans leurs changements de contexte qui peuvent être accélérés puisqu'ils ne requièrent plus directement l'intervention du système. GNU Portable Threads [105] et Green Threads [106] de Sun sont deux exemples de bibliothèques de niveau utilisateur.

— Enfin, les bibliothèques mixtes combinent l'avantage des deux bibliothèques en autorisant l'association des processus légers utilisateurs à plusieurs processus légers noyaux. Marcel [107], de la suite logicielle PM2 [108], et *MPC* qui sera abordé plus loin dans ce chapitre, font partie de ce genre de bibliothèques plus flexibles mais plus complexes à élaborer.

La programmation d'applications parallèles avec des processus légers est réputée difficile. Le développeur doit s'assurer que l'accès à certaines ressources partagées est correctement restreint afin d'éviter que l'état d'un processus ne devienne incohérent. Il est ainsi nécessaire de mettre en place des mécanismes de synchronisation complexes. Selon Herb Sutter, "*reconcevoir une application avec des processus légers pour qu'elle fonctionne sur des machines multi-cœurs, c'est comme apprendre à nager dans le grand bain*" [95].

[67] *Pipes* en anglais.
[68] *Threads* en anglais.
[69] **POSIX :** *Portable Operating System Interface* (le X est associé à l'héritage UNIX).

3.2.2 Le passage de messages

Ces modèles se basent sur un cloisonnent des données dont chaque morceau est affecté à un fil d'exécution. Plusieurs processus communiquent des résultats intermédiaires en s'envoyant des messages. Le programmeur est chargé de spécifier les données à envoyer entre les processus dont les unités de traitement sont interconnectées. Grâce à ce principe, des cœurs d'un même nœud de calcul à mémoire partagée ou de nœuds distincts peuvent être exploités. Il permet en particulier de recourir à une quantité de mémoire plus conséquente via un espace d'adressage plus vaste puisqu'il permet d'utiliser plusieurs nœuds de calcul à la fois. Les connexions sont prises en charge par les systèmes d'exploitation et nécessitent la mise en place de tampons mémoire de communication pour accueillir et préparer l'envoi des données véhiculées. À l'intérieur d'un même nœud, ce modèle présente l'avantage de ne pas induire d'effets NUMA lors des phases de calcul du fait de la copie locale des données échangées. En contrepartie, avec l'accroissement de la quantité de cœurs de calcul, la part mémoire occupée par les zones tampons augmente fortement et finit par poser de problèmes de surconsommation mémoire. En effet, les espaces mémoires monopolisés par ces zones tampons qui sont imposées par ces types de modèles de programmation, ne sont clairement pas toujours indispensables. Des données contenues dans une même mémoire physique peuvent être ainsi dupliquées plusieurs fois.

PVM

La bibliothèque de communication *Parallel Virtual Machine* [109] (PVM) est issue d'un projet interne de l'*Oak Ridge National Laboratory* en 1989 et permet d'établir des communications par passage de message en langage C et Fortran. Elle rend possible l'exploitation de machines hétéroclites connectées à un réseau : des nœuds de calcul élaborés avec des architectures bien distinctes peuvent alors travailler de concert. La bibliothèque est constituée de deux parties. Un processus *démon* est chargé d'établir les authentifications et de gérer les communications entre chaque instance. La bibliothèque expose également une interface permettant d'assembler et désassembler les messages, ainsi que d'autres fonctions effectuant des appels systèmes. Cela en fait un modèle simple et portable qui était apprécié par la communauté scientifique. Celle-ci s'est néanmoins progressivement tournée vers MPI, un modèle similaire plus efficace proposant des fonctionnalités plus avancées.

MPI

Un consortium d'industriels et d'équipes de recherche ont mutualisé leurs efforts pour proposer un standard de programmation par échanges de messages, capable de communiquer sur tout type de réseau. La norme *Message Passing Interface* (MPI [110]) est ainsi définie en 1993, suivie par MPI-2 [111] quatre années plus tard. MPI permet des communications variées par envois de messages synchrones ou asynchrones. Ce dernier mode, dit non bloquant, offre la possibilité de concevoir des programmes capables de recouvrir les phases d'échange de messages par du calcul. Une prochaine version du standard est en cours d'élaboration [112] et permettra, entre autres, des communications collectives non bloquantes et un interfaçage avec d'autres modèles de programmation.

Deux principales implémentations libres ont vu le jour *MPICH2* [113, 114] et *Open MPI* [115, 116]. Certains constructeurs de machines ont établi leur propre version sur une de ces bibliothèques libres. Ainsi, la bibliothèque MPIBull2 [117] est basée sur Open MPI et permet un meilleur support des machines *Bull* en offrant une prise en charge des spécificités de réseaux particuliers comme Infiniband [118]. IntelMPI [119] est une version développée par les équipes d'Intel et permet, par exemple, de recourir à ce modèle de programmation avec un coprocesseur *Intel Xeon Phi* [120, 121].

3.2.3 Les espaces d'adressage partitionnés (PGAS)

Le modèle par passage de messages impose aux développeurs de renseigner les transferts de données. De plus, lorsque ces données se trouvent déjà en mémoire du nœud de calcul, les transferts sont opérés même s'ils ne sont pas nécessaires. Le paradigme de programmation *PGAS*[70] [122] peut lever ces deux contraintes en permettant de contrôler les accès aux données de manière plus transparente, qu'ils soient locaux en mémoire partagée ou distants depuis des nœuds de calcul distincts. Pour ce faire, chaque processus possède ses propres espaces privé et partagé. Les données sont réparties et chaque espace contribue à constituer un espace global.

Co-Array Fortran [123] a été créé dans les années 90. Il étend syntaxiquement le langage *Fortran* pour décrire des opérations parallèles en décrivant la distribution des tâches et la répartition des données. Les accès distants restent cependant explicites. ***UPC***[71] [124, 125] est quant à lui une extension du langage de programmation *C* dont la première version a été officialisée en 2001. Il se base sur un unique espace d'adressage partitionné où les variables sont privées par défaut mais peuvent être déclarées comme partagées. Dans ce dernier cas, elles sont physiquement attachées à un unique processus léger mais restent accessibles, de manière transparente, par tous les autres. Berkeley UPC [126] et GNU-UPC [127] sont deux implémentations d'UPC.

La version asynchrone du modèle, appelée *APGAS*[72] [128] permet, entre autres, de déclencher des opérations distantes sur d'autres nœuds de calcul. Il introduit en particulier le concept de *places* qui définit une portion d'espace mémoire cohérente, associée aux activés des processus légers qui opèrent sur les données qui y sont contenues. Le langage ***X10*** [129], élaboré par *IBM*, est apparu en 2004 en introduisant ce nouveau paradigme. Une approche similaire est adoptée par le langage ***Chapel*** [130] développé par l'entreprise *Cray*.

3.2.4 Modèles pour accélérateurs

Les modèles suivants sont conçus pour décrire du parallélisme adapté à des accélérateurs massivement parallèles. Ces ressources de calcul, de par leur construction, nécessitent d'employer une instruction simultanément sur de multiples données (SIMD). En particulier, le *parallélisme de flux* de Bill Dally [131] offre une façon pragmatique de définir ce type d'exploitation. Une suite d'opérations, même dépendantes, peut être effectuée de manière *pipelinée* sur un flux de données. Ce type de parallélisme facilite la mise en place de la technique de la double copie en mémoire tampon (cf. **chapitre 2.4.1**, page 48).

CUDA

CUDA[73] [132] est un kit de développement propriétaire apparu en 2007. Il s'articule autour d'un modèle permettant de programmeur des accélérateurs graphiques de la marque Nvidia. La programmation doit satisfaire des contraintes liées au matériel. Les copies entre la mémoire du système et les mémoires déportées sont explicitement renseignées. Il est également nécessaire de s'appuyer sur plusieurs milliers de processus très légers. Généralement, ces processus légers sont regroupés au moins par paquets de 32, lesquels nécessitent d'emprunter le même chemin d'exécution pour atteindre de meilleures performances. Ces contraintes imposées par les architectures demandent un effort important d'optimisation pour exploiter pleinement ce type d'accélérateur [133].

[70] **PGAS :** *Partitioned Global Address Space*.
[71] **UPC :** *Unified Parallel C*.
[72] **APGAS :** *Asynchronous Partitioned Global Address Space*.
[73] **CUDA :** *Compute Unified Device Architecture*.

Des solutions permettent de convertir un code écrit en CUDA vers un code exploitable par d'autres architectures que celles conçues par Nvidia. L'intérêt est de conserver une application fonctionnelle sur un nœud de calcul dépourvu d'accélérateur de ce type. PGI CUDA x86 [134] et MCUDA [135] permettent de compiler un code CUDA pour des processeurs généralistes (architecture de type *X86*). Le projet Ocelot [136] est une plateforme de compilation dynamique (JIT[74]) qui propose des architectures cibles variées telles qu'un émulateur du langage pseudo assembleur PTX[75], des processeurs multi-cœurs via une traduction vers le compilateur LLVM [137], ou encore des processeurs graphiques AMD.

OpenCL

OpenCL[76] [138] est un modèle de programmation dérivé du C permettant de cibler des processeurs généralistes ou des accélérateurs. C'est une spécification ouverte initiée par Apple et standardisée en 2008 par le consortium *Khronos Working Group* qui compte une centaine de membres dont AMD, Intel, NVIDIA, SGI et Sun Microsystems. Ce modèle de programmation offre globalement les mêmes fonctionnalités que CUDA pour l'exploitation d'accélérateurs matériels, en supportant de surcroît un plus grand nombre d'architectures. Mis à part l'initialisation et la gestion des transferts qui requièrent une réécriture complète, un portage de CUDA vers OpenCL nécessite peu de changements. Les performances sont cependant légèrement en retrait lorsqu'il s'agit d'exploiter un accélérateur Nvidia par rapport à la solution propriétaire CUDA [139].

Autres modèles

En 2006, apparaît l'interface de bas niveau CTM[77] [140] qui fournissait un accès au jeu d'instructions natif des processeurs graphiques ATI[78]. En 2007 ATI acquière Brook [141], une plate-forme développée par l'université de Stanford. Cette plate-forme possède un modèle de programmation de flux pour accélérateurs permettant un plus haut niveau d'abstraction en se basant sur une forme de langage C étendu. Elle est alors optimisée pour les processeurs ATI et rebaptisée Brook+ [142]. Depuis, AMD a choisi de mettre davantage en avant la solution OpenCL dans son kit de développement.

Exploitation à base de directives

Des programmeurs souhaitent parfois accélérer un code de calcul conséquent, élaboré à l'origine pour des processeurs généralistes. Réécrire complètement une telle application n'est souvent pas envisageable puisque cela demanderait trop d'efforts et serait trop coûteux. Un portage partiel et progressif est alors plus adapté. Des outils permettent d'accompagner le programmeur à réaliser cette tâche en lui fournissant une forme d'abstraction. Des modèles de programmation par directives ont ainsi vu le jour et permettent au développeur d'ajouter simplement des annotations au niveau des portions de code à accélérer. Ces informations sont ensuite interprétées par un compilateur qui génère un programme dans un modèle plus complexe de manière transparente. Ces modèles sont bien adaptés à des codes suffisamment réguliers. Ils demandent souvent un effort quasiment comparable à une réécriture dans le modèle abstrait lorsqu'il s'agit d'optimiser les performances de sections de codes plus compliquées.

hiCUDA[79] [143] est une approche permettant de produire du code CUDA à partir de telles directives. Bien que cette approche ne masque pas la complexité de programmation, elle apporte

[74] **JIT :** *Just In Time* (compilation juste à temps).
[75] **PTX :** *Parallel Thread Execution*.
[76] **OpenCL :** *Open Computing Language*.
[77] **CTM :** *Close To Metal*.
[78] ATI a été racheté par AMD en 2010, les processeurs graphiques ATI et AMD font donc références au même matériel.
[79] **hiCUDA :** *high-level CUDA*.

tout de même un gain de productivité en supportant une adaptation progressive des applications existantes sans engendrer de perte de performance importante par rapport à l'emploi direct de CUDA. Elle permet en particulier d'extraire les opérations déportées dans des fonctions qui pourront être prises en charge par CUDA et de générer les appels aux fonctions de transfert.

PGI Accelerator [144], fournit par The Portland Group, et **HMPP**[80] [146] développé par CAPS entreprise sont des produits commerciaux qui permettent d'accroître l'abstraction en allégeant la quantité d'annotations nécessaires, notamment en ce qui concerne les transfert de données.

OpenACC [147] est un standard de programmation émergeant pour l'exploitation hétérogène de processeurs généralistes et d'accélérateurs à base de directives. La norme est définie par un groupe de travail formé d'entreprises telles que CAPS, Cray, Nvidia et PGI. En 2013, des solutions commerciales voient le jour ainsi qu'un compilateur ouvert (accULL [148]), élaboré à l'université de *La Laguna*. En 2013, la norme 2.0 est établie et étend la sémantique pour permettre un meilleur support des accélérateurs Nvidia, AMD et Intel notamment en améliorant la gestion des transferts de données.

Pour résumer cette section, plusieurs types de parallélismes statiques ont été décrits et permettent d'exploiter plusieurs nœuds de calcul, des cœurs partageant la même mémoire, une combinaison des deux ou bien encore des accélérateurs massivement parallèles. Ces moyens d'expression du parallélisme peuvent servir de briques de base pour des modèles offrant plus de flexibilités quant à l'ajustement dynamique de la charge de travail. Ces modèles adaptatifs sont présentés dans la section suivante.

3.3 Les modèles de programmation parallèles dynamiques

Il devient de plus en plus difficile pour les programmeurs de calibrer de manière appropriée la répartition des calculs du fait de la complexification des machines. Parfois, des codes irréguliers ne peuvent pas s'appuyer sur ces définitions statiques et requièrent un comportement dynamique pour minimiser les périodes d'inactivités de ressources de calcul disponibles. Des *ordonnanceurs* basés sur des modèles de programmation spécifiques permettent un équilibrage de charge dynamique au cours de l'exécution. Ils contribuent à réduire l'écart de performances entre les programmes élaborés par des programmeurs experts et ceux établis par les autres développeurs[81], en prenant des décisions "en ligne" pour réduire au global le temps de calcul. C'est désormais sur ces chefs d'orchestre que reposent, en partie, les performances de l'application. Or, cette gestion dynamique entraîne inévitablement un surcoût puisque les prises de décision demandent des cycles processeurs qui auraient pu être employés pour du calcul. Ce surcoût doit être aussi négligeable que possible face au gain apporté. Une personne anonyme a déclaré aux Victoires de la Musique Classique de 1997 : « *Les chefs orchestre se divisent en trois catégories : – ceux qui laissent jouer – ceux qui font jouer – ceux qui empêchent de jouer*». Il en va de même pour les plate-formes d'ordonnancement qui doivent à tout prix favoriser une cohabitation adéquate des unités d'exécution et éviter d'être à l'origine d'un ralentissement du calcul. Des techniques d'équilibrage de charge efficaces sont ainsi élaborées.

[80] **HMPP :** *Hybrid Multicore Parallel Programming* - OpenHMPP [145] est un standard ouvert basé sur la verison 2.3 d'HMPP.

[81] *Fossé du ninja* cf. fin du **chapitre 2** (page 33).

3.3.1 Modèles à équilibrage de charge pour ressources de calcul homogènes

Les Modèles de programmation présenté dans cette sous section offrent un support de processeurs généralistes d'un même nœud de calcul à mémoire partagée. Ils fonctionnent indépendamment du nombre de processeurs à exploiter. Cependant, ils ne possèdent pas de support natif de matériel spécialisé.

OpenMP 2.5

Le consortium *OpenMP ARB*[82] composé d'industriels et de laboratoires de recherche aboutissent en 1997 au standard *OpenMP*[83]. OpenMP définit des directives qui permettent de définir rapidement du parallélisme dans une application existante. Une transformation de code est opérée à la compilation et une bibliothèque permet de supporter le parallélisme en s'appuyant sur une bibliothèque de processus légers. La norme *OpenMP 2.5* [149] est officialisée en 2005 et supporte les langages *C, C++ et Fortran*. Des directives avec des attributs adéquats quant à la visibilité des variables (privées ou partagées) permettent de paralléliser les itérations d'une boucle et d'autoriser un équilibrage dynamique. D'autres fonctionnalités peuvent être employées comme définir des sections d'exclusion mutuelles afin d'assurer la cohérences de certains calculs. *OpenMp* possède un modèle d'exécution de type "*Fork/Join*", où les processus légers sont exploités en début de région parallèle marquant le passage d'une phase séquentielle à parallèle (fork), puis après s'être synchronisés, ils deviennent inactifs pour poursuivre un exécution séquentielle du flot d'exécutions (*join*).

Le parallélisme de tâches de calcul

Le parallélisme de tâches de calcul permet de regrouper des lots d'opérations qui seront effectuées successivement par une même unité d'exécution sur des données bien identifiées. L'association de ces tâches peut être effectuée statiquement par une analyse préalable du code ou dynamiquement grâce à un support exécutif. C'est cette dernière possibilité qui est explorée dans cette sous-section. Le modèle par tâche de calcul permet en particulier de simplifier les mécanismes de synchronisation entre les unités de calcul lors d'une gestion dynamique. Certaines approches se basent sur un fonctionnement dit *synchrone par lots*, c'est-à-dire que des tâches indépendantes sont créées et un point de synchronisation est nécessaire pour attendre que toutes les tâches aient bien été exécutées avant de générer une nouvelle un série de tâches. Une autre approche consiste à générer plus de tâches et de résoudre dynamiquement les dépendances de données. Les tâches sont débloquées au fur et à mesure lorsqu'elles peuvent être traitées.

Cilk [150] est élaboré en 1994 au MIT[84]. Ce modèle de programmation étend le langage C pour exprimer du parallélisme de tâches en mode *fork/join*. Cilk construit dynamiquement un *graphe dirigé et acyclique* (DAG[85]), afin d'exécuter en priorité les tâches qui sont sur le chemin critique. Ce modèle emploie aussi une équilibrage de charge par *vol de tâches*. Chaque unité d'exécution logique (s'exécutant dans à un processus léger) possède une liste de tâches à exécuter. Elle les traite par ordre d'arrivée dans la liste. Lorsqu'une unité a épuisé son stock de tâche à traiter, elle est va en "*voler*" une dans une liste d'une autre unité aléatoirement déterminée (appelée "*victime*"). Elle la récupère cette fois-ci depuis l'autre bout de la liste afin de ne pas perturber le travail de la *victime*. Ce modèle a été conçu à l'origine pour des processeurs mono cœurs reliés à une mémoire à accès uniforme. De nos jours, ce modèle souffre d'un manque de flexibilité quant à la gestion de la localité des données, en particulier pour éviter les effets *NUMA* et tirer partie de mémoires locales partagées. En 2006, l'entreprise *Cilk Arts, Inc* commercialise

[82] **OpenMP ARB :** *OpenMP Architecture Review Board*.
[83] **OpenMP :** *Open Multi Processing*.
[84] **MIT :** *Massachusetts Institute of Technology*.
[85] **DAG :** *Directed Acyclic Graph*.

Cilk++, une version commerciale supportant le langage C++. Cette version est rachetée par *Intel* en 2009 et étendue pour devenir *Intel Cilk plus* [151].

Intel TBB[86] [152] apparaît en 2006 après l'introduction du premier processeur *X86* à double cœurs d'*Intel*. *TBB* permet de définir du parallélisme de tâches grâce à une bibliothèque en *C++*. Tout comme *Cilk*, ce modèle a recours au *vol de tâches* pour équilibrer dynamiquement la charge de travail. Il fournit des conteneurs *C++*, des algorithmes parallèles (scan, réduction, ...) et facilite l'interopérabilité avec d'autres modèles tels que *Cilk* ou *OpenMP*.

En 2008, le standard ***OpenMP 3.0*** [153] supporte le parallélisme de tâches en mode synchrone par lots. Les tâches et barrières de synchronisation sont définies par des directives. Ce modèle de programmation manque également d'information sur le placement et l'affinité des données pour tenir compte des contraintes de la mémoire associées aux nœuds de calcul actuels.

Modèles à équilibrage de charge hybrides

Un support exécutif associé à l'implémentation d'un modèle de programmation cherche à exploiter au maximum les ressources de calcul qui lui sont mises à disposition. Lorsque plusieurs supports coexistent, leur concurrence d'accès aux ressources peut devenir contre productive et dégrader les performances globales. Des solutions facilitent l'association de plusieurs modèles de programmation.

L'environnement de développement libre ***MPC***[87] [154] facilite la cohabitation de plusieurs modèles de programmation pour une meilleure exploitation de grappes de calcul multi-cœurs NUMA. Cet environnement est élaboré conjointement par le *CEA* et le laboratoire *ECR*[88] et vise, entre autres, à résoudre le problème de concurrence de plusieurs supports exécutifs. *MPC* repose sur des *processus légers mixtes* (cf. **chapitre 3.2.1**, page 60) non-préemptifs pour gérer les processus des différents support exécutifs. Un ordonnanceur unifié permet schématiquement d'arbitrer quelle plate-forme d'exécution a la main sur les ressources de calcul. *MPC* fournit ainsi une implémentation de la norme *MPI 1.3* en reposant sur ces processus légers mixtes. Elle permet notamment une réduction de consommation mémoire pour des exécutions à grande échelle [155]. Une implémentation de la norme d'*OpenMP 2.5* est aussi intégrée à l'environnement de développement et est établie sur les mêmes fondations [156].

3.3.2 Modèles à équilibrage de charge pour ressources de calcul hétérogènes

Les modèles de programmation présentés dans cette sous section proposent une gestion de CPUs et d'accélérateurs. Certains permettent un équilibrage de charge dynamique entre des ressources de calcul hétérogènes.

XKaapi

XKAAPI[89] est une plate-forme d'exécution pour grappes de calcul qui est développée par l'équipe MOAIS[90]. Elle est basée sur le modèle de programmation *Athapascan* [157]. Initialement nommée Kaapi [158], son implémentation a évolué pour offrir plusieurs améliorations comme la réduction des coûts de création et de synchronisations liées au parallélisme de tâches. Ainsi, la plate-forme intègre un ordonnanceur particulièrement performant pour traiter des tâches à

[86] **TBB :** *Intel Threading Building Blocks*.
[87] **MPC :** *Multi-Processor Computing, sous licence française Cecill-C*.
[88] **ECR :** *Exascale Computing Research*, issu d'une collaboration entre le CEA, GENCI, Intel et l'UVSQ.
[89] **Kaapi :** *Kernel for Adaptative, Asynchronous Parallel and Interactive programming interface*.
[90] **MOAIS :** *Parallel Algorithms, Programming Models, Scheduling and Interactive Computing*.

granularité fine. Elle emploie un vol de tâches collaboratif [159] basé sur des verrous fonctionnant par *agrégation plate* [160], élaborés pour réduire les invalidations des mémoires locales et les surcoûts des synchronisations associés aux mécanismes de vol. La génération des tâches et la résolution des dépendances se fait “en ligne", lors des requêtes de vol, afin de réduire l'impact des instructions de création sur les performances. Il est possible de définir des tâches imbriquées. Ainsi, un affinage de la granularité peut être utilisé, mais sans gestion distincte de l'équilibrage de charge entre les tâches des différents niveaux.

La plate-forme a été tout d'abord modifiée pour supporter des accélérateurs sur une application itérative [161]. Le choix d'un accélérateur ou d'un cœur généraliste est guidé par l'attribution de *poids*, lesquels permettent d'atteindre, dans certains cas, des accélérations “collaboratives”. Le vol de tâches est guidée par la localité des traitements en s'inspirant des travaux de A. Umut et al. [162]. Cette localité se base sur les accès aux tâches d'une même *partition* mais ne permet pas de capturer les contraintes d'accès non-uniformes à la mémoire. La plate-forme a été étendue [163] pour intégrer des caches logiciels qui évincent en priorité les données les moins récemment utilisées (*LRU*[91]) en mode écriture tardive (*write-back*). Une politique d'ordonnancement dédiée permet de réduire le nombre d'invalidations de ces caches logiciels.

OMPSs

Le projet *OMPSs* [164] intègre les avancées développées autour de StarSs[92] [165] par le *Barcelona Supercomputing Center*. Il fournit un jeu de directives permettant la déclaration de tâches de calcul hétérogènes. Elles viennent enrichir la norme OpenMP pour assurer une interopérabilité notamment avec *CUDA*. Les directives comportent des informations telles que les données impactées par chaque tâche et leur type d'accès. Elles sont interprétées par le compilateur source à source *Mercurium* [166]. La plate-forme d'exécution *Nanos++* [167] ordonnance ces tâches en résolvant les dépendances dynamiquement et en gérant les transferts de données nécessaires. *StarSs* a été décliné sous de nombreuses formes :

— **SMPSs** [168] permet d'exploiter des processeurs multi-cœurs en s'appuyant sur un équilibrage dynamique par vols de tâches. Une fonctionnalité récente [165] permet de mémoriser le processus qui a touché un bloc de données pour la première fois. Une affinité est conservée tout au long de l'exécution pour les tâches qui dépendent ensuite de ce bloc.

— **CellSs** [169] est élaboré pour exploiter un processeur hétérogène *Cell*. La génération des tâches, la résolution des dépendances et l'ordonnacement est géré par la sous unité *PowerPC Processing Element* (PPE) alors que l'exécution des tâches est prise en charge par les *Synergystic Processing Elements* (SPEs). Ainsi, il n'y a pas d'exécution hétérogène des calculs. Les tâches sont regroupées par lots pour chaque PPE afin d'optimiser les transferts vers des caches logiciels permettant d'effectuer de la rétention de données dans des mémoires programmables disjointes.

— *SMPSs* et *CellSs* ont été fusionnés pour aboutir à un ordonnancement hiérarchique [170] capable d'exploiter plusieurs processeurs *Cell*. Une décomposition à grosse granularité permet une première répartition entre les processeurs puis une décomposition plus fine permet d'exploiter les unités hétérogènes avec *CellSs*.

— **ClusterSs** [171] étend *StarSs* pour supporter plusieurs nœuds de calcul distribués en s'appuyant sur la combinaison des modèles de programmation *PGAS* et tâches de calcul.

— **GPUSs** [172, 173] est basé sur une extension de *CellSs* pour supporter des processeurs graphiques. Une directive supplémentaire est introduite et permet de spécifier l'architec-

[91] **LRU :** *Least Recently Used*.

[92] **StarSs :** *Star SuperScalar*.

ture ciblée lors de la déclaration de la tâche. Par conséquent, il n'y a pas d'équilibrage de charge hétérogène. La plateforme d'exécution ne capture pas non plus la hiérarchie mémoire de la machine. En effet, l'ordonnancement est organisé autour d'une unique liste de tâches. Des caches logiciels permettent de contrôler la rétention de données dans les mémoires déportées en suivant la politique d'éviction *LRU*.

StarPU

StarPU [174, 175] est une suite logicielle intégrant une plate-forme d'ordonnancement de tâches pour des nœuds de calcul hétérogènes. Cette dernière s'appuie sur des extensions en C permettant de définir des abstractions de tâches appelées *codelets*. Contrairement à OmpSs, StarPU permet des exécutions hétérogènes d'une tâche, le choix de l'unité cible est décidée dynamiquement. Tout comme StarSs et XKaapi, les dépendances de données sont résolues à l'exécution, la cohérence des données est cependant assurée par un protocole de type MSI (cf. **chapitre 2.2.2**, page 38). Des caches logiciels à évictions LRU permettent également d'accroître la localité temporelle des mémoires déportées. De bonnes performances sont atteintes pour des opérations d'algèbre linéaire dense [176], grâce à des accélérations collaboratives. Ces dernières permettent de dégager de meilleures performances d'une exécution hétérogène par rapport à la somme des performances obtenues à partir d'ordonnancements n'utilisant qu'un type d'architecture à la fois. Des accélérations similaires sont constatées avec les méthodes proposées dans ce manuscrit, mais pour des raisons différentes. L'ordonnancement peut être sélectionné parmi un panel de politiques prédéfinies, voire même personnalisé afin de répondre à des besoins spécifiques d'une application.

La plupart des politiques d'ordonnancement proposées sont dérivées du HEFT[93], qui vise à assigner chaque tâche à l'unité de calcul qui sera en mesure de la terminer au plus tôt. Pour cela, des métriques sont collectées pour estimer les temps d'exécution de chaque type de tâche [178] et la durée approximative des transferts. Il s'agit de méthodes gloutonnes qui ne sont pas capables de faire des sacrifices à court terme pour gagner en performance à long terme. De plus, il peut arriver que les modèles de coûts employés soient faussés. Les durées d'exécution peuvent varier en fonction des temps d'accès aux données [179] et la quantité de calcul embarquée dans une tâche peut être irrégulière. De plus, les mécanismes d'attribution des tâches reposent sur une liste centralisée. L'ordonnancement est géré à plat, sans tenir compte du placement des données en mémoire centrale et des effets NUMA associés. Les unités accélératrices sont en concurrence avec les unités associées à chaque cœur de processeur généraliste. Bien que l'utilisateur puisse recourir à de multiples tâches de natures différentes pour exploiter différemment les ressources de calcul hétérogènes, il n'y a pas de mécanisme hiérarchique permettant de contrebalancer l'écart important de puissance de calcul à partir d'un seul type de tâche.

Autres travaux

Charm++ est une bibliothèque parallèle en C++ fournissant des optimisations sur des mécanismes de communication. La bibliothèque emploie un équilibrage de charge conçu pour exécuter en parallèle des objets appelés *chares* [180, 181]. Des extensions supportent des machines hétérogènes comme des processeurs CELL [182] et des processeurs graphiques [183]. Un ordonnanceur hiérarchique sert à équilibrer la charge entre plusieurs nœuds de calcul.

Qilin [184] permet un ordonnancement hétérogène de portions de code. Il offre des mécanismes automatiques de division de tableaux. Les portions de code sont compilées pour un support de TBB pour les processeurs centraux et par CUDA pour les processeur graphiques. Des modèles de performances entre en scène pour attribuer les tâches de calcul aux différentes unités de calcul. Ces modèles peuvent se baser sur des mesures de performances d'exécution antérieures pour affiner la division et la répartition des opérations associées.

[93] **HEFT :** *Heterogeneous Earliest Finish Time* [177].

3.4 Codes irréguliers et exemple récurrent

Des codes de calcul peuvent contenir des schémas d'accès irréguliers à des données ou exploiter une quantité d'information variable au cours de l'exécution. Ces programmes de calculs déséquilibrés sont complexes à concevoir pour qu'ils puissent s'exécuter efficacement en minimisant les temps d'inactivité des unités de calcul. Le terme de creux renvoie à la présence de valeurs nulles dans des structures telles que des vecteurs ou des matrices. Ces zéros permettent, dans certains cas, de réduire la quantité de calculs selon le traitement effectué. Les données contenues dans les structures peuvent également varier au cours de l'exécution et considérablement modifier le nombre d'opérations si elles deviennes nulles. Cette section décrit un algorithme de factorisation LU pour matrices creuses qui sera exploité en tant d'exemple récurrent tout au long de ce manuscrit. La version dense de cet algorithme est détaillée dans un premier temps avant d'aborder les spécificités de la version creuse et d'une adaptation permettant l'exploitation d'accélérateurs déportés.

3.4.1 Factorisation LU de matrices par blocs

La factorisation, ou décomposition LU permet de résoudre des systèmes linéaires. Elle est encore employée en tant qu'étape principale pour déterminer l'inverse d'une matrice ou calculer son déterminant. Soit une matrice $A \in \mathbb{R}^{n \times n}$, une factorisation LU calcule L et U tel que $A = LU$ où L est une matrice $n \times n$ triangulaire inférieure et U triangulaire supérieure. Les matrices peuvent être délimitées par blocs et s'appuyer sur des algorithmes ayant recours à des bibliothèques de noyaux optimisés pour certaines étapes. Plusieurs travaux prônent l'utilisation des matrices par blocs [185, 186, 187] pour atteindre de meilleures performances de calcul et de localité. Certaines séparations par blocs peuvent peut être abordées de manière récursive afin de gagner en flexibilité et intensifier la localité spatiale [188, 189]. Une matrice A contenant $N \times N$ blocs peut être définie ainsi :

$$A = \left(\begin{array}{c|c|c|c} A_{0,0} & A_{0,1} & \dots & A_{0,N-1} \\ \hline A_{1,0} & A_{1,1} & \dots & A_{1,N-1} \\ \hline \vdots & \vdots & \ddots & \vdots \\ \hline A_{N-1,0} & A_{N-1,1} & \dots & A_{N-1,N-1} \end{array} \right)$$

Algorithme sans pivot

Aucun pivot n'est utilisé dans la version de factorisation présentée. Les pivots et pivots partiels sont généralement employés pour accroître la stabilité et prévenir l'apparition de zéros sur la diagonale qui ferait échouer le processus de décomposition LU. Lors des phases d'évaluations dans les chapitres suivants, les matrices sont choisies pour qu'aucun zéro ne survienne sur la diagonale. Bien que le but ne soit pas de proposer un nouveau solveur, certaines techniques statistiques pourraient être employées pour accroître la stabilité de cet algorithme sans recourir à des pivots. Par exemple, la méthode PRBT[94] [190] présentée par Baboulin et al. permet d'éviter les pivots dans la factorisation LU par blocs, tout en permettant une précision proche de la solution par pivots partiels.

[94] **PRBT :** *Partial Random Butterfly Transformation*.

La figure 3.4 formalise la version appelée *"right-looking"* de l'algorithme avec et sans blocs, en utilisant la notation *FLAME*[95] [191, 192]. À chaque itération de l'algorithme, les opérations font apparaître, en place, une partie des matrices résultantes L et U ; c'est à dire en replaçant progressivement les valeurs de la matrice A.

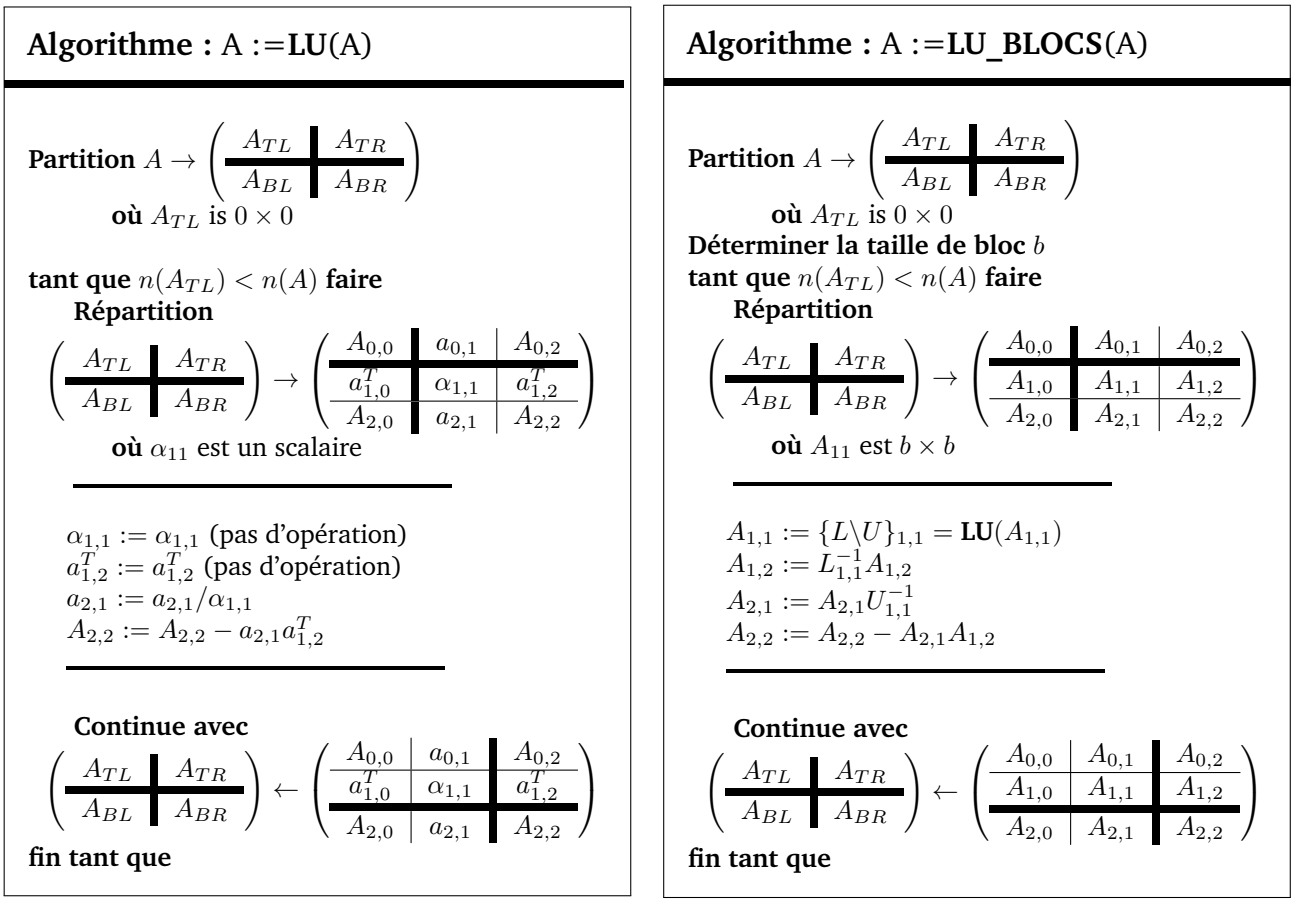


FIGURE 3.4 – Algorithme de factorisation LU sans pivot comme présenté dans l'article [193]
a) Algorithme sans bloc. **b)** Algorithme par blocs.

Les opérations de cette version par blocs permet de reposer la majorité des calculs sur des noyaux optimisés de type *BLAS* [194, 195, 196] implémentées par des bibliothèques de performance telles qu'*Intel MKL* et *AMD ACML* (cf. chapitre 2.2.1, page 37). À chaque itération, trois étapes interdépendantes peuvent être démarquées. Une factorisation LU élémentaire est réalisée sur un bloc situé sur la diagonale. Le résultat est utile pour effectuer des résolutions triangulaires avec le noyau optimisé *TRSM*. Enfin, les données calculées servent à la troisième étape faisant appel à des multiplications en s'appuyant sur la routine optimisée *GEMM*. Lorsque la matrice A est suffisamment grande, la majorité des calculs sont concentrés dans ces multiplications de blocs. Les performances de la factorisation dépendent de la bonne gestion de la localité des données et d'une répartition parallèle adéquate des opérations sur les différentes unités de calcul. La figure 3.5 présente les blocs impactés à chaque itération par les différentes étapes sur une matrice constituées de 3×3 blocs. Les étapes sont identifiées par le nom des noyaux optimisés.

[95] **FLAME :** *Formal Linear Algebra Methods Environment*.

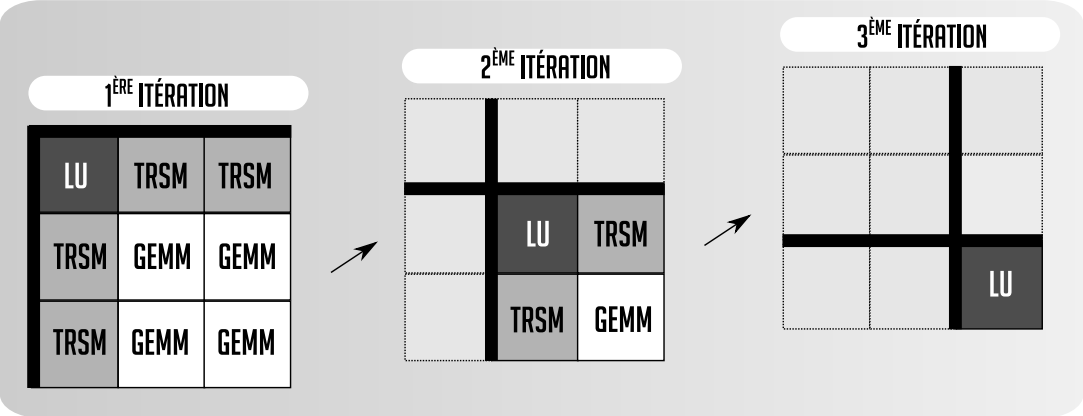


FIGURE 3.5 – Les trois étapes identifiées par le nom des noyaux employés au cours d'une factorisation LU sans pivot d'une matrice composée 3×3 blocs (3 itérations nécessaires).

Travaux connexes

Des bibliothèques de calcul fournissent des implémentations de décompositions LU pour des processeurs généralistes. *LAPACK*[96] [197], développée à l'origine en Fortran 77 puis en Fortran 90 en 2008, offre des fonctions optimisées pour machines à mémoire partagée. *ScaLAPACK*[97] [198, 199] emploie la même interface est permet une exécution sur plusieurs nœuds de calcul. *FLAME* [200, 201] diffère de LAPACK en proposant des notations à plus haut niveau d'abstraction et ayant recours à un ordonnanceur de tâches de calcul (*SuperMatrix* [202]) pour mieux exploiter des processeurs multi-cœurs. *PLASMA*[98] [203] fournit un algorithme de découpage de matrices en blocs de tailles adéquates, permettant de minimiser les défauts de cache, le faux partage et d'accroître le potentiel de préchargement matériel des données. Les fonctions sont organisées en tâches de calcul s'appuyant sur ces décompositions et gérées à l'exécution par l'ordonnanceur QUARK[99] [204]. Ce dernier prend notamment en compte les dépendances de données en exploitant un DAG des tâches. *DPLASAMA*[100] [205] est une version distribuée permettant l'exploitation de plusieurs nœuds de calcul.

Des études révèlent que la factorisation LU sur matériel spécialisé peut mener à des accélérations importantes. Tomov et al. ont proposé des techniques appliquées à la décomposition LU et menant à un bon équilibrage de charge entre processeurs génériques et graphiques [206]. Ils ont élaboré un nouvel algorithme visant à limiter l'utilisation de pivots. Puis, par expérimentations, ils ont trouvé le meilleur équilibrage statique pour atteindre les performances voulues. Cet ordonnancement nécessite donc une connaissance de la charge de travail à chaque itération. Song et al. [207] utilisent également une répartition statique après avoir décomposé la matrice en partitions rectangulaires réparties entre processeurs graphiques et processeurs multi-cœurs. Une exploitation hétérogène efficace peut être atteinte mais requiert une phase de préréglage afin mieux équilibrer la charge de travail. Leur méthode devient plus difficile à mettre en œuvre avec plusieurs accélérateurs. Kim el al. présentent *BS2*[101] [193], un ordonnanceur basé sur des

[96] **LAPACK :** *Linear Algebra PACKage*.
[97] **ScaLAPACK :** *Scalable Linear Algebra PACKage*.
[98] **PLASMA :** *Parallel Linear Algebra for Scalable Multi-core Architectures*.
[99] **QUARK :** *QUeueing And Runtime for Kernels*.
[100] **DPLASMA :** *Distributed Parallel Linear Algebra Software for Multi-core Architectures*.
[101] **BS2 :** *Bulk Synchronous Block Scheduling*.

blocs à décompositions hiérarchiques afin de mieux adapter la charge de travail entre des unités de calcul hétérogènes. Ils adoptent une approche similaire à celle présentée plus loin dans ce manuscrit. Des barrières de synchronisation sont utilisées au lieu de reposer sur un DAG de tâches. L'ordonnanceur est ainsi organisé sur deux étages dont le dernier niveau repose sur le modèle de tâches imbriquées offert par OpenMP 3.0. Ce dernier est sollicité pour d'équilibrer la charge entre les cœurs de calcul des processeurs généralistes. L'emploi des deux ordonnanceurs peuvent entraîner des surcoûts associés à leur cohabitation. La cohérence entre les mémoires disjointes est imposée de manière forte (*write through*) et ne permet malheureusement pas une rétention efficace des données. Deisher et al. ont quant à eux élaboré un ordonnanceur dédié à la factorisation LU dense [208] pour exploiter un accélérateur *Intel Xeon Phi* et des cœurs de processeurs généralistes. *MAGMA*[102] [209] étend les fonctionnalités de PLASMA [210] pour supporter des machines hétérogènes. Les premières versions étaient basées sur une répartition statique des opérations entre les ressources de différentes natures. A partir de la version 1.2, la bibliothèque repose sur l'ordonnanceur de tâches hétérogènes *StarPU* en permet désormais un équilibrage de charge dynamique [211]. *MAGMA* exploite des accélérateurs NVIDIA avec CUDA, *clMAGMA* [212] prend en charge d'autres accélérateurs avec le support d'OpenCL. Enfin, *MAGMA MIC* est complètement dédiée à l'utilisation des coprocesseurs *Intel Xeon Phi*.

En résumé, à partir d'un algorithme similaire, il existe plusieurs approches pour organiser le calcul d'une factorisation LU. Dans le cadre d'une accélération hétérogène, des méthodes statiques nécessitent de bien calibrer la répartition des opérations entre toutes les unités de calcul. Des ordonnanceurs de tâches hétérogènes permettent un équilibrage automatique au cours de l'exécution et de contourner la limitation imposée par un précalibrage performant. La partie suivante démontre que cette fonctionnalité est davantage intéressante pour un code irrégulier.

3.4.2 Factorisation LU de matrices creuses et adaptations

Lorsqu'une matrice contient beaucoup de valeurs nulles, il peut être intéressant d'employer une représentation de matrice creuse par blocs. Le premier avantage est d'éviter de stocker la plupart des zéros en évitant d'allouer les blocs creux en mémoire. Un autre avantage est que certaines opérations impliquant des blocs creux peuvent être évitées. La difficulté de l'algorithme de factorisation LU de matrices creuses concerne l'évolution de la matrice au cours de l'algorithme : des valeurs peuvent apparaître dans des blocs creux qui sont alors contraints de se transformer en blocs denses. La **figure 3.6a** illustre cette variation de quantité de blocs creux. De la première à la deuxième itération, des blocs creux se sont densifiés, impliquant l'apparition d'une plus grande quantité d'opérations. Ainsi, la charge de travail peut changer à tout moment. Des solveurs creux ont émergé dans les dix dernières années pour les machines à mémoire partagée, à mémoire distribuée ou l'union des deux (MUMPS[103] [213] , PaStiX[104] [214], SuperLU[105] [215]).

Outre des bibliothèques commerciales comme *CULA* [216], peu de solutions permettent l'exploitation d'accélérateurs matériels dans ce contexte. Un déport de calculs sur une quantité réduite de données aboutit généralement à des performances peu convaincantes. Il s'agit de concevoir un algorithme adapté qui contourne cette contrainte. Les données doivent être partiellement densifiée pour que les accélérateurs puissent délivrer des performance suffisamment intéressantes. L'ajustement choisi consiste à stocker la matrice par *super-blocs*, renfermant un certain nombre de blocs stockés de manière contiguë en mémoire. Ces blocs sont adaptés tout au long de l'exécution et une phase de calibration permet d'anticiper l'espace final occupé par chaque

[102] **MAGMA :** *Matrix Algebra for GPU and Multicore Architecture*.

[103] **MUMPS :** *a MUltifrontal Massively Parallel sparse direct Solver*.

[104] **PaStiX :** *Parallel Sparse matriX package*.

[105] **SuperLU :** *Supernodal LU*.

super-bloc. Ainsi ces blocs peuvent être transférés plus aisément vers les mémoires déportées et leur regroupement sert à rassembler plus d'opérations sur une grande quantité de données. La **figure 3.6b** illustre cette gestion par super-blocs.

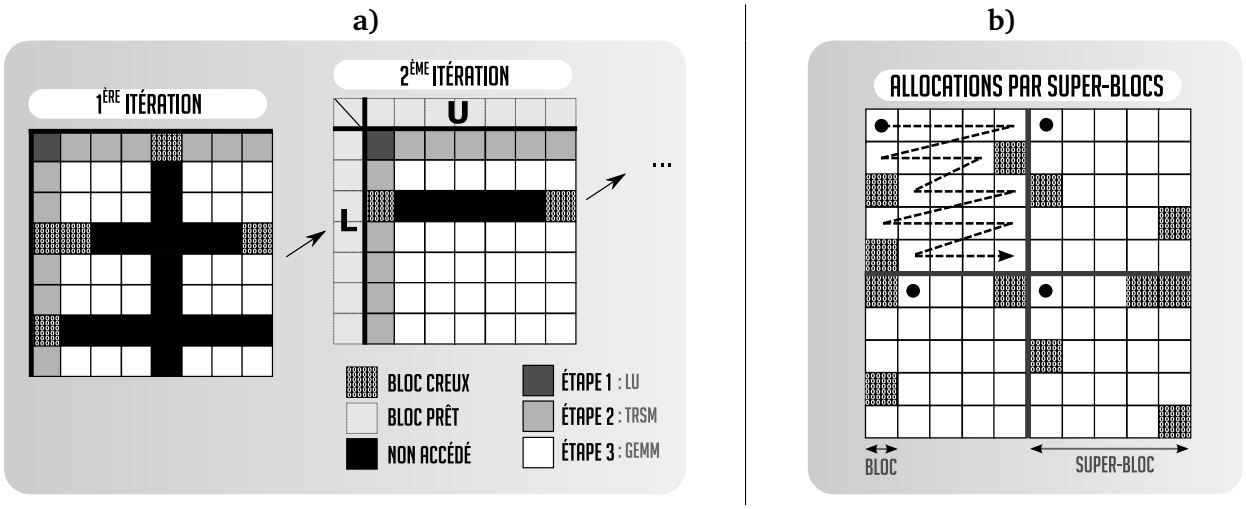


FIGURE 3.6 – **a)** Données modifiées par chaque étape au cours de la première et deuxième itération d'un algorithme LU creux par blocs dans la version *right-looking*. **b)** Allocation par lot de blocs (super-blocs). Chaque super-bloc contient 5×5 blocs creux ou denses.

L'algorithme de factorisation LU dense présenté dans la section précédente est ensuite appliqué à de telles matrices, en prenant soin d'exclure certaines opérations impactant des blocs creux. L'étude du temps d'exécution de cet algorithme irrégulier permet d'évaluer les performances d'adaptation d'une plate-forme d'ordonnancement. Des tâches contenant des quantités variables d'opérations sont générées, ce qui justifie un équilibrage dynamique entre toutes les unités de calcul hétérogènes.

Discussions et motivations

Équilibrage de charge hétérogène à granularité variable

Des modèles de programmation encouragent à utiliser un parallélisme à grain fin pour améliorer l'équilibrage de charge. Cette stratégie n'est pas forcément adaptée aux accélérateurs déportés qui nécessitent d'une grande quantité de calcul pour, à la fois exploiter au mieux l'architecture, et déclencher des déplacements de données efficaces. CellSs agrège à la volée des tâches afin d'amoindrir les surcoûts induits par l'initiation de ces transferts. Une autre approche consiste à ajuster dynamiquement la granularité des tâches en fonction de l'architecture et de réduire la compétition entre des ressources de calcul possédant des capacités de traitement nettement différentes. XKaapi et OMPSs permettent cette pratique en ayant recours à des tâches imbriqués, sans pour autant offrir un second niveau d'équilibrage séparé et plus adapté aux unités hétérogènes. OMPSs n'offre d'ailleurs pas d'équilibrage de charge hétérogène puisque l'architecture ciblée est directement spécifiée lors de la déclaration de la tâche. Cela se traduit par une flexibilité amoindrie, causée par une plus faible collaboration entre les unités de calcul hétérogènes. Lorsque des tâches intensives sont déportées sur des accélérateurs, les processeurs généralistes peuvent être ainsi sous exploités.

Meilleure rétention des données dans les caches logiciels

Les performances globales des opérations déportées sur des accélérateurs sont souvent conditionnées par les transferts de données. La majorité des plate-formes d'exécution hétérogènes ont recours à des caches logiciels afin d'accentuer la localité temporelle des mémoires déportées et réduire la quantité globale de données véhiculées. StarPU, XKaapi et OMPSs font appels à de mécanismes plus ou moins poussés pour aiguiller l'attribution des tâches en fonction des données présentes dans les caches logiciels, et ainsi amplifier le recyclage de données sans déclencher de communication. La probabilité de présence des données repose alors en grande partie sur la politique d'éviction. La politique LRU, adoptée par la plupart des plate-formes, n'est pas toujours la plus adaptée car ses performances dépendent fortement de la capacité du cache logiciel et ne permet pas de capturer pleinement les opportunités de rétention tout au long de la durée de vie du programme. La sémantique associée à la déclaration des tâches peut être exploitée pour récupérer des informations visant à améliorer la politique d'éviction et mieux anticiper les potentiels de réutilisation. Un couplage plus fort avec l'ordonnanceur peut y être adjoint en favorisant des attributions de tâches qui accroissent d'avantage cette rétention.

Exploitation de la hiérarchie mémoire

Accroître la localité spatiale des données est également un facteur clé des performances. Bien souvent, les plate-formes hétérogènes ne possèdent pas de mécanismes adaptés à l'imbrication de mémoires, notamment en termes d'accès non uniformes à la mémoire centrale. Peu de travaux s'attachent à résoudre des contraintes à la fois d'ordre topologique et liées à la gestion de ressources hétérogènes. Associer des tâches à granularités multiples à une gestion hiérarchique peut contribuer à accroître la localité spatiale à la fois des nœuds NUMA et de leur affinités aux mémoires déportées. StarPU possède une représentation "à plat" des unités de traitement. OmpSs emploie un vol de tâche guidé par affinité qui ne permet pas d'organiser une collaboration des cœurs de calcul d'un même processeur afin de mieux exploiter une mémoire locale partagée. Des principes d'équilibrage de charge hiérarchiques combinés à une répartition initiale des données dans les différentes mémoires sont étudiés dans ce manuscrit.

Rétention de données entre appels à une bibliothèque

De plus en plus de bibliothèques de calculs hétérogènes reposent sur des plate-formes à équilibrage dynamique qui ont recours à des caches logiciels. Ces bibliothèques s'appuient généralement sur des noyaux optimisés pour offrir des performances de traitement accrues. Or, à l'issu d'un appel à une telle bibliothèque, les données stockées en cache logiciel sont évacuées pour assurer leur cohérence en mémoire centrale. Si un code de calcul à recours à plusieurs traitements ou bien du même appel de bibliothèque plusieurs fois de suite, une série de transferts inutiles peut en résulter. Ces mouvements engendrent des pertes d'efficacité non négligeables lors de phases de calcul intensif. Il peut être judicieux de recourir à une interface unifiée, laquelle autoriserait une rétention des données entre plusieurs appels d'une bibliothèque hétérogène et des tâches de calcul définies à la main.

En résumé, la parallélisation des codes de calcul est indispensable pour atteindre de hautes performances. Les portions conservées séquentielles peuvent fortement impacter les performances globales et doivent ainsi être limitées. La difficulté avec les nœuds de calcul hétérogènes est de permettre un équilibrage entre différentes architectures dont les mémoires sont potentiellement dissociées. Des modèles de programmation dynamiques permettent d'épauler le programmeur en se répartissant les calculs au cours de l'exécution. Bien souvent, ils se basent sur une agrégation de modèles statiques. La majorité des solutions offrant un tel équilibrage de charge font appel au parallélisme de tâches, permettant de récupérer, via une sémantique particulière, les informations suffisantes à la fois pour déclencher des transferts de données et faciliter la répartition des calculs. Le reste de ce manuscrit décrit les contributions proposées pour améliorer les performances d'une plate-forme d'exécution s'appuyant sur ce genre de parallélisme, en se focalisant sur l'accès aux données dans différentes mémoires. Il s'agit d'élaborer des mécanismes logiciels ajustés aux contraintes et spécifications matérielles tout en masquant une partie de la difficulté de programmation des architectures actuelles aux développeurs.

Deuxième partie

Contributions

Chapitre 4

Localités spatiales au sein des tâches hétérogènes de calcul

> *“ Supprimer la distance, c’est augmenter la durée du temps. Désormais, on ne vivra pas plus longtemps ; seulement, on vivra plus vite. ”*
>
> Alexandre Dumas, *Mes Mémoires*, 1863

Les chapitres précédents ont mis en lumière les contraintes matérielles associées à l’accès aux données et l’intérêt d’exploiter l’organisation hiérarchique de la mémoire. Les principes de localités doivent être ainsi respectés, pour espérer tirer parti de hautes performances que peuvent théoriquement fournir les nœuds de calcul. Dans un programme reposant sur du parallélisme de tâches et lorsque son algorithme le permet, la décomposition du travail et l’accès aux données peuvent être étudiés afin de favoriser les accès locaux. Au sein d’un nœud de calcul hétérogène, deux formes de localités spatiales peuvent être dénombrées :

— La première forme de localité est associée aux accès à la mémoire centrale. Il convient de porter un intérêt particulier au placement des données dans les pages mémoires pour réduire les effets non désirés des accès non uniformes (NUMA et NUIOA). Le placement et la répartition du travail à l’intérieur de ces pages peuvent également favoriser l’exploitation de mémoires locales partagées.

— La seconde forme se focalise sur la mise à disposition des données pour les accélérateurs matériels qui embarquent leur propre mémoire. Puisque ces mémoires sont gérées de manière séparée et que le placement des données n’est pas automatiquement assuré par le matériel, le rapatriement des données nécessite une prise en charge par le logiciel. Les modifications des données déportées doivent être tracées et répercutées en mémoire centrale pour maintenir un état cohérent de celles-ci. L’espace mémoire embarqué est relativement limité (quelques Go), ce qui rend cette manipulation encore plus complexe. D’autres contraintes liées aux accès DMA et aux accès par les unités de calculs peuvent être considérées.

Ces contraintes d’accès aux données impactent directement la répartition des calculs, c’est-à-dire le choix de l’affectation du travail à effectuer sur les ressources disponibles. Des algorithmes d’équilibrage en ligne comme le vol de travail présentent l’avantage de fonctionner en l’absence d’informations sur la durée d’exécution des tâches. Par rapport à un ordonnancement purement

statique, les algorithmes en ligne peuvent aussi ajuster dynamiquement la charge de travail après l'introduction d'un déséquilibre induit par divers mécanismes matériels qui ne peuvent pas toujours être anticipés. C'est notamment le cas des interruptions matérielles, des phénomènes de congestion des bus ou encore d'accès la mémoire par un périphérique de communication. Dans le cadre d'une machine à ressources de calcul hétérogène, l'équilibrage de charge doit également s'opérer en tenant compte des différences de performances de calcul, lesquelles dépendent des opérations effectuées. Ce déséquilibre peut être relativement important et est donc un problème substantiel lorsque des processeurs d'architectures différentes doivent travailler de concert.

Ce chapitre présente une approche permettant de concilier ces différents aspects de localité et d'ordonnancement hétérogène, lorsque le programme cible le permet. Les premières contributions de cette thèse y sont introduites à savoir des tâches hétérogènes à granularités variables, un ordonnancement hiérarchique et un placement spatial contrôlé des données. La première partie (4.1) démontre l'intérêt de recourir à des tâches à multi-granularité. La partie suivante (4.2) présente comment les mécanismes d'équilibrage de charge sont bâtis autour de ces tâches particulières dans une machine à mémoires hiérarchiques et disjointes. Enfin, la dernière partie (4.3) aborde la répartition des données pour favoriser un comportement adéquat avec les mécanismes adoptés. L'efficacité de ces contributions est évaluée sur des micro-programmes de calcul matriciel. Les opérations d'algèbre linéaire interviennent fréquemment dans certains codes de simulation en tant que phases intensives de calcul. Accentuer la localité spatiale permet d'accélérer ces traitements. En particulier, une factorisation LU de matrices creuses permet de mesurer la pertinence de la multi-granularité dans un contexte de code irrégulier. Des multiplications de matrices denses sont également employées pour apprécier les améliorations apportées par la combinaison des trois contributions de ce chapitre.

4.1 Tâches de calcul hétérogènes à granularités multiples

Une tâche hétérogène embarque des données utiles à l'ordonnancement, le contexte de la tâche en sauvegardant les valeurs des données à la création, ainsi que deux pointeurs de fonctions permettant un mode d'exécution adapté à chacune des cibles. Le choix des unités d'exécution peut être effectué de manière tardive. Cela peut avoir des répercussions sur les performances d'accès aux données, et un impact sur l'équilibrage de la charge en affectant une trop grosse quantité de calcul à une unité relativement lente. Adapter la quantité de travail à la volée permet plus de souplesse pour répartir les opérations entre des ressources de calcul hétérogènes.

Typiquement, une tâche de calcul est dimensionnée pour une architecture spécifique. Lorsqu'il est également prévu qu'elle soit exécutée par un accélérateur massivement parallèle, une grosse granularité est choisie afin d'occuper les centaines, voire les milliers de cœurs de calcul embarqués. La fonction associée à l'exécution sur cet accélérateur est représentée à partir d'un modèle de programmation tels que CUDA ou OpenCL (cf. **chapitre 3.2.4**, page 62). Les opérations de la tâche sont décrites à partir du langage adapté afin d'occuper chacun des cœurs légers embarqués. Il est fréquent de vouloir répartir les opérations en un nombre égal à plusieurs fois le nombre de cœurs légers disponibles. Plusieurs flux de calcul par unité d'exécution permettent de recouvrir la récupération des données en mémoire embarquée par du calcul. La granularité d'exécution sur accélérateur, décrite par le langage spécifique, est donc très fine.

Les modèles de programmation parallèles pour les processeurs multi-cœurs reposent quant à eux sur des processus ou des processus légers (cf. **chapitre 3.2.1**, page 60). Par exemple, un processus léger peut être associé à un cœur de calcul et diriger les opérations. C'est notamment le cas d'OpenMP qui, grâce à des directives, permet de transformer le code à la compilation et de répartir les calculs en ligne avec une plate-forme d'exécution basée sur des processus légers. Or, un ordonnanceur de tâches hétérogènes a aussi recours à des processus légers pour répartir les calculs sur les processeurs multi-cœurs. Faire appel à un autre modèle de programmation parallèle pour répartir des opérations d'une tache sur plusieurs cœurs de calcul est un problème compliqué. Il s'agit dans ce cas de faire cohabiter plusieurs plate-formes d'exécution, chacune basée sur des processus légers. Un arbitrage doit donc être opéré et peut entraîner un surcoût lié à cette gestion. De plus, puisque l'exécution est déléguée à une autre plate-forme, l'ordonnanceur principal n'a plus la main sur la façon dont les opérations sont réparties. Une autre solution serait de supporter certains modèles de programmation et de les faire reposer sur la même plate-forme, gardant ainsi un contrôle global sur l'exécution des opérations. Cette solution est exploitée et présentée dans le **chapitre 6** (page 113). Une autre approche proposée en complément dans cette thèse est de décrire la décomposition en tâches de calcul plus fines au sein des fonctions dédiées à une exécution sur les processeurs généralistes et d'y associer un ordonnancement spécifique.

Cette section met en évidence la disparité des performances de calcul qui peut exister entre des architectures différentes. La notion de *super-tâche* de calcul est ensuite introduite et servira de pièce maîtresse à l'ordonnancement. En effet, tous les mécanismes décrits dans les sections et chapitres suivants viendront se greffer sur cette brique de base et conditionner l'organisation des calculs.

4.1.1 Granularité et performances disparates

Plusieurs raisons peuvent expliquer les différences de comportement en matière de performances de calcul. La première peut être associée à la quantité de travail nécessaire. Des accélérateurs massivement parallèles requièrent une grosse granularité de calcul. Ensuite, selon la quantité de travail, les performances atteintes dépendent fortement du matériel, tels que le nombre de registres disponibles, la quantité et la hiérarchie de mémoires locales, la bande passante atteignable, le nombre de bancs d'accès à la mémoire, etc [217] (cf. **chapitre 2**, page 33). Les performances dépendent également des techniques d'optimisations employées pour coder les noyaux de calcul. Les **figures 4.1a et 4.1b** révèlent les différents comportements de processeurs généralistes face à des processeurs graphiques sur un enchaînement de 25 multiplications de matrices à partir de bibliothèques de calcul optimisées pour chacune de ces architectures. Le noyau *SGEMM* issu de la bibliothèque de calcul *Intel MKL* (cf. **figure 4.1a**) atteint globalement une asymptote pour des tailles de matrices supérieures à 192×192. Au-delà, un cœur *Intel Xeon E5520* permet de fournir un maximum de 16 à 17 GFLOP/s contre environ 9 à 11 GFLOP/s pour un cœur *AMD 6164HE*. Sur processeurs graphiques les performances de la bibliothèque *Nvidia cuBLAS* [218] sont bien plus erratiques avec des variations allant de 510 à 320 GFLOP/s avec une carte accélératrice *Geforce GTX 480* (cf. **figure 4.1b**).

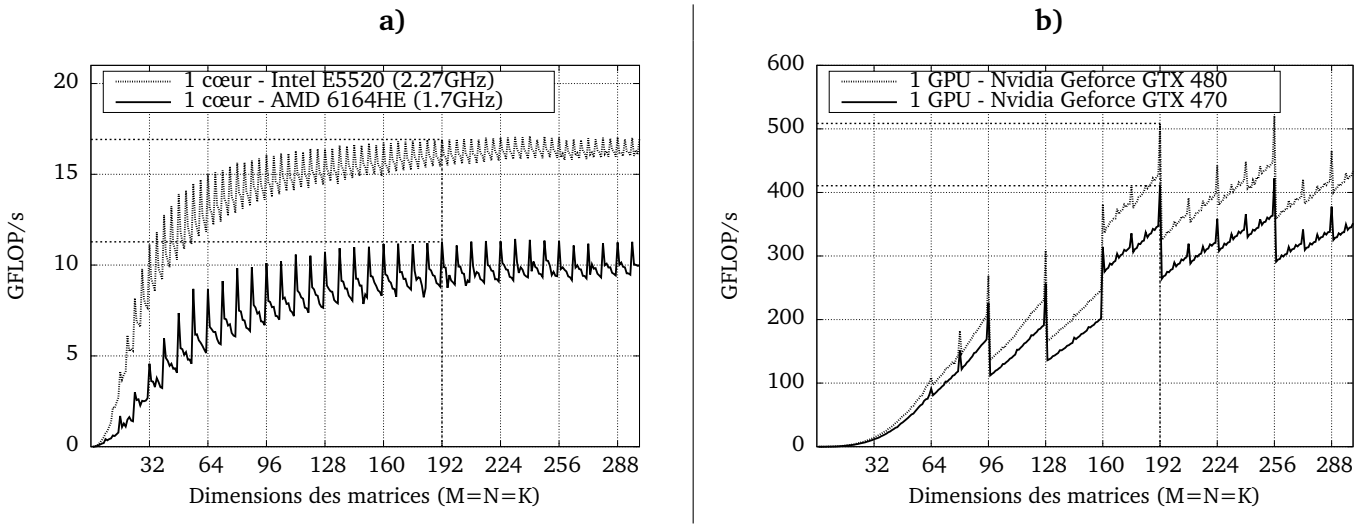


FIGURE 4.1 – Hétérogénéités des performances de 25 multiplications en fonction de la taille des matrices. Version simple précision (SGEMM) avec des matrices carrées **a)** un cœur de processeur avec la bibliothèque *Intel MKL* **b)** performance de processeurs graphiques avec *NVIDIA cuBLAS*, hors transferts de données.

Les cœurs de calcul des processeurs graphique requièrent également des accès groupés aux données pour atteindre de bandes passantes maximales (cf. **chapitre 2.4.3**, page 51). De plus, grouper spatialement les données accédées dans une tâche de calcul permet de déclencher moins de transferts et de réduire le surcoût d'initiation des copies par les contrôleurs DMA (cf. **chapitre 2.4.1**, page 48).

Enfin, la différence de performance est variable entre les unités de calcul hétérogènes en fonction des opérations effectuées et du degré d'optimisations du code. Il peut être alors intéressant d'aiguiller l'exécution de certaines tâches vers une architecture plutôt qu'une autre afin de maximiser les performances globales. Il s'agit d'attribuer à chaque tâche une affinité pour un lot d'unités de calcul pour lesquelles les opérations sont plus efficaces. Par exemple, un mode d'ordonnancement de *StarPU* permet de mesurer les temps d'exécution des tâches hétérogènes sur chaque architecture. L'affectation de la tâche est choisie en ligne en fonction de divers paramètres

afin de minimiser la durée d'exécution. L'ordonnancement se base alors sur des informations recueillies soit au cours de l'exécution, soit à partir d'exécutions antérieures. Néanmoins, plus l'ordonnanceur intègre des informations dans sa prise de décision, plus le surcoût dû au calcul de l'attribution peut devenir important et finir par affecter les performances. Il se peut aussi que la quantité de travail varie dans les tâches du même type. Dans ce cas particulier, l'équilibrage de charge basé sur la mesure de temps de calcul ne permet plus d'ordonnancer précisément les opérations. Une solution est de faire appel à un mode d'équilibrage reposant sur moins d'informations obtenues *a priori*.

4.1.2 Tâches hétérogènes décomposables

Les tâches décomposables permettent d'adapter la quantité de travail en fonction de l'architecture d'exécution. Ainsi, une grosse quantité de calculs peut être fractionnée en plusieurs tâches à granularité plus faible pour mieux équilibrer la charge de travail entre des unités d'exécution plus lentes. C'est précisément le rôle d'une *super-tâche* (cf. figure 4.2), laquelle est également associée à d'autres caractéristiques. Elle doit tout d'abord embarquer une quantité d'opérations relativement conséquente pour permettre cette décomposition. Des tâches légères ou des tâches lourdes sont ainsi générées en fonction de la quantité de travail souhaitée.

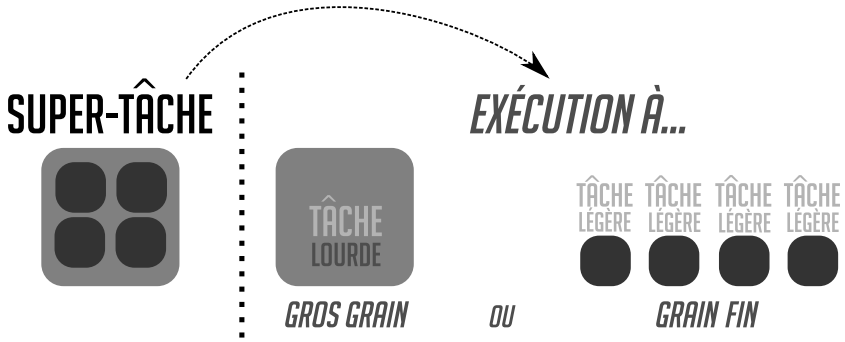


FIGURE 4.2 – Super-tâche et choix de la granularité à l'exécution

Ensuite, les données qui y sont associées doivent, autant que faire se peut, permettre des accès respectant les principes de localité. Dans l'idéal, les données modifiées ou produites doivent être localisées sur des pages mémoires d'un unique nœud NUMA. Regrouper spatialement les données accédées pour une super-tâche, réduit d'une part le nombre de transferts nécessaires lorsque les données doivent être déportées, et favorise la localité des calculs lorsqu'ils sont affectés à un lot de cœurs. Cela laisse notamment des opportunités pour exploiter, de manière conjointe, une mémoire locale partagée.

La figure 4.3 présente la différence d'équilibrage de charge en mode hétérogène sur des multiplications de matrices de diverses dimensions. Dans tous les cas, les calculs reposent sur les bibliothèques optimisées *Nvidia cuBLAS* et *Intel MKL*. Des blocs de 1024×1024 sont utilisés pour paralléliser en tâches de calcul les opérations. Pour l'ordonnancement à multi-granularité, les blocs sont conservés tels quels dans les tâches lourdes et sont décomposées en sous-blocs de 256×256 pour les tâches légères. L'ordonnancement à simple granularité ne considère quant à lui que les blocs de 1024×1024 en faisant appel au noyau *SGEMM* de la bibliothèque pour architectures hétérogènes MAGMA [219, 210, 209, 211]. Cette bibliothèque exploite l'ordonnanceur StarPU et l'expression des tâches de calcul est prise en charge par l'interface MORSE [220]. Pour des matrices de faibles dimensions, les performances sont meilleures avec MAGMA car les tâches

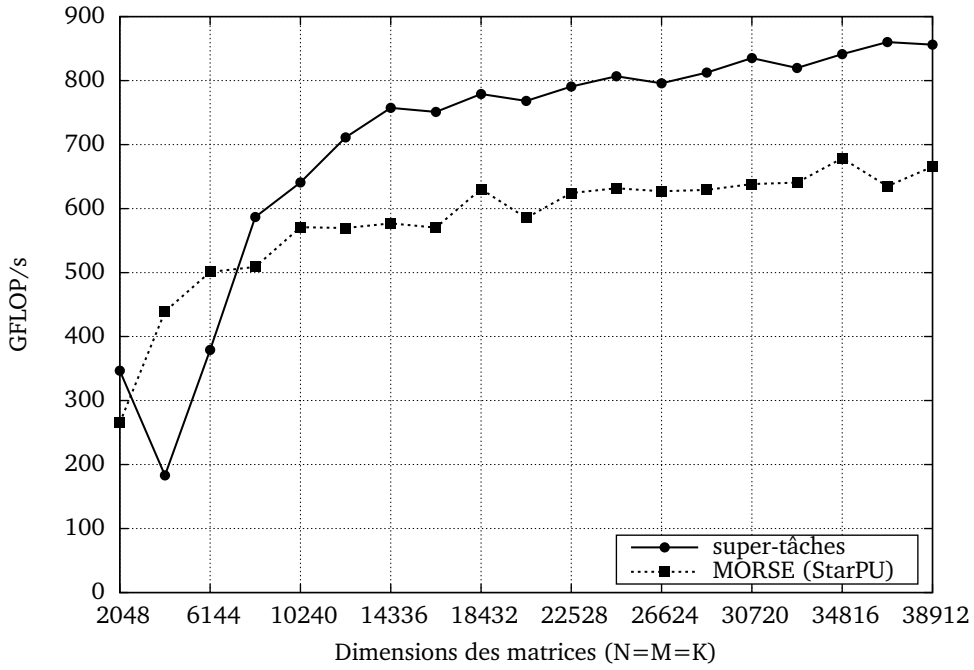


FIGURE 4.3 – *SGEMM* hétérogènes avec deux processeurs Intel Xeon et deux accélérateurs Nvidia M2050 (*Inti hétérogène*, cf. **annexe A.1**), décomposition en blocs de 1024×1024 valeurs. Les temps de transfert des données sont compris. Comparaison entre MAGMA 1.3 (MORSE/StarPU) (ordonnancement de tâches à simple granularité) et un ordonnancement basé sur des super-tâches.

sont exclusivement exécutées par les accélérateurs pour ces dimensions. Une condition sur la récupération des super-tâches pourrait lisser ces différences comme le recours à un seuil sur le nombre de super-tâches présentes dans la liste à partir duquel une super-tâche peut être récupérée pour être décomposée. Pour des matrices de dimensions supérieures à 6144×6144, les super-tâches permettent en général de mieux répartir les opérations avec moins d'informations *a priori*.

Granularité de la décomposition

Le choix de la granularité des tâches et super-tâches de calcul est un problème difficile que cette thèse ne tente pas de résoudre. C'est souvent le fruit d'un compromis entre l'équilibrage de charge, répartition des données et performances brutes des sous-opérations. La granularité des sous-tâches doit, par exemple, éviter d'aboutir à des situations de faux partage. Plusieurs autres paramètres entrent en jeu, tels que les contraintes matérielles des architectures exploitées, la taille de problème traité ou encore le type d'opérations effectuées. La quantité de données d'une super-tâche a aussi un impact direct sur la quantité de données transférées vers les mémoires des accélérateurs déportés puisque la cohérence doit être assurée.

Conserver la possibilité d'adapter ces paramètres de granularité permet plus de flexibilité. Cela laisse aussi des opportunités pour assurer la pérennité du code et mieux s'accorder avec les architectures à venir. Une autre solution est de déléguer le choix d'une granularité de décomposition d'une super-tâche à une bibliothèque optimisée. La bibliothèque *PLASMA* utilise l'ordonnanceur spécialisé *QUARK* [221] et embarque une granularité adéquate pour l'exécution de noyaux d'algèbre linaire sur les cœurs de calcul. Cette information est notamment exploitée par la bibliothèque *MAGMA* pour permettre une décomposition permettant un ordonnancement

hétérogène. Des méthodes d'apprentissage[106] ou d'autocalibration permettent de trouver automatiquement des paramètres et d'aboutir à de meilleures performances. Ces méthodes sont par exemple utilisées dans certaines bibliothèques portables et optimisées (*ATLAS* [222, 223], *PHiPAC* [224, 225], *FFTW* [226, 227], *Sparsity* [228, 229], etc.) et pourraient être employées pour déterminer automatiquement les facteurs des décompositions, même si le problème demeure complexe lorsque plusieurs accélérateurs sont exploités.

4.1.3 Évaluation de la granularité variable à la demande sur LU creux par blocs

Ordonnancement et choix des paramètres expérimentaux

Les matrices évaluées sont constituées de N^2 blocs de taille 192×192. Le choix de la taille des blocs est déterminé expérimentalement et appuyé par les résultats présentés dans la section précédente (cf. **figures 4.1a et 4.1b**). Il s'agit de trouver un compromis pour la taille des blocs. Si les blocs sont trop gros, il y a moins d'opportunités pour représenter l'aspect creux de la matrice puisqu'une valeur non nulle contraint un bloc à être dense. Des blocs de petites tailles permettent *a priori* un meilleur équilibrage de charge, mais en contrepartie, une granularité trop fine peut aboutir a ne pas tirer correctement parti d'un accélérateur matériel. Le nombre de blocs varie entre 40 et 160 par dimension pour un total de $1\,600$ à $25\,600$ par matrices. Un tiers de ceux situés en dehors de la diagonale sont creux. La position de chaque bloc creux est déterminée en avance pour une taille de matrice donnée pour que les performances entre deux exécutions à partir de la même taille de matrice puissent être comparées. La matrice est allouée par lot de 5×5 blocs qui peuvent être creux ou denses.

Un ordonnancement ne tenant pas compte de la hiérarchie mémoire est choisi dans cette évaluation. Cette caractéristique sera appréciée dans une section suivante. Un producteur ajoute des tâches dans une unique liste partagée. Chaque ouvrier ou unité d'exécution peut récupérer une tâche avant de l'exécuter. Pour l'ordonnancement à multi-granularité, les tâches partagées sont des super-tâches (cf. **figure 4.4**). Lorsque l'une d'entre elles est récupérée par une unité associée aux processeurs multi-cœurs, elle est décomposée et chaque sous-tâche est distribuée à une unité du même type. Ces unités peuvent également se voler mutuellement des tâches et permettre un second niveau d'équilibrage de charge entre unités du même type.

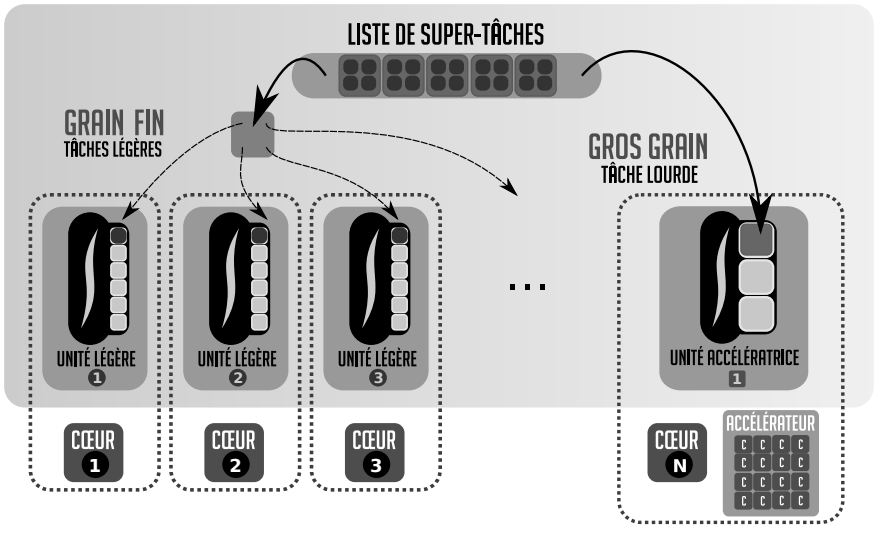


FIGURE 4.4 – Ordonnancement dynamique à liste de super-tâches centralisée pour unités de calcul hétérogènes.

[106] *Machine Learning*.

Performances

La figure 4.5a révèle les performances de l'étape 3 de l'algorithme LU creux pour des exécutions homogènes sur la machine *ENS-fermi* (cf. annexe A.6). Ces résultats démontrent qu'une faible granularité est préférable pour des cœurs de calcul généralistes puisque cela permet un meilleur équilibrage de charge pour des matrices de petites et moyennes dimensions. À l'inverse, une grosse granularité est préférable pour un processeur graphique où les transferts de données sont plus efficaces. Il est important de noter que les performances très moyennes de l'accélérateur sont dues aux transferts systématiques, problème dont les effets seront amoindris grâce aux contributions introduites dans le chapitre suivant.

La figure 4.5b présente les performances d'un ordonnancent hétérogène en ayant recours à des tâches à simple ou multi-granularité. Les accélérateurs et cœurs de processeurs généralistes ne sont pas exploités de la même manière. Les premiers nécessitent généralement d'un processus léger associé à un cœur généraliste qui est dédié à son pilotage, tandis que les autres cœurs généralistes possèdent chacun leur processus léger. Un ordonnancement sans surdécomposition peut induire une compétition entre les cœurs généralistes et un accélérateur dont les performances crêtes sont théoriquement bien supérieures. Cette différence est habituellement compensée par des modèles de coût évoqués précédemment. La multi-granularité présente ainsi deux intérêts en combinant des granularités plus adaptées à chaque architecture et en maintenant un équilibrage de charge possible entre les unités de calcul hétérogènes sans recourir à des modèles de coût.

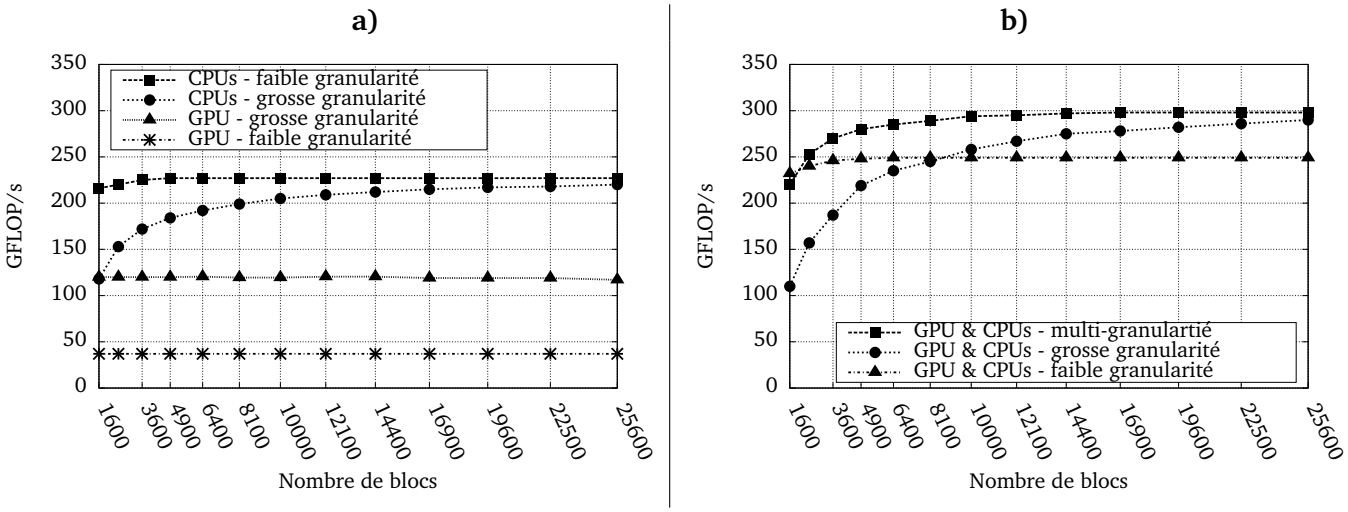


FIGURE 4.5 – Performances de l'étape 3 de la factorisations LU de matrices creuses en simple précision **a)** Ordonnancement homogène à faible et grosse granularités. **b)** Ordonnancement hétérogène avec différentes granularités.

4.2 Affinités hiérarchiques des super-tâches

Le concept de mutigranularité a été introduit dans la section précédente avec les super-tâches. Ces dernières peuvent être exécutées par des unités de calcul qui peuvent être issues d'architectures différentes. Certaines de ces unités possèdent une affinité à un *nœud NUMA* particulier (cf. chapitre 2.3.1, page 42). Employer une unique liste de super-tâches partagée par toutes les unités d'un nœud de calcul n'est pas adapté à une exécution favorisant la localité spatiale. À l'image du bus dans les architectures *SMP UMA* (cf. chapitre 2.3, page 41), l'utilisation d'une telle liste introduit un point de contention, puisque toutes les unités de calcul y accéderont de manière concurrente. De plus, ce type d'accès ne permet pas de guider aisément les super-tâches vers les unités appropriées en tenant compte de la localité des données. Cette section se focalise sur l'organisation de l'exécution des super-tâches au sein d'un nœud de calcul. Des contraintes hiérarchiques y sont décrites et permettent d'aiguiller l'exécution des super-tâches afin de favoriser les accès locaux à la mémoire et permettre un meilleur travail collaboratif. Cette section s'attache dans un premier temps à décrire les contraintes de ces mécanismes en construisant une représentation abstraite. Puis, quelques exemples d'abstractions sont présentés en fonction de la machine cible choisie.

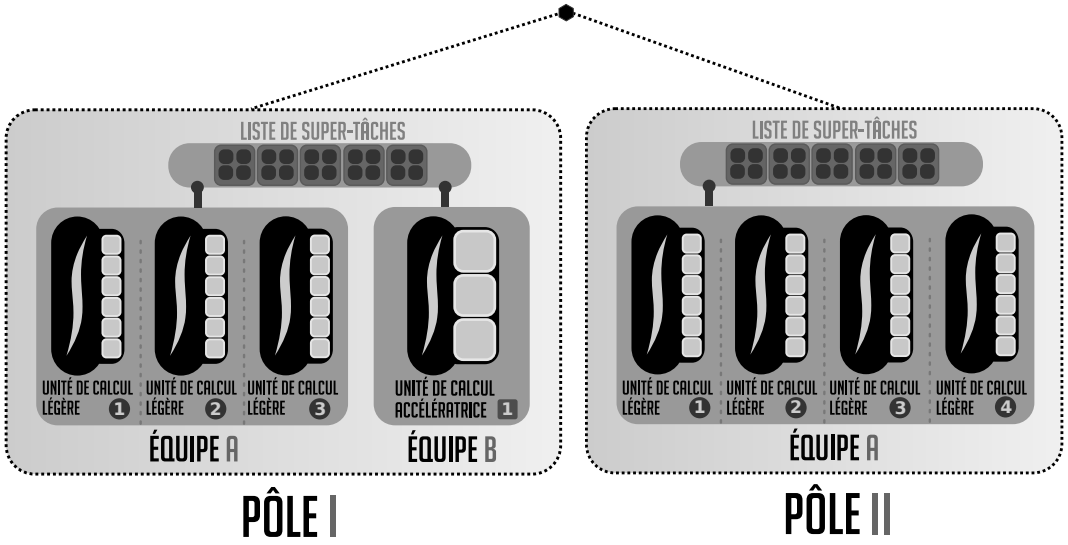


FIGURE 4.6 – Organisation abstraite. Un exemple composé de deux pôles, chacun constitué de quatre unités logiques.

4.2.1 Organisation abstraite

Cette section présente l'organisation et les accès aux super-taches en tenant compte des affinités de calcul. Cette organisation repose sur une abstraction de la topologie de la machine pour permettre plus de souplesse et termes d'affinités et d'adaptabilité à des futures configurations. Chaque nœud de calcul est ainsi composé d'*unités logiques*, regroupées en une hiérarchie d'*équipes* et de *pôles* de travail. La figure 4.6 dévoile une de ces représentations abstraites. Chaque unité logique est pilotée par cœur de processeur généraliste. Les flots d'exécutions des opérations sont supportés par des processus légers qui sont schématisés par un fil en forme de "*S*". L'empilement de carrés clairs représentent les listes locales de tâches attribuées à une unité logique. La taille des carrés symbolise la granularité globalement associée à l'unité logique. Enfin, les accès privilégiés aux listes de super-tâches sont également matérialisés. Les caractéristiques des différents composants imbriqués sont décrites ci-après.

Unité de travail : détermine l'association à une granularité d'exécution

La représentation abstraite repose sur deux types d'unités logiques aux comportements distincts. Le premier type, appelé *unité de calcul légère*, est rattaché à un cœur de calcul de processeur généraliste et nécessite d'une étape de décomposition d'une super-tache avant d'en exécuter les opérations contenues. La deuxième unité logique est désignée par le terme d'*unité de calcul accélératrice*. Cette unité permet généralement l'exploitation d'un accélérateur. Dans la majorité des cas, un accélérateurs nécessite un cœur de processeur généraliste afin de le piloter, c'est à dire d'initier les noyaux de calcul et éventuellement de déclencher des transferts de données. Ainsi, le couple accélérateur et cœur de processeur généraliste forme une unité logique spécifique où l'exécution à gros grain d'une super-tâche est déportée sur l'accélérateur. Les unités sont regroupées dans des équipes de travail qui permettent notamment d'organiser les accès aux super-tâches.

Équipe de travail : impose des contraintes sur la cohérence des données

Chaque équipe de travail ne peut comporter que des unité logiques de même type. La granularité d'exécution y est donc uniforme. De plus, les unités d'une équipe doivent partager un accès direct à une même mémoire physique. Cette contrainte permet d'assurer un accès concurrent aux données sans se soucier de leur cohérence. Si une donnée est accessible par une unité logique de l'équipe, les autres unités peuvent également y accéder. Par exemple, deux accélérateurs déportés ne peuvent pas appartenir à la même équipe puisque qu'ils embarquent chacun leur propre mémoire. Des transferts explicites seraient nécessaires afin d'assurer la cohérence des données. Dans l'idéal, les unités de calcul peuvent être regroupées dans la même équipe si elles possèdent au moins une mémoire locale commune. Cette caractéristique permet d'accroître la localité des mécanismes d'équilibrage de charge qui seront détaillés dans le **chapitre 6** (page 113).

Pôle de travail : fixe les affinités aux mémoires

Plusieurs équipes de travail sont ensuite mises en relation dans un pôle de travail. Ce dernier est destiné à réduire les effets des accès non uniformes à la mémoire. Ainsi, un pôle regroupe les équipes possédant une plus forte affinité à la mémoire d'un nœud NUMA ou NUIOA. Chaque pôle contient au minimum une liste de super-tâche. Les équipes associées à des unités accédant directement à la mémoire centrale accèdent aléatoirement à toutes les listes de super-tâches. Les autres équipes, dont les unités qui la compose emploient une mémoire dédiée, possèdent chacune un accès privilégié à une unique liste de super-tâches. Cette propriété facilite la mise en place des mécanismes qui seront abordés dans le **chapitre 5** (page 97) et qui visent à accroître la localité temporelle dans les mémoires déportées.

Hiérarchie de pôles de travail : ordonne les mécanismes d'équilibrage de charge

Enfin, les différents pôles communiquent en tenant compte des facteurs NUMA. Ils sont connectés de manière hiérarchique afin de privilégier les accès les moins pénalisant. Cette hiérarchie est destinée à faciliter le déploiement d'un équilibrage de charge dynamique tenant compte de la distance des accès aux nœuds NUMA.

4.2.2 Représentation en fonction de la topologie de la machine

Cette section associe concrètement une représentation abstraite à différents nœuds de calcul en fonction des contraintes décrites précédemment. La découverte de la topologie de chaque nœud de calcul est confiée à la bibliothèque *HWLOC* [230, 231]. Cette dernière permet de récupérer des informations précises sur la hiérarchie d'un nœud de calcul. Cela comprend notamment l'imbrication des mémoires locales, la disposition des mémoires caches partagées, la configuration des nœuds *NUMA* et leurs facteurs de distance, la position des cartes d'extension. La bibliothèque *HWLOC* fait partie intégrante de l'implémention d'Open MPI et est également employée dans un nombre croissant de projets.

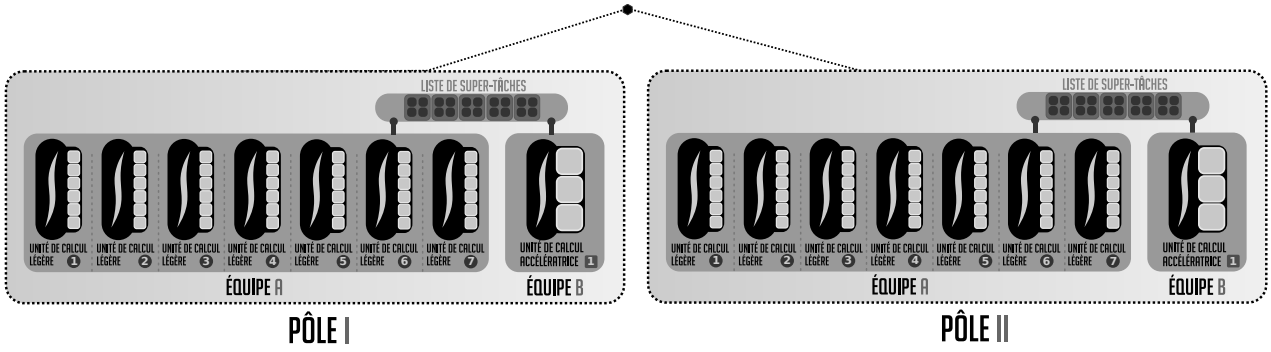


FIGURE 4.7 – Organisation abstraite pour un nœud de calcul *Cirrus hétérogène* (cf. **annexe A.5**), exemple composé de deux pôles symétriques.

Le nœud de calcul *Cirrus hétérogène* (cf. **annexe A.5**) embarque deux processeurs généralistes et deux accélérateurs qui sont connectés de manière symétrique. La machine est ainsi constituée de deux nœuds *NUIOA* équivalents. La **figure 4.7** présente cette répartition en deux pôles similaires. Chaque pôle comprend deux équipes hétérogènes pour un total de 8 unités logiques. Une seule liste de super-tâches est employée dans chaque pôle, permettant la mise en place de mécanismes d'équilibrage de charge hétérogène à multi-granularité dans chaque pôle.

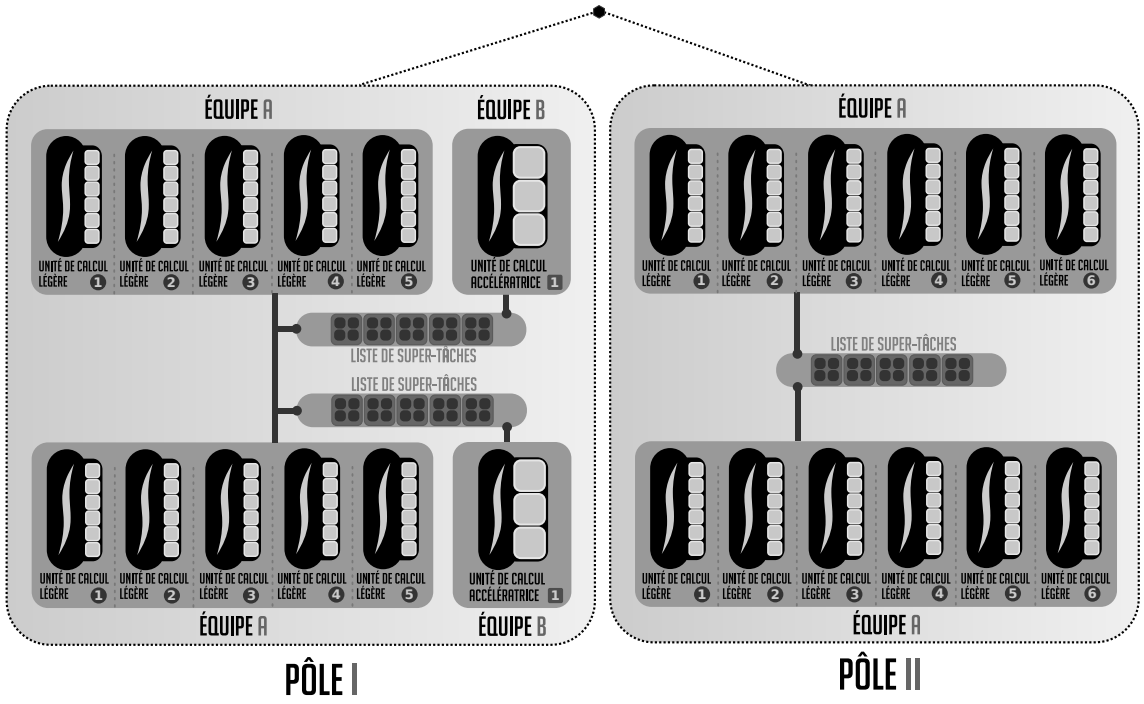


FIGURE 4.8 – Organisation abstraite pour un nœud de calcul asymétriques *ENS-fermi* (cf. **annexe A.6**).

La machine *ENS-fermi* (cf. **annexe A.6**) possède plusieurs particularités. Elle intègre des processeurs comportant chacun deux mémoires caches partagées de niveau 3 ainsi que des effets *NUMA* internes. Deux accélérateurs de type processeurs graphiques sont également rattachés au même nœud *NUIOA*. La **figure 4.8** présente la hiérarchie adoptée pour ce nœud asymétrique. Le pôle 1 contient quatre équipes. Deux des équipes sont associées aux deux unités accélératrices et les deux autres sont associées aux deux groupes de cœurs en fonction des mémoires caches de niveau 3. Plusieurs associations aux listes de super-tâches sont possibles mais elles restent équivalentes. Ce choix est déterminé en fonction du numéro d'identification des unités de calcul.

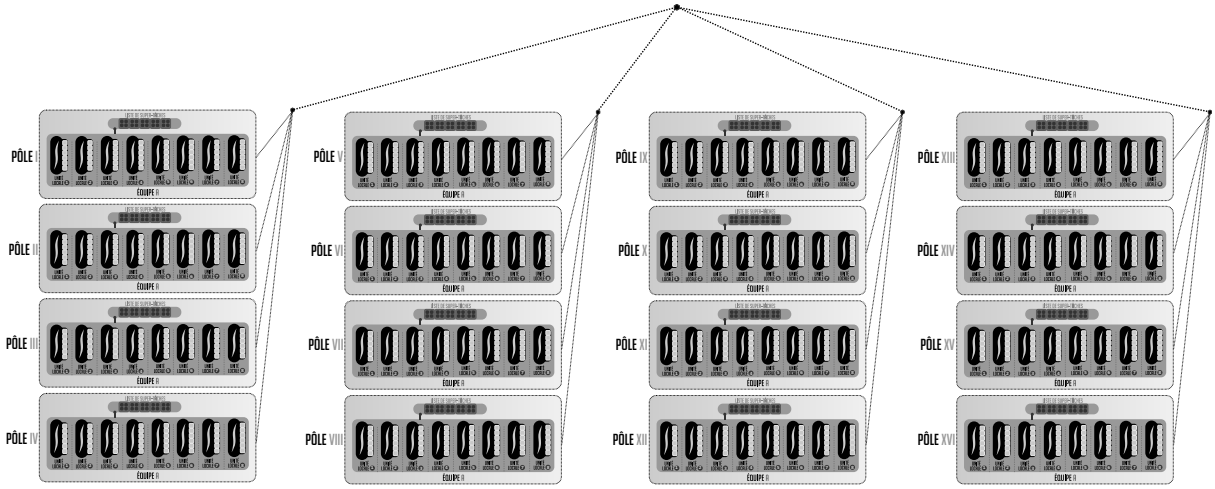


FIGURE 4.9 – Organisation abstraite, exemple composé de seize pôles homogènes pour un nœud de calcul *Curie large* (cf. **annexe A.4**).

Le nœud de calcul *Curie large* (cf. **annexe A.4**) est homogène en termes d'unités de calcul. Il embarque seize processeurs généralistes avec deux niveaux d'accès *NUMA*. La **figure 4.9** présente l'imbrication des 16 pôles pour tenir compte des facteurs *NUMA* associés.

4.3 Placement initial des données

Les performances d'un code de calcul peuvent être conditionnées par la répartition initiale des données. Lors de l'attribution d'une tâche de calcul, il est possible de tenir compte de l'appartenance des données associées à un nœud NUMA. Par exemple, une affinité à un nœud peut être évaluée pour chaque super-tâche en fonction des données dont elle dépend, afin de réduire les accès distants. Il convient ensuite d'aiguiller ces super-tâches vers les unités logiques les plus appropriées en fonction des affinités calculées. Lorsque les données sont initialement concentrées sur un seul nœud NUMA, cela implique un déséquilibre trop important en termes d'affinités et les répartitions peuvent initialement se focaliser sur seulement un seul pôle de calcul, entraînant ainsi un nombre conséquent d'accès distants. Par défaut, sous Linux les pages mémoires sont réparties sur le nœud NUMA ayant une plus grande affinité avec le cœur de calcul qui accède pour la première fois aux données de la page [232]. Cette attribution est bien adaptée au modèle de programmation MPI. En effet, chaque instance MPI accède localement aux données avant d'envoyer par passage de messages les informations nécessaires aux autres instances. Ainsi, les données sont initialisées et modifiées par la même instance MPI. La localité NUMA est préservée tout au long du calcul. A contrario, pour d'autres modèles de programmation où les données ne sont pas dupliquées, le placement initial doit être attentivement étudié puisqu'il peut entraîner de sévères dégradations de performances. La pire des situations survient lorsque le mode d'allocation par défaut est conservé et que les données sont initialisées séquentiellement par le même cœur de calcul. Les données sont alors placée en priorité sur les nœud NUMA associé. Lorsque les cœurs issus d'autres nœuds NUMA utilisent ces données, les accès sont alors constamment distants. Cela peut conduire à une congestion des bus et accroître sensiblement la latence. L'accès aux données

par les accélérateurs amplifient également la pression sur le nœud hébergeant les données.

Sous *LINUX*, les programmeurs peuvent recourir à la bibliothèque *libnuma* [233]. Celle-ci peut être employée pour forcer l'allocation de la mémoire sur des *nœuds NUMA* spécifiques ou de changer la politique de répartition de page. Elle permet quelques schémas d'allocation tels que la répartition alternée (ou entrelacée) des pages sur tous les *nœuds NUMA*. Il est également possible de spécifier une allocation sur un nœud particulier. L'allocateur embarqué dans MPC permet également ce type de placement [234]. Or, il est parfois souhaitable de garder un contrôle plus fin sur certaines allocations. Par exemple, lorsque de gros vecteurs ou matrices sont manipulés, des placements par blocs de pages pourraient être mieux adaptés aux traitements des super-taches qui y seront associées. *Minas* [235] est une bibliothèque qui offre une telle fonctionnalité. Elle permet de placer automatiquement ou explicitement des données sur une machine à effets non uniforme d'accès à la mémoire. Plusieurs politiques de répartition sont possibles notamment pour des tableaux de grandes tailles. Certains concepts de placement on été évalués avec Charm++ (cf. **chapitre 3.3.2**, page 68) [236].

Cette section décrit les mécanismes intégrés au module ***COMPAS***[107], élaboré pour les besoins de cette thèse afin de mieux aiguiller l'ordonnancement hétérogène en fonction de la localité spatiale. Tout comme *Minas*, *COMPAS* permet de garder la main sur des placements initiaux en définissant des schémas de répartition de plus haut niveau que ceux proposés par la bibliothèque *libnuma*. Cependant, il se focalise sur un placement de données dédié à un ordonnancement de tâches de calcul. En particulier, plusieurs informations complémentaires sont conservées pour guider l'attribution des calculs au cours de l'exécution. Les interfaces de programmation sont présentées dans cette section et les performances sont ensuite évaluées sur une multiplication de matrice.

4.3.1 Remplacement de la fonction d'allocation

Le système gère un placement des données à l'échelle des pages mémoires. Lorsque une quantité de mémoire est réclamée par un programme, une granularité plus fine peut être utile afin de réduire le gaspillage des ressources mémoires dû à l'exploitation partielle de certaines pages. Un allocateur mémoire permet justement une meilleure utilisation de ces ressources. Il gère lui même la récupération des pages et peut placer consécutivement plusieurs zones mémoires réclamées. Il tente ainsi de prendre en charge la problème complexe de la fragmentation de la mémoire, c'est à dire de diminuer les espaces mémoires réservés en mémoire physique non exploitables. Lorsqu'une plage mémoire est libérée et rendue à l'allocateur, celui-ci peut alors l'attribuer à une autre demande de ressource. Cela permet d'accélérer les allocations en évitant de passer par des appels systèmes coûteux occasionnés par la récupération de nouvelles pages. Néanmoins, à cause de ces mécanismes, les pages mémoires peuvent déjà avoir servies et être déjà positionnées sur des nœuds NUMA. Le contrôle de la répartition des données devient compliqué, voire dans certains cas impossible.

L'allocateur retourne une adresse mémoire à partir de laquelle débute la zone réservée. Les mécanismes associés posent des problèmes de deux natures. D'une part, l'allocation ne permet d'obtenir une affinité qu'à un unique nœud NUMA pour tout l'espace souhaité. Une sous-répartition sur plusieurs nœuds n'est pas prévu par la plupart des allocateurs qui doivent faire face à la fragmentation des données. D'autre part, l'alignement de la zone allouée n'est pas contrôlable. Une allocation inférieure à la capacité d'une page peut être associée à zone mémoire à cheval sur deux pages mémoires, lesquelles peuvent être réparties sur différents nœuds NUMA.

[107] **COMPAS :** *Coordinate and Organize Memory Placement and Allocation for Scheduler*.

Afin de conserver la main sur le placement des pages au sein d'une zone mémoire, les appels à la fonction d'allocation (*malloc* en C) sont replacés par un traitement plus adapté à la situation. Typiquement, les demandes de ressources en mémoire dont la taille est plus petite que la contenance d'une page, n'ont pas besoin d'une répartition spécifique autre qu'une affinité particulière à *nœud NUMA*. Dans ce cas de figure, la demande est redirigée vers l'allocateur. Dans le cas contraire, des pages sont récupérées et placées selon un schéma spécifié par le programmeur. Toutes ces informations de répartition sont ensuite mémorisées pour permettre, le cas échéant, une libération adaptée. La mémoire est soit libérée en rendant directement les pages au système, soit en laissant l'allocateur s'en charger s'il est à l'origine de la réservation. Ces informations permettent également de conserver la position de chaque page, indication qui est primordiale pour un ordonnanceur tenant compte de la localité. Il est intéressant de noter que cette information n'est pas fournie par la bibliothèque *libnuma*, ce qui renforce l'intérêt porté au contrôle du positionnement des données, combiné au maintien des informations associées.

Cette approche présente quelques limitations. Tout d'abord, puisque qu'une partie des réservations ne repose pas sur un allocateur, des pages mémoires peuvent être sous-exploitées et mener à une sur-consommation de la mémoire. Cela ne peut concerner que la dernière page de chaque zone mémoire. Par conséquent, plus la taille de chaque allocation est conséquente, moins de mémoire sera potentiellement perdue en proportion. De plus, les données allouées *statiquement* sur la pile ne sont pas gérées. Des solutions existent pour transformer à la compilation ces allocations en allocations dynamiques en y générant les appels à un allocateur pour les quantités de mémoire demandées. Par exemple, le module *MApp*[108] de *Minas* permet cela.

4.3.2 Les schémas de répartition des pages mémoires

La figure 4.10 est un exemple d'allocation dont le placement de pages est spécifié par le programmeur. Une matrice de 6144×6144 nombres à virgule flottante en simple précision (*float*) est allouée par la fonction *compas_malloc*. Cet appel permet le placement des pages mémoires selon le schéma spécifié en amont. En l'occurrence, une répartition par blocs de 2×2 pages est demandée. Le placement de ces blocs est souhaité de manière alternée (*COMPAS_CYCLIC*) sur les nœuds NUMA et les pages mémoire sont verrouillées (cf. chapitre 2.4.1, page 48) avec l'argument *COMPAS_PLOCKED*.

Code : distribution des pages via un appel explicite (COMPAS)

```
1  /* Allocation d'une matrice de 6144x1644 float et repartition des pages */
2  struct compas_pages_s pages;
3  compas_pages_init(&pages);
4  pages.geometry = COMPAS_2D;
5  pages.pattern = COMPAS_CYCLIC;
6  pages.xblocks = 3;
7  pages.yblocks = 3;
8  pages.xblocksize = 2;
9  pages.yblocksize = 2;
10 pages.elsize = sizeof(float);
11 float *A = compas_malloc(&pages, COMPAS_PLOCKED,6144*6144*sizeof(float));
```

FIGURE 4.10 – Allocation précédée d'un appel explicite de répartition des pages. Cas d'une matrice de 6144×6144 *float*, décomposée en 3×3 blocs de 2×2 pages mémoire.

[108] **MApp :** *Memory Affinity preprocessor*.

Le résultat est représenté par la figure 4.11. Des placement par colonnes, lignes ou entièrement décrits par le programmeur peuvent être également choisis, tout comme la dimension des blocs (1D, 2D, 3D).

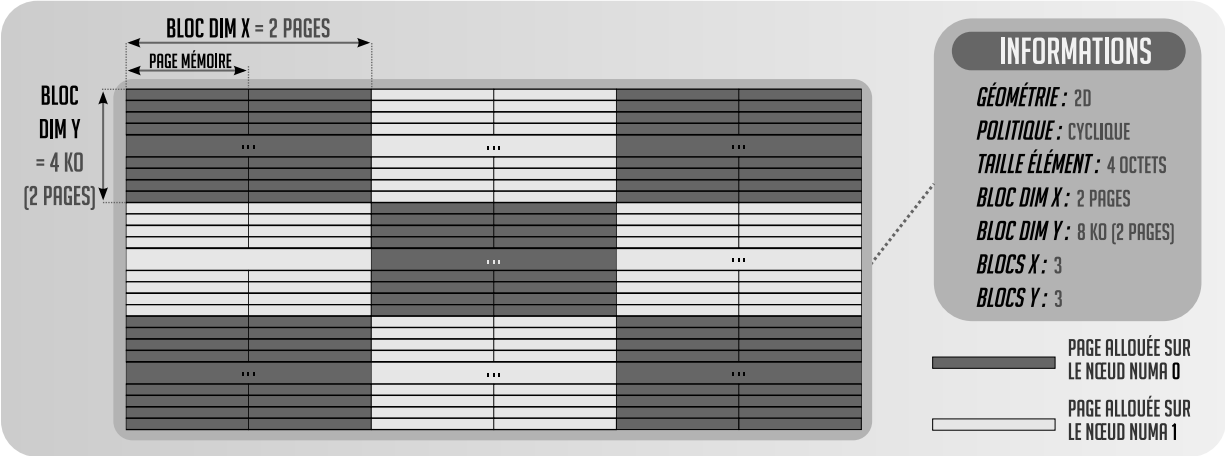


FIGURE 4.11 – Exemple d'une allocation par blocs 2D pour une matrice de nombre flottants à simple précision (*float*). Les blocs ont une granularité équivalente à 2×2 pages mémoires, soit dans cette situation 8192×8192 octets. La politique de répartition *cyclique* permet d'associer alternativement des blocs sur les *nœuds NUMA*.

4.3.3 Granularités

Une fois les pages mémoires réparties sur les nœuds NUMA, la décomposition des données en blocs associée aux super-tâches doit, autant que possible, coïncider avec cette répartition. Par défaut c'est cette décomposition qui est choisie, mais elle peut être remplacée par le programmeur pour les besoins de l'application. Cette définition manuelle sera présentée dans le chapitre 5 (page 97). Une mauvaise association impliquera inévitablement plus d'accès distants. Cette granularité peut être redéfinie au cours du programme au détriment des performances car toutes les données sont alors rendues cohérentes en mémoire centrale, rapatriant aussi toutes les données contenues dans les mémoires déportées.

4.3.4 Les informations de localité spatiale pour aiguiller les super-tâches

Les informations d'affinités NUMA sont ensuite transmises au blocs manipulés par les super-tâches. Ces informations sont intégrées dans la prise de décision pour l'assignation d'une super-tâche à un pôle. Le choix d'une liste particulière dans le pôle tient compte d'autres éléments qui seront détaillés dans le chapitre suivant. Il convient de considérer les types d'accès et la taille des différents blocs manipulés par les super-tâches, puisqu'ils peuvent impliquer une plus ou moins grande quantité d'accès distants. Les dépendances en écriture ou lecture seules ne comptent que pour un accès de la taille du bloc. En revanche, les dépendances en lecture et écriture comptent pour deux accès. La figure 4.12 illustre ce principe sur une super-tâche issue d'une multiplication de matrices. Si la dimension y de de chaque bloc contient 16 valeurs (16 lignes par blocs) et que chaque valeur est codé sur 4 octets, alors la super-tâche accédera à 512 Ko stockés dans le *nœud NUMA 0* et $2\,048$ Ko de données du *nœud 1*. Assigner cette tâche au pôle de calcul associé au *nœud 1* permet de réduire par 4 les accès distants par rapport à une exécution sur le *nœud 0*.

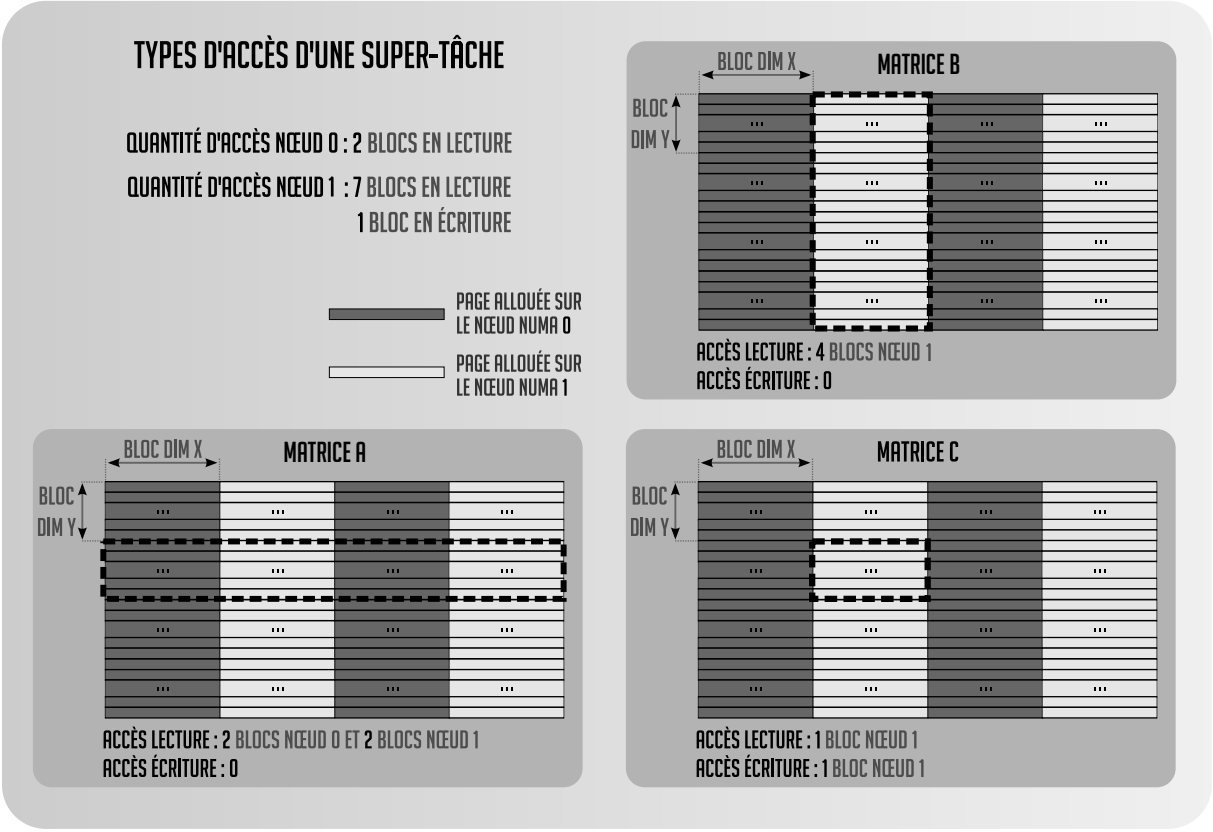


FIGURE 4.12 – Exemple de transferts locaux et distants à partir des types d'accès et du placement des blocs de données. Super-tâche issue d'une multiplication de matrices de type $C = C + A \times B$

De manière générale les super-tâches sont assignées au pôle qui minimise les accès NUMA distants. En cas d'égalité de répartition des données sur les nœuds, une répartition aléatoire est choisie. D'autres propositions d'ordonnancements plus complets en tenant comptes d'autres facteurs sont abordés dans le chapitre 6 (page 113).

4.3.5 Évaluations

La figure 4.13 complète la figure 4.3 présentée précédemment. La nouvelle courbe est associée à un ordonnancement hétérogène à multi-granularité avec une répartition préalable des pages mémoires. Pour chaque matrice, les schémas de placement des pages correspondent à la granularité des super-tâches. Les blocs de 1024×1024 *float* (équivalent à une page mémoire par dimension) sont répartis sur les différents nœuds NUMA à l'allocation de manière cyclique. Pour cette évaluation, c'est grosso modo le placement sur un nœud NUMA du bloc en lecture-écriture (i.e. le bloc de la matrice C dans $C = C + A \times B$) qui détermine l'assignation de la super-tâche à un pôle. Ainsi, a moitié des blocs est répartie sur chacun des deux nœuds NUMA de la machine cible. Les accès distants sont réduits et les performances des transferts entre la mémoire centrale et les accélérateurs sont ainsi améliorées.

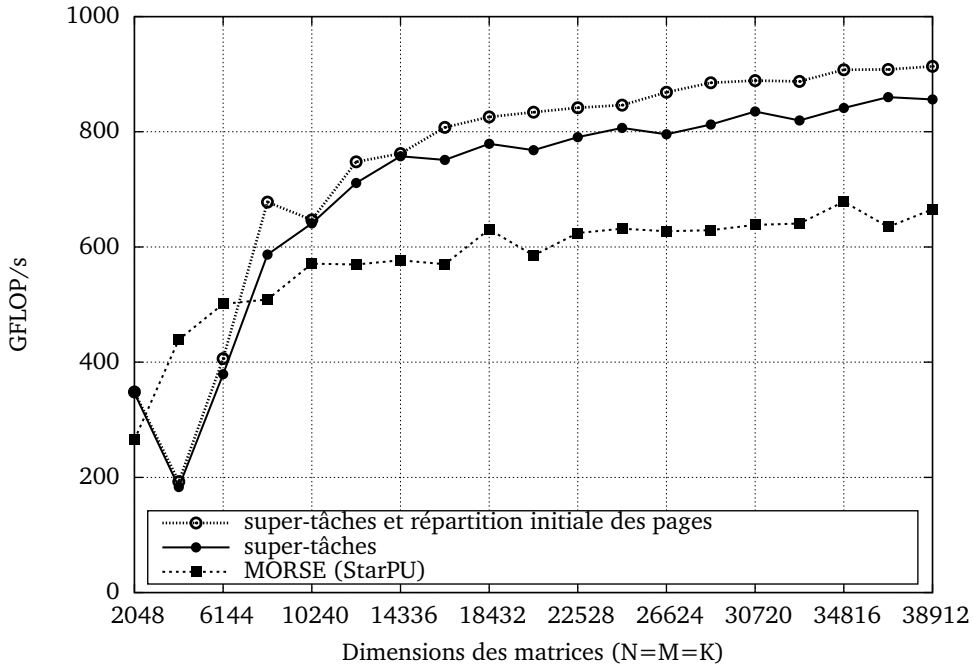


FIGURE 4.13 – *SGEMM* hétérogènes avec deux processeurs Intel Xeon et deux accélérateurs Nvidia M2050 (*Inti hétérogène*, cf. **annexe A.1**), décomposition en blocs de 1024×1024 valeurs. Les temps de transfert des données sont compris. Comparaison entre MAGMA (MORSE/StarPU), et un ordonnancement basé sur des super-tâches avec et sans placement initial des pages mémoires.

Pour résumer, des *super-tâches* hétérogènes à granularité variable ont été présentées dans une première partie. Elles permettent de concilier les avantages des tâches à faibles et à grosses granularités avec des processeurs hétérogènes. Elle offrent un meilleur équilibrage de charge dynamique entre ressources de calcul hétérogène en embarquant moins d'information *a priori*. Cette approche apporte un gain de 20% sur des multiplications de matrices avec des ressources de calcul hétérogènes par rapport à la bibliothèque de référence *Magma*. L'utilité du module *COMPAS* a été également abordée. Celui-ci offre un contrôle de plus haut niveau sur le positionnement initial des données par rapport à la bibliothèque *libnuma*. De plus, il aiguille l'attribution des super-tâches en fonction de ces placements. Associé à une organisation hiérarchique des unités de calcul, un gain supplémentaire de 5% est atteint sur le test de multiplications de matrices.

Chapitre 5

Gestion des données favorisant la localité temporelle

> *“ Aussitôt qu'ils sont déplacés, ils perdent toutes les données antérieures qui pouvaient éclairer leur jugement. ”*
>
> Benjamin Constant, *Principes de politique*, 1815

Le chapitre précédent s'est concentré sur la localité spatiale, laquelle permet une meilleure organisation des données pour limiter les accès distants lors de l'ordonnancement des super-tâches. Néanmoins, certains de ces accès coûteux surviennent inévitablement afin de maintenir un état cohérent des données manipulées depuis les diverses mémoires. Ainsi, des transferts sont nécessaires pour que plusieurs unités de calcul puissent accéder à une version à jour des données. Du fait de l'augmentation progressive des cœurs de calcul dans les processeurs, certains prévoient une évolution vers des architectures comportant des mémoires purement distribuées dont la cohérence matérielle n'est plus maintenue [237]. Actuellement, l'emploi d'accélérateurs déportés pose déjà un problème similaire en embarquant leur propre mémoire qui n'est pas rendue cohérente avec celle du système. En d'autres termes, c'est la couche logicielle qui doit se charger de déclencher les mouvements de données adéquats. Les modèles de programmation permettant l'exploitation de certains accélérateurs, tels que *CUDA* ou *OpenCL* confient ces actions directement au programmeur. Ce dernier est donc chargé de renseigner les paramètres adaptés pour que les données soient véhiculées, au moment opportun, d'une mémoire à une autre. Pourtant, ces opérations peuvent être prises en charge par une plate-forme d'exécution, moyennant le renseignement préalable de quelques informations. Outre l'intérêt indéniable que cela présente en termes de programmabilité, certains mécanismes peuvent y être adjoints et réduire la quantité globale des accès distants.

Les mémoires caches, en particulier, permettent une rétention locale des données. Les mécanismes associés peuvent être déployés de manière logicielle sur n'importe quelle mémoire contrôlable, telle que la mémoire embarquée dans un accélérateur. Ainsi, des données sont maintenues dans une mémoire associée à une ou plusieurs unités d'exécution, tant qu'elles ne sont pas modifiées ailleurs. L'efficacité des caches logiciels peut même être amplifiée à condition que les décisions d'ordonnancement tiennent compte des données qui y sont présentes. La difficulté réside dans l'attribution des nouvelles tâches pour réduire les mouvements de données. Tout en maintenant un équilibrage de charge global, il s'agit de diriger les tâches de calcul vers les unités d'exécution dont les mémoires contiennent déjà tout ou une partie des données dans leur état cohérent. Une mauvaise attribution des tâches peut dégrader l'efficacité des mécanismes de rétention en générant des déplacements de données trop fréquents. Dans son ouvrage *Principes de politique*, Benjamin Constant soutient que les performances et l'intelligence des soldats varient en

fonction de leur localisation. Cette vision contestable du système militaire semble toutefois bien refléter le problème de localité des données dans des caches logiciels : lorsque des données sont présentes dans un cache, des actions adéquates peuvent être menées pour favoriser leur maintien en place. Ainsi, en attribuant à l'unité de calcul correspondante des tâches dont les données sont déjà préchargées, la localité temporelle est exploitée. En revanche, lorsqu'aucun cache logiciel ne contient les données suite à un déplacement, les informations de localité temporelles disparaissent. Une décision arbitraire d'ordonnancement s'opère, laquelle peut entraîner un mouvement plus important de données par la suite. Cette situation peut survenir fréquemment car la contenance des caches logiciels est bien entendu limitée par la capacité physique de la mémoire employée. L'insertion d'une donnée implique alors l'évacuation d'une ou plusieurs autres. Une nouvelle politique de gestion de cache associée aux avantages de la programmation par tâches de calcul est détaillée dans ce chapitre. Elle permet d'éclairer la plate-forme sur les données à évincer en priorité, en s'appuyant sur une prévision globale de leur utilisation.

Parfois, la rétention des données doit être annihilée afin qu'un traitement puisse accéder à des données cohérentes en mémoire centrale. C'est notamment les cas des bibliothèques de calcul hétérogènes qui, pour des raisons de portabilité, ne sont pas élaborées pour une plate-forme d'exécution spécifique. Ainsi, avant et après chaque appel, une évacuation complète des caches logiciels doit être déclenchée. Si plusieurs appels manipulent les mêmes données, de nombreux déplacements peuvent en résulter. Une interface peut être adoptée pour rendre possible une rétention entre ces appels de bibliothèques et d'éviter, dans la plupart des cas, des transferts superflus.

Ce chapitre se focalise sur l'amplification de la localité temporelle en s'appuyant sur les avantages que présente la programmation par tâches de calcul. Des contributions liées à une gestion automatisée des données en réduisant la quantité de données transférées y sont détaillées. La première section aborde les mécanismes de maintien de la cohérence et de l'adaptation de la disposition des données en mémoire en étendant le module *COMPAS* (cf. **chapitre 4.3**, page 90). La section suivante, présente l'intérêt d'une politique d'éviction globale des données conservées dans les caches logiciels, couplée à un ordonnancement adapté. Cette politique d'éviction est évaluée sur un code régulier de factorisation LU de matrices denses et creuses. Enfin, la dernière section propose une interface permettant de conserver les bénéfices des caches logiciels en facilitant l'exécution conjointe des tâches de calcul avec des traitements issus de bibliothèques hétérogènes. Le gain de performance est mesuré sur le test de multiplications de matrices déjà employé dans le chapitre précédent.

5.1 Gestion autonome des données

Certaines mémoires ne sont pas rendues cohérentes entre elles par le matériel. Par conséquent, le programmeur doit s'assurer que les versions à jour des données soient préalablement chargées avant d'être exploitées par les diverses ressources de calcul. Ces mouvements de données sont initiés en indiquant différents paramètres tels que la mémoire source contenant les données à propager, la mémoire de destination, ainsi que la quantité d'information à copier. Ainsi, un nœud de calcul intégrant deux accélérateurs, peut être composé de trois mémoires programmables disjointes en comptant la mémoire centrale. Cette gestion explicite complexifie l'écriture des codes de calculs et multiplie également les risques d'erreurs. Une plate-forme d'exécution peut prendre à charge cette gestion et décharger le programmeur de cette tâche fastidieuse. Cette section détaille les informations utiles à la plate-forme, puis décrit les mécanismes (cf. figure 5.1) permettant une telle gestion autonome de la cohérence des données. Enfin, d'autres avantages qu'offre cette automatisation combinée à un modèle de programmation par tâches de calcul sont présentés.

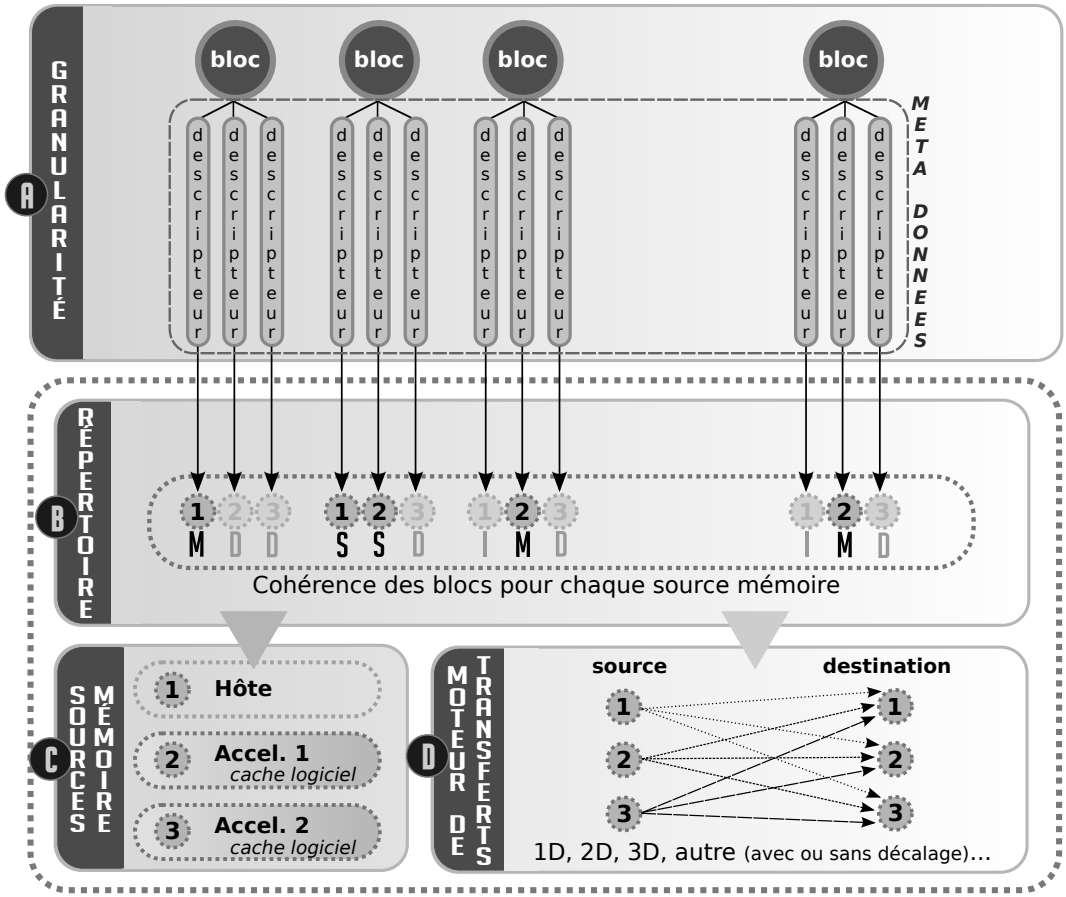


FIGURE 5.1 – Aperçu des mécanismes permettant une gestion automatisée des données.

5.1.1 Enregistrement des blocs de données

Afin de profiter d'une gestion automatisée des données, le programmeur est avant tout chargé de décrire les blocs de données à partir desquels la cohérence sera assurée. Cette définition correspond également à la façon dont les données vont être globalement accédées par les super-tâches. Ces blocs doivent contenir une quantité suffisante de données afin d'éviter les problèmes évoqués

dans le chapitre 2.4.1 (page 48) concernant les performances des transferts via le bus *PCIe*. La disposition des ces blocs requiert, autant que possible, d'être harmonisée avec les placements des pages mémoires, abordé dans le chapitre 4.3.2 (page 92) afin de limiter les effets *NUMA*.

La figure 5.2 décrit la façon dont 144 blocs de 512×512 valeurs à virgules flottantes sont définis par le programmeur. Cette déclaration enrichit le module *COMPAS* et vient en complément des informations d'allocations introduits par la figure 4.10. Ainsi, ce type de déclaration permet d'enregistrer la taille, la disposition des blocs (1D, 2D, 3D ou personnalisée), de manière analogue à la répartition des pages sur les différents *nœuds NUMA*. Elle peut être redéfinie en cours d'exécution pour mieux correspondre aux besoins spécifiques de chaque application. En contrepartie, une telle réorganisation déclenche un rapatriement des données conservées dans les mémoires déportées vers la mémoire centrale, afin de réinitialiser les schéma d'accès et la cohérence des nouveaux blocs. L'enregistrement d'un bloc de données génère des *méta-données* (cf. figure 5.1A) lesquelles permettent, entre autres, de conserver les caractéristiques de stockage des blocs dans chaque mémoire et de permettre un suivi de leur cohérence.

Code : enregistrement de blocs de données (COMPAS)

```
1  /* Declaration de 12x12 blocs de 512*512 float */
2  float *A = compas_malloc(&pages, COMPAS_PLOCKED, 6144*6144*sizeof(float));
3
4  int blocs = 12; /* 6144/512  */
5  struct compas_data_s data;
6
7  compas_data_init(&data);
8  data.geometry = COMPAS_2D;
9  data.xsize = 512;
10 data.xstride = 6144*sizeof(float);
11 data.ysize = 512;
12 data.elsize = sizeof(float);
13
14 for (int i=0; i<(blocs); i++) {
15   for (int j=0; j<(blocs); j++) {
16     compas_data_register(&ptr[i*6144*512+j*512], &data, COMPAS_COLMAJOR);
17   }
18 }
```

FIGURE 5.2 – Enregistrement de 144 blocs à deux dimensions, chacun composé de 512×512 *float*.

5.1.2 Protocole de cohérence des blocs de données

L'état de cohérence de tous les blocs dans chaque mémoire est stocké dans un module qui joue en quelque sorte le rôle d'un *répertoire* chapitre 2.3.1 (page 42) en permettant d'identifier l'emplacement des blocs à jour (cf. figure 5.1B). Un simple protocole de cohérence est adopté en suivant un mode en réécriture tardive (*write-back*). Il se rapproche du protocole *MSI* décrit dans le chapitre 2.2.2 (page 38). L'état *invalide (I)* est remplacé par l'état *supprimé (D)* pour les mémoires déportées, afin d'éviter d'occuper inutilement de l'espace. Au début du programme, toutes les données sont dans l'état *exclusif (E)* en mémoire centrale et aucune donnée n'est présente dans les mémoires déportées (état *supprimé*). Si un bloc en lecture seule est transférée vers une mémoire déportée, l'état du bloc passe à *partagé (S)* dans toutes les mémoires qui le contient. Lorsqu'un bloc est accédé en écriture dans une mémoire, son état passe à *modifié (M)* ou *exclusif (E)* pour le cas de la mémoire centrale. Depuis les autres mémoires, ce même bloc est soit *invalidé* (mémoire centrale), soit *supprimé* (mémoires déportées). La figure 5.3 illustre ce mécanisme.

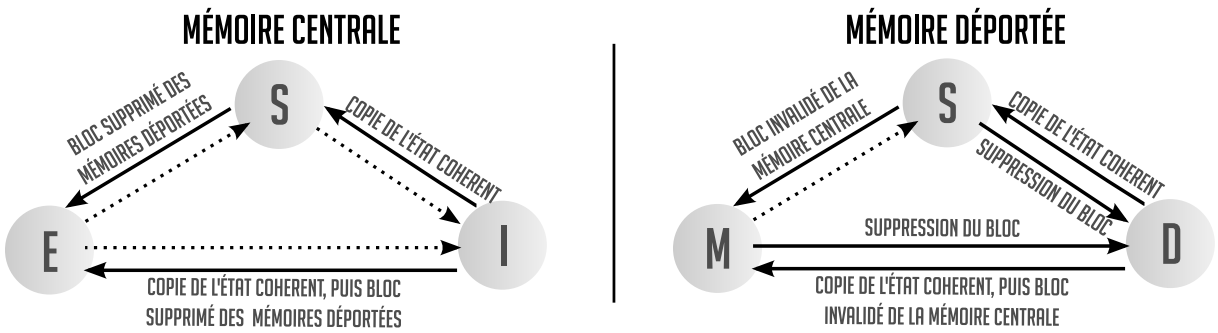


FIGURE 5.3 – Protocole logiciel de la cohérence des données. L'état *invalide (I)* est remplacé par l'état *supprimé (D)* pour les mémoires déportées.

5.1.3 Transferts automatisés des blocs de données

Une gestion autonome des transferts présente plusieurs intérêts. La charge du programmeur est allégée, tout en facilitant le déploiement des mécanismes d'équilibrage de charge dynamique entre les diverses ressources de calcul hétérogènes. Lorsqu'une super-tâche est assignée à une équipe, les blocs de données qui en dépendent sont le cas échéant véhiculées vers la mémoire adéquate. L'exécution des transferts est dicté par l'état de cohérence de chaque bloc. Le déclenchement des transferts est ensuite pris en charge par le module de *transferts* (cf. **figure 5.1D**). Ce dernier apporte des optimisations en termes de copies et de stockage des données.

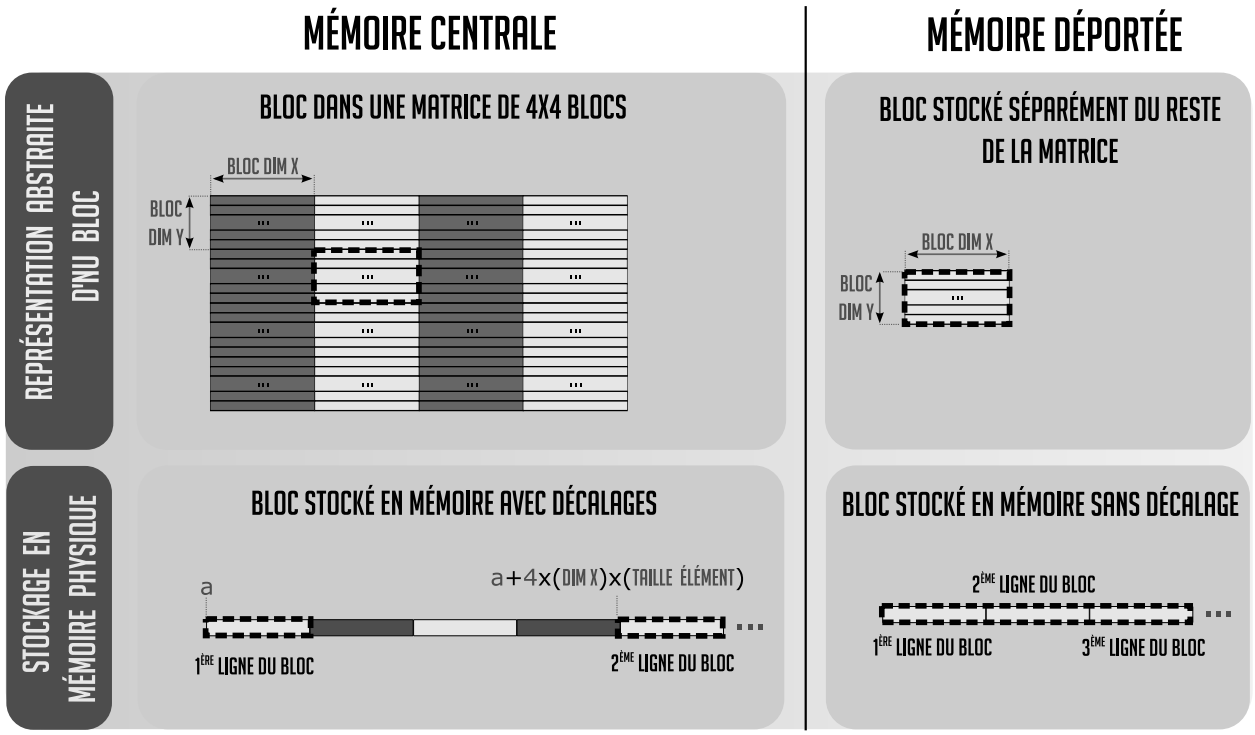


FIGURE 5.4 – Exemple d'adaptation du stockage d'un bloc. Cas d'un bloc à deux dimensions issu d'une matrice dont le stockage est défragmenté dans les mémoires déportées.

La disposition des données en mémoire peut être adaptée en fonction de la mémoire ciblée. Cela permet à la fois d'optimiser l'espace occupé et de réduire la quantité de données transférées. En effet, les données associées à un bloc de cohérence ne sont pas nécessairement tous stockées

de manière consécutive en mémoire. En particulier, pour des blocs à plusieurs dimensions, des décalages sont a considérer. La taille de ces décalages correspond aux données non-concernées par le blocs et dont il faut faire abstraction. Lorsque les données sont déplacées vers la mémoire d'un accélérateur, elles peuvent y être défragmentées afin d'occuper moins d'espace en se concentrant uniquement sur les valeurs nécessaires. La figure 5.4 illustre ce mécanise de défragmentation à la volée dans les mémoires déportées à partir d'une matrice associée à des blocs de cohérence à deux dimensions. Les calculs effectués à partir de ces données défragmentées doivent être également adaptés afin de tenir compte de leur nouvelle disposition.

Les mémoires déportées possèdent typiquement une capacité bien plus limitée que celle de la mémoire centrale. Il est possible que la totalité des données utiles à une opération de calcul dépasse cette capacité. L'accélération de ce traitement devient alors complexe à programmer sans appui logiciel. La séparation en blocs de données, la gestion automatisée de leur cohérence et leur défragmentation permettent de résoudre ce problème d'exploitation et d'employer des accélérateurs sur des codes de calcul traitant une grande quantité de données. Les super-tâches se focalisent sur une partie des données qui peut être temporairement stockée dans une mémoire déportées jusqu'à ce que les opérations associées soient achevées. Les calculs et leurs données concernées peuvent donc être déportés au fur et à mesure, en toute transparence pour le programmeur. La figure 5.5 matérialise ce principe de déports et d'exécutions progressives.

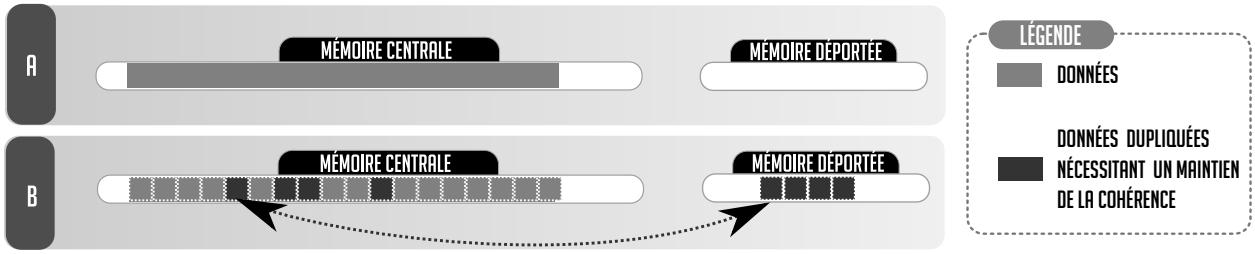


FIGURE 5.5 – Gestion de la mémoire par blocs. **a)** Sans morcellement, ni cohérence les données ne peuvent pas être transférées automatiquement vers une mémoire déportée car l'espace qu'elles occupent excède la capacité disponible. **b)** Avec séparation en tâches de calcul, gestion adéquate de la cohérence et éventuellement déframentation des données, les opérations s'opèrent au fur et à mesure sur les données déportées.

5.2 Cache logiciel pour mémoire déportée

Des caches logiciels sont rattachés aux accélérateurs pour maintenir localement des blocs de données qui sont accédées plusieurs fois. Cette rétention permet de réduire la quantité globale de transferts et d'alléger la pression sur les bus. Le gain de temps économisé sur ces communications permet ainsi d'accroître la vitesse d'exécution des applications qui peuvent bénéficier d'une localité temporelle. La totalité des mémoires embarquées est ainsi exploitée afin de conserver un maximum d'informations potentiellement utiles ultérieurement. Lorsque le cache est plein et qu'un nouveau bloc de données doit y être inséré, un ou plusieurs autres blocs doivent être évacués et rapatriés vers la mémoire centrale pour préserver un état cohérent. L'espace libéré peut alors être employé pour le nouveau bloc. La sélection des données à évacuer, appelées *victimes*, dépend entre autres de la capacité et de la politique d'éviction du cache. La politique la plus répandue consiste à évacuer en premier les données qui ont été les moins récemment employées (LRU). Une autre politique courante consiste à évincer en priorité les données qui ont

été le moins fréquemment employées (LFU[109]), ce qui permet ainsi de miser sur la probabilité de leur réutilisation en fonction des accès passés.

La figure 5.6a démontre l'intérêt d'un cache logiciel sur l'étape 3 d'une factorisations LU d'une matrice dense de 25×25 blocs de chacun 1024×1024 valeurs à virgule flottante en double précision. En associant une cache logiciel d'environ 1,17 Go (150 blocs de 1024×1024 double) à un accélérateur *Nvidia Tesla K20*, un gain de 40 GFLOP/s peut être obtenu à partir d'une politique d'éviction LRU et d'un simple ordonnancement glouton homogène. La figure 5.6b révèle l'économie en données transférées avec les mêmes paramètres. Une réduction de 25% des données véhiculées est alors mesurée.

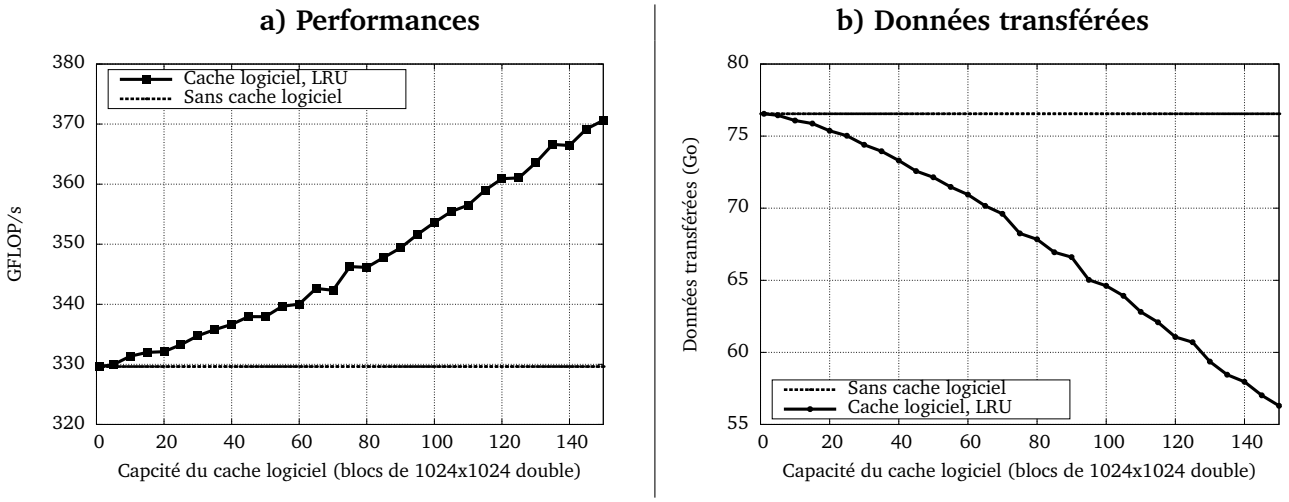


FIGURE 5.6 – Mise en évidence de l'intérêt d'un cache logiciel sur l'étape 3 (multiplications de blocs) d'une factorisation LU d'une matrice dense de 625 blocs de 1024×1024 valeurs à virgule flottante (double précision). Opérations uniquement effectuées par un accélérateur *Nvidia Tesla K20*. **a)** Performances avec et sans cache LRU en fonction de la capacité de ce dernier. **b)** Comparaison de la quantité de données transférées en fonction de la capacité du cache logiciel.

Cette section a pour objectif l'étude de l'efficacité des caches logiciels en termes de réduction de données transférées et leur impact sur le temps d'exécution d'une application. Elle démontre en particulier l'importance des interactions entre un cache logiciel et l'ordre d'exécution des super-taches. Une nouvelle politique d'ordonnancement basée sur une métrique de potentiels de réutilisation des blocs est également présentée.

5.2.1 Interaction entre l'ordonnancement et un cache logiciel

Lorsque le cache logiciel ne possède pas une capacité suffisamment importante pour conserver la majorité des données du programme, la politique d'éviction joue un rôle déterminant dans son efficacité de réutilisation. De mauvais choix d'éviction peuvent mener à une quantité de données transférée bien plus importante que nécessaire. Des données peuvent ainsi être évacuées avant qu'elles aient eu le temps d'être réutilisées. Les caches logiciels emploient majoritairement des politiques d'éviction basées sur des informations d'accès aux données collectées au cours de l'exécution du programme tel que la politique LRU. Dans ces cas bien précis, plus la distance de réutilisation d'un bloc de donnée est important, plus sa probabilité d'être évacué augmente. Or, l'ordre des accès aux blocs de donnés est imposé par l'ordonnancement. Ce dernier peu donc

[109]**LFU :** *Least Frequently Used*.

impacter la distance temporelle de réutilisation des blocs en réorganisant l'ordre d'exécution des super-tâches. Entre deux barrières de synchronisation, les super-tâches peuvent justement être exécutées dans n'importe quel ordre. Ainsi, en planifiant en priorité l'exécution des super-tâches dont des blocs de données sont déjà contenus dans le cache logiciel, l'efficacité de ce dernier peut être accrue. La figure 5.7 illustre ce principe sur un cas simple où le cache logiciel peut contenir deux blocs de données. L'intérêt d'un ordonnancement aiguillé par la présence des données en mémoire cache y est présenté face à un ordonnancement glouton. En réorganisant l'ordre des super-tâches, deux blocs accédés en lecture et écriture peuvent être réutilisés en évitant des transferts.

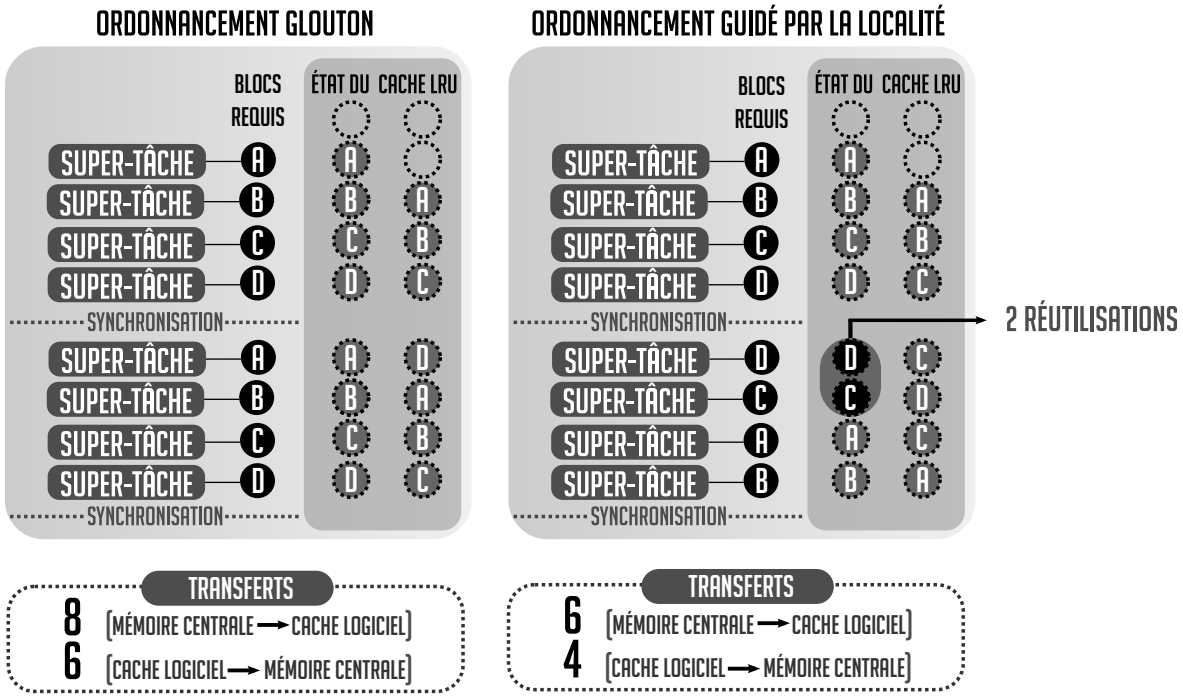


FIGURE 5.7 – Exemple de couplage de l'ordonnanceur avec les informations de présence dans le cache logiciel permettant d'accroître les performances de ce dernier. L'ordre d'exécution des super-tâches est modifié entre deux barrières de synchronisation en fonction de l'état du cache logiciel.

Les figures 5.8a et 5.8b révèlent l'impact de l'ordonnancement sur l'efficacité d'un cache logiciel LRU d'une capacité de 150 blocs de 1024×1024 valeurs sur l'étape 3 de factorisations LU de matrices denses. Lorsque la capacité du cache logiciel ne permet pas de contenir la majorité des données, un ordonnancement dirigé par la localité des blocs permet de réduire la quantité d'évictions et donc de données transférées. Les figures 5.8c et 5.8d comparent des caches logiciels à politique d'éviction LRU et LFU alliés à un ordonnancement glouton ou aiguillé par la localité des blocs. Les deux politiques d'éviction mènent sensiblement au mêmes résultats pour ce type d'application. Plus de 400 GFLOP/s sont atteints avec un cache d'une capacité de 150 blocs et une planification de super-taches orientée par la présence des blocs, contre environ 370 GFLOPS/s pour un ordonnancement glouton.

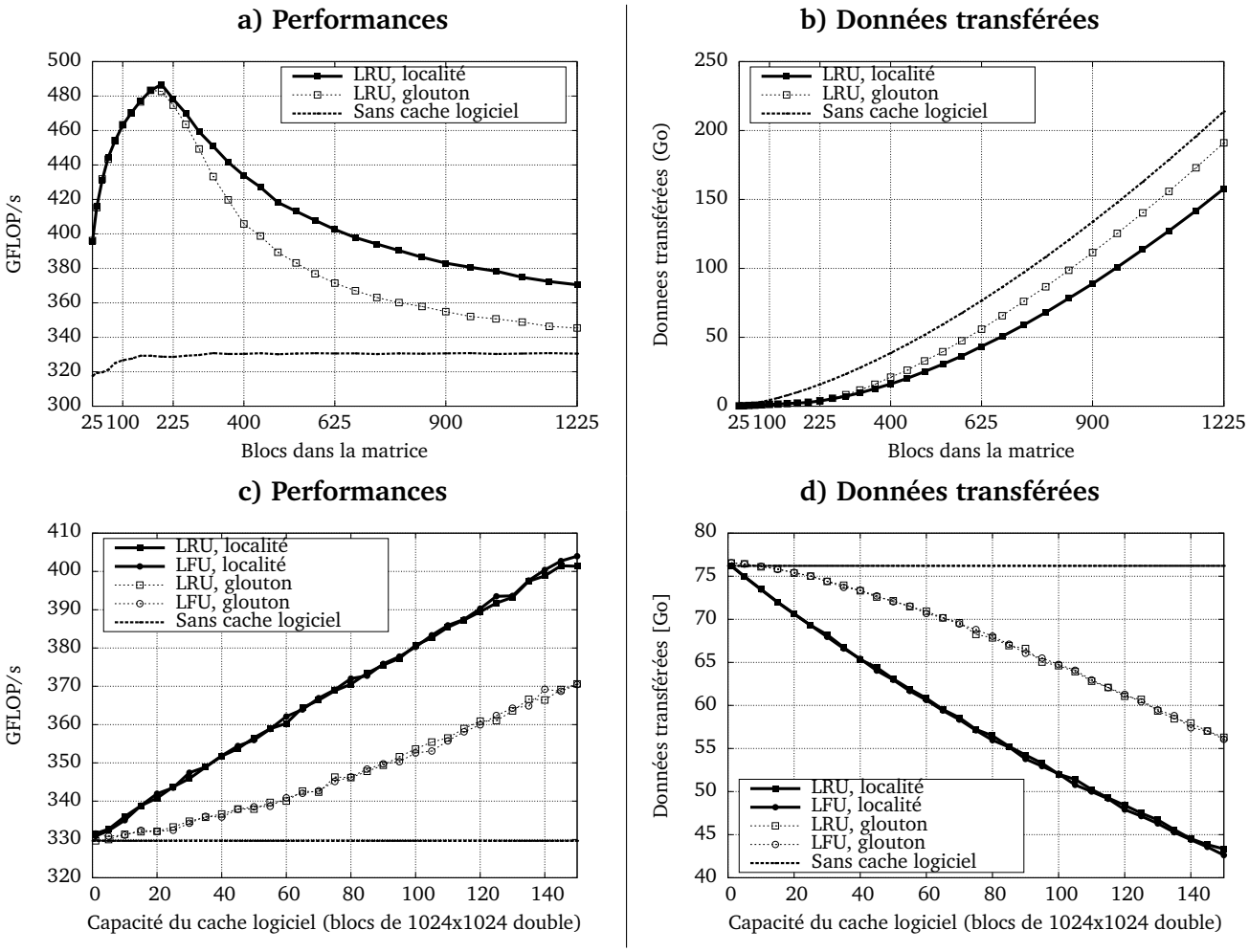


FIGURE 5.8 – Comparaison des politiques d'éviction et d'ordonnancement sur l'étape 3 d'une factorisation LU d'une matrice dense constituée de blocs de 1024×1024 valeurs à virgule flottante en double précision. Les opérations sont exécutée sur un accélérateur *Nvidia Tesla K20* (*Cirrus hétérogène*, cf. **annexe A.5**). **a-b)** La capacité du cache logiciel est fixée à 150 blocs. **c-d)** La matrice employée contient 25×25 blocs.

5.2.2 Politique d'éviction basée sur la réutilisation des données

Les politiques d'éviction LFU et LRU présentent l'avantage d'être simples à mettre en œuvre. Cependant, elles sont très sensibles aux choix d'ordonnancement et peuvent par exemple mener à sous exploitation des caches logiciels pour des codes de calcul irréguliers qui requièrent un équilibrage de charge constant.

La méthode LFU peut aussi conduire à une pollution du cache, c'est à dire que les données réutilisées très fréquemment en début de programme, peuvent rester pendant toute la durée de vie de l'application en cache logiciel, même si elles ne sont plus utiles. Cela diminue artificiellement la capacité du cache qui doit donc se concentrer sur une capacité exploitable réduite.

Cette sous-section explore l'opportunité d'exploiter une métrique qui correspond davantage au fonctionnement spécifique d'un programme. Le fait que les données soient déjà étendues par des méta-données et que leur géométrie soit déjà renseignée par le programmeur pour assurer la cohérence, une information sur les accès temporels peut y être associée. Cette information représente le *taux de réutilisation*, permettant de mieux capturer le potentiel de rétention d'un

bloc dans cache logiciel. Ce renseignement est couplé à un suivi du nombre d'accès au blocs afin de faire décroître le potentiel au fur et a mesure de son utilisation. Ainsi les données à fort taux de réutilisation sont conservées en priorité. Les indices de réutilisation peuvent soit être fournis par le programmeur au moment où les blocs de données sont décrits, ou bien peuvent être collectés au cours d'une phase d'initialisation où les accès sont décomptés pour chaque bloc et mémorisés pour une exécution ultérieure. Le potentiel de réutilisation d'un bloc est ainsi défini par la somme des accès potentiels à ce bloc durant toute la durée de vie du programme de calcul (cf. **formule 5.1**).

$$\boxed{P(bloc) \simeq \sum H_{pot}(bloc)} \tag{5.1}$$

P : potentiel de réutilisation
H_{pot} : accès potentiel (*hits*)

A partir de cette formule, un indice d'intensité de rétention est calculé à chaque fois qu'une donnée est accédée localement. Il sert à trier les données en mémoire cache pour faciliter l'évacuation des données en cas de manque d'espace mémoire. Cette métrique tient compte des accès déjà effectués au bloc de données et au potentiel de réutilisation global. Ainsi, plus un bloc est accédé au cours de l'exécution, plus sont potentiel de réutilisation effectif doit décroître afin de réduire une éventuelle pollution du cache, tel que le ferait une politique LFU. L'indice d'éviction intègre également une information permettant de tenir compte de l'ancienneté du dernier accès au bloc depuis le cache logiciel. Ainsi lorsque plusieurs blocs possèdent un même potentiel de réutilisation à un instant *t*, le bloc évincé est le plus ancien à avoir servi. Ainsi, si aucun potentiel de réutilisation n'est renseigné, les *victimes* sont choisies de la même manière qu'avec la politique LRU. La **figure 5.9** illustre le fonctionnement de ce cache à deux niveaux mêlant potentiel de réutilisation et fraîcheur d'accès. Les blocs sont triés par intensité totale de rétention. Ceux qui possèdent la plus faible valeur sont évincés en priorité. La **formule 5.2** explicite la façon dont l'indice d'intensité de rétention d'un bloc est calculé.

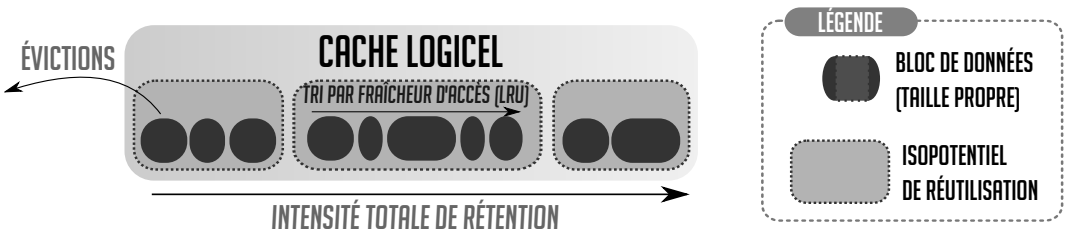


FIGURE 5.9 – Cache logiciel avec politique d'éviction basée sur le potentiel de réutilisation et l'ancienneté d'accès.

$$\boxed{I(bloc) = \left[\, P(bloc) - \sum H_{glob}(bloc) \,\right] \times \sum P + \sum H_{loc}(cache)} \tag{5.2}$$

I : indice d'intensité de rétention
P : potentiel de réutilisation
H_{glob} : accès, indépendamment de l'unité d'exécution
$\sum P$: somme de tous les potentiel de réutilisation en considérant tous les blocs
H_{loc} : compteur d'accès au cache pour réactualiser la fraîcheur d'un bloc. Permet un comportement de type LRU.

La **figure 5.10a** révèlent les performances obtenues avec la politique d'éviction basée sur l'indice d'intensité de rétention à partir du code de factorisation LU de matrices denses présenté précédemment. Un ordonnancement aiguillé par la localité est utilisé. Lorsque le choix de l'indice est adéquat, une hausse jusqu'à environ 10 GFLOP/s est constatée par rapport à un cache logiciel LRU. La **figure 5.10b** démontre que cette hausse de performance est apporté par une économie de 10% des données transférées hors recouvrement. Les **figures 5.10c et 5.10d** révèlent que la politique d'éviction rend le cache moins sensible à l'ordre exécution des super-tâches. Cette propriété présente un intérêt lorsque des techniques d'équilibrage de charge dynamique sont employées et où l'ordre d'exécution des super-tâche n'est pas déterministe.

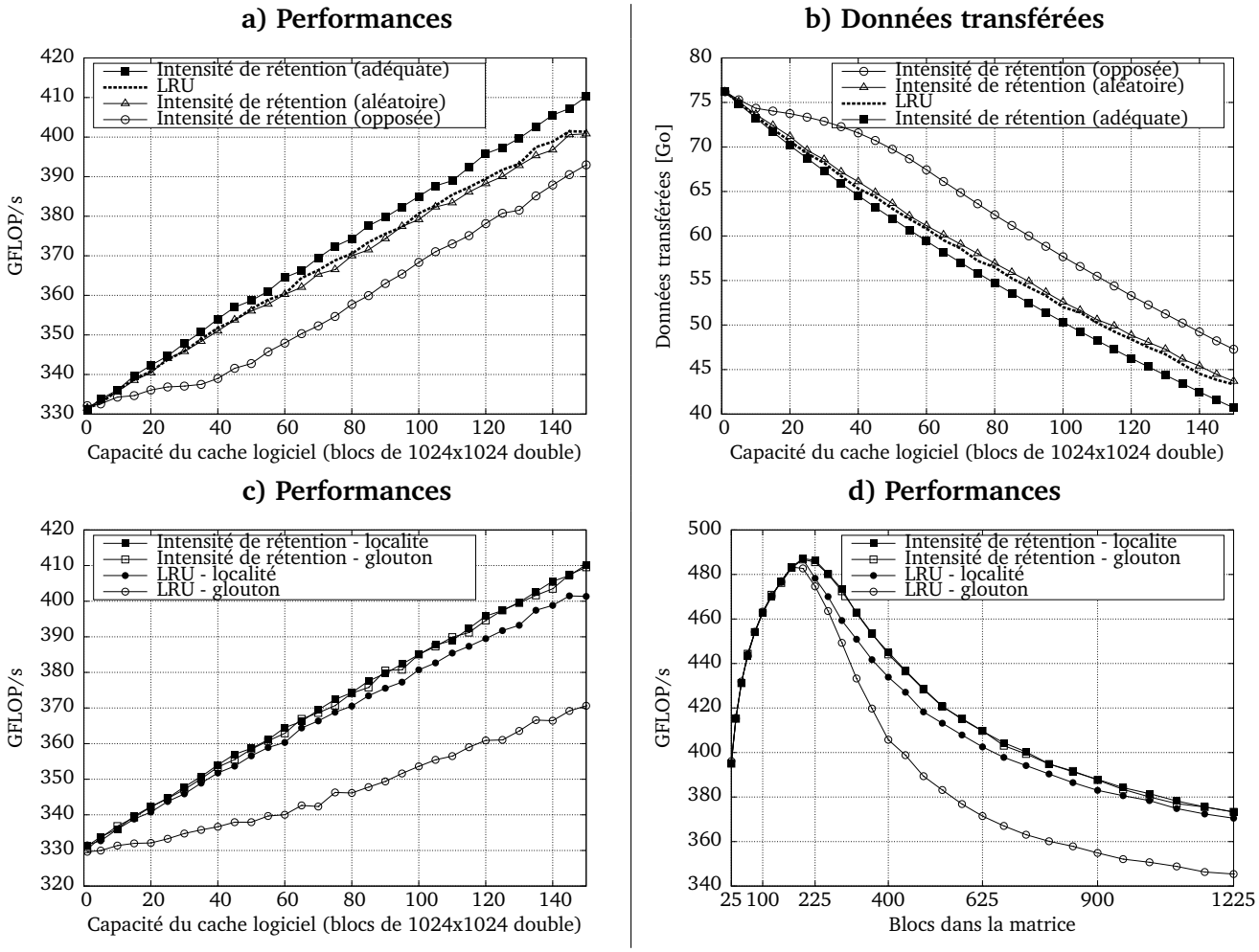


FIGURE 5.10 – Évaluation des performances et de la stabilité de la politique d'éviction basée sur l'indice d'intensité de rétention à partir de l'étape 3 d'une factorisation LU de matrices denses constituées de blocs de 1024×1024 valeurs à virgule flottante en double précision. Les opérations sont exécutée sur un accélérateur *Nvidia Tesla K20* (*Cirrus hétérogène*, cf. **annexe A.5**). **a-b)** Comparaison des performances et de la quantité de transferts selon la qualité de renseignement des potentiels de réutilisation (la matrice employée contient 25×25 blocs). **c-d)** Variation des performances selon le type d'ordonnancement utilisé.

Ainsi, cette politique d'éviction présente deux propriétés intéressantes. Premièrement, elle est utile lorsque des blocs de donnés possèdent des potentiels de réutilisation variables. Dans ce cas, les opportunités de rétention sont mieux capturées en possédant une vision globale de la quantité probable d'accès par bloc. Deuxièmement, le cache logiciel est moins sensible aux choix ordonnancement en évinçant toujours les blocs qui possèdent un faible indice d'intensité de rétention.

Si les potentiels ne sont pas renseignés, la politique d'éviction se comporte alors comme avec un cache LRU. Néanmoins, lorsqu'ils sont mal renseignés, les performances attendues peuvent être y inférieures. Cette fonctionnalité doit donc être renseignée par un programmeur averti ou déployée après une phase de profilage permettant de récupérer de manière plus adéquate les potentiels de réutilisation.

5.2.3 Affinités des tâches pour une isolation des données

Jusqu'ici les performances d'un cache logiciel n'ont été évaluées qu'à partir de l'exploitation d'une seule ressource de calcul déportée. Lorsque des unités hétérogènes à mémoire disjointes sont employées, la localité peut être bouleversée par les mécanismes de cohérences appliqués aux blocs de données. Il s'agit alors d'orienter les super-tâches pour accroître la localité. La technique présentée dans cette partie vise à aiguiller en priorité vers un accélérateur possédant un cache logiciel, les super-tâches qui touchent aux blocs à plus fort potentiel de réutilisation. Lorsque les dépendances de données le permettent, l'objectif est d'isoler au maximum les blocs à fort potentiel de réutilisation dans les cache logiciels des mémoires déportées.

La section 4.2 présentait l'association d'une liste partagée de super-tâches par couple hétérogène, composé d'une équipe associée à l'hôte et une autre équipe accélératrice. Cette liste de super-tâches peut être ordonnée par affinités, de telle sorte que les blocs à plus forts potentiels de réutilisation soient dirigés vers l'équipe accélératrice. La **figure 5.11** illustre ce mécanisme.

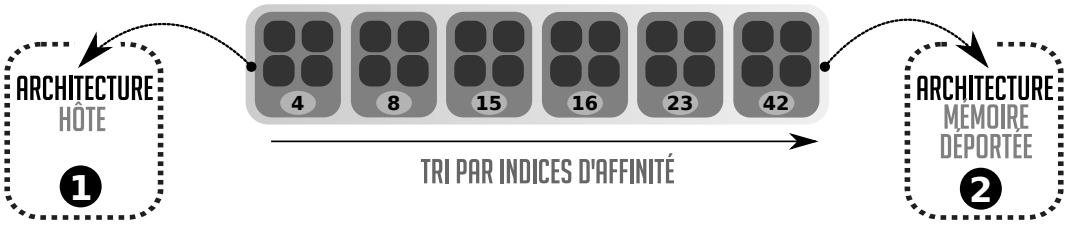


FIGURE 5.11 – Liste de super-tâches triée par affinité de localité.

L'indice permettant le tri de la liste de super-tâche est défini par la formule suivante :

$$A(supertache, liste) = \sum_{i=1}^{blocs} L(i, liste) \times I(i, cache) \tag{5.3}$$

A : affinité pour une équipe déportée
L : présence en mémoire de la ressource de calcul déportée (0 ou 1)
I : indice de rétention d'un bloc lié au cache logiciel considéré

Cette organisation peut mener à une meilleure coordination entre les équipes hétérogènes. L'équipe accélératrice déportée se focalise alors sur un jeu de données réduit puisque qu'une partie des traitements sont réalisés par la seconde équipe associée à l'hôte. L'efficacité du cache est alors amplifiée car moins d'évictions sont nécessaires. Les **figures 5.12a et 5.12b** confirment ce phénomène sur deux nœuds de calcul différents, à partir de la factorisation LU de matrices creuses présentée dans la section 3.4.2. Pour des matrices suffisamment grandes (plus de 12 000 blocs denses et creux), l'affinité des super-tâches en fonction de la localité et du potentiel de réutilisation permet de réduire la quantité de transferts. Des performances supérieures au cumul de celles obtenues par les exécutions homogènes (accélérateur seul et CPUs seuls) sont ainsi atteintes.

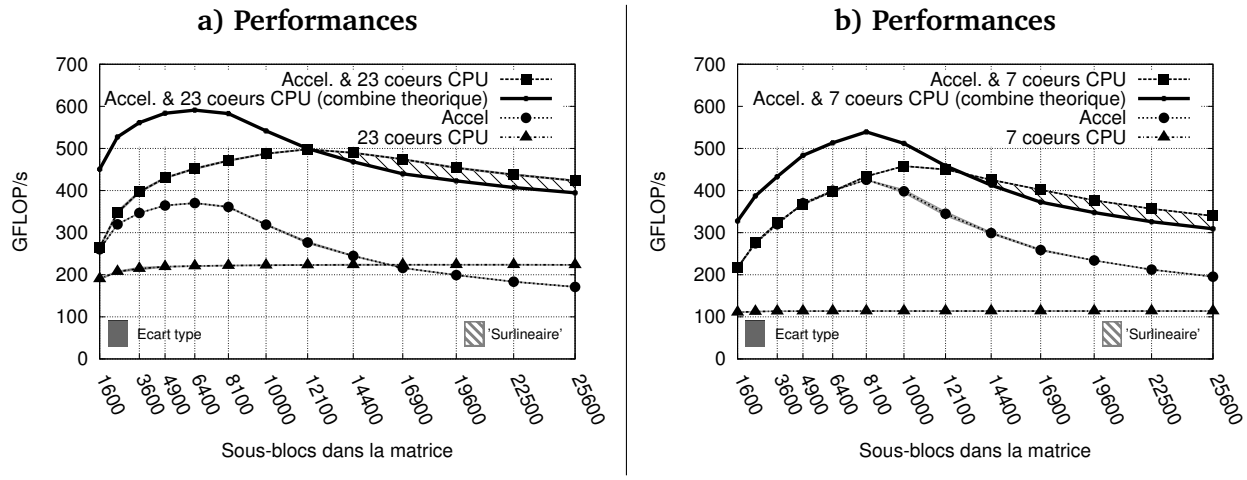


FIGURE 5.12 – Étape 3 de factorisations LU de matrices creuses avec des blocs constitués de 5×5 sous-blocs creux ou contenant 192×192 valeurs à virgule flottante en simple précision.Une séparation des super-tâches permettant d'atteindre des performances hétérogènes supérieures aux performances homogènes cumulées. **a)** Nœud de calcul *ENS-Fermi* (cf. **annexe A.6**) **b)** *Station hétérogène* (cf. **annexe A.7**)

5.3 Localité temporelle entre plusieurs appels de bibliothèques

Les bibliothèques hétérogènes permettent d'alléger la charge de travail du programmeur en fournissant des méthodes permettant d'exploiter conjointement plusieurs ressources de calcul. En mode homogène, les données sont placées dans un seule mémoire. Plusieurs appels consécutifs à la bibliothèque n'engendrent donc pas nécessairement de mouvements de données. Par exemple, la bibliothèque d'algèbre linéaire d'*Nvidia cuBLAS* charge le programmeur de transférer les données adéquates vers les accélérateurs. Dans le cadre d'une exécution hétérogène avec équilibrage de charge, la position des données n'est pas déterminée à l'avance. C'est pour cela que lorsque de telles bibliothèques hétérogènes sont employées dans un code de calcul, toutes les données sont préalablement rendues cohérentes en mémoire centrale. Les mouvements de données et les opérations sont ensuite pris en charge par la bibliothèque. À l'issu du calcul, les données modifiées sont à nouveau transférées en mémoire centrale, pouvant ainsi être exploitées par le reste du programme. Ce mode d'exécution hétérogène amplifie les communications et conduit à un nombre important de transferts de données.

MORSE est une interface permettant de lier facilement des plate-formes d'exécution à des bibliothèques d'algèbre linéaires comme MAGMA. Ce genre d'interface améliore la collaboration de groupes de travail intervenant sur l'ordonnancement hétérogène et les noyaux d'algèbre linéaire. Elle permet une utilisation plus flexible des plate-formes d'exécution pour l'élaboration de bibliothèques hétérogènes optimisées. Cette section explore la possibilité d'étendre une interface de type MORSE afin d'accroître la localité temporelle entre plusieurs appels de bibliothèque et d'autres portions déclarées à la main dans un code de simulation. Cela permet d'employer d'une part la même plate-forme d'exécution à la fois pour la bibliothèque hétérogène et pour le code défini par le programmeur. D'autre part, les bénéfices des caches logiciels sont conservés entre les différents appels de noyaux optimisés.

5.3.1 Extension de la gestion des données

Afin de maintenir la localité des caches logiciels, il est nécessaire que les bibliothèques et le code de calcul travaillent de concert. Pour cela, ils doivent partager des caractéristiques communes, de manière transparente pour le programmeur. En particulier, la granularité de la cohérence définie dans un code de calcul doit être récupérée et exploitée par la bibliothèque appelée. Ainsi, les transferts dus au maintient de la cohérence ne s'opèrent que lorsque c'est nécessaire, en fonction de la position de chaque bloc.

Un module appelé résolveur permet, à partir de plages d'adresse, de retrouver les caractéristiques de l'allocation associée et de la granularité de la cohérence des données. Des opérations peuvent ensuite s'adapter à cette répartition et commander directement des mouvements de blocs. Ce module vient s'intercaler entre l'allocation étendues (cf. chapitre 4.3.4, page 93) et le module de gestion de la granularité. La figure 5.13 schématise l'environnement de gestion des données ainsi complété.

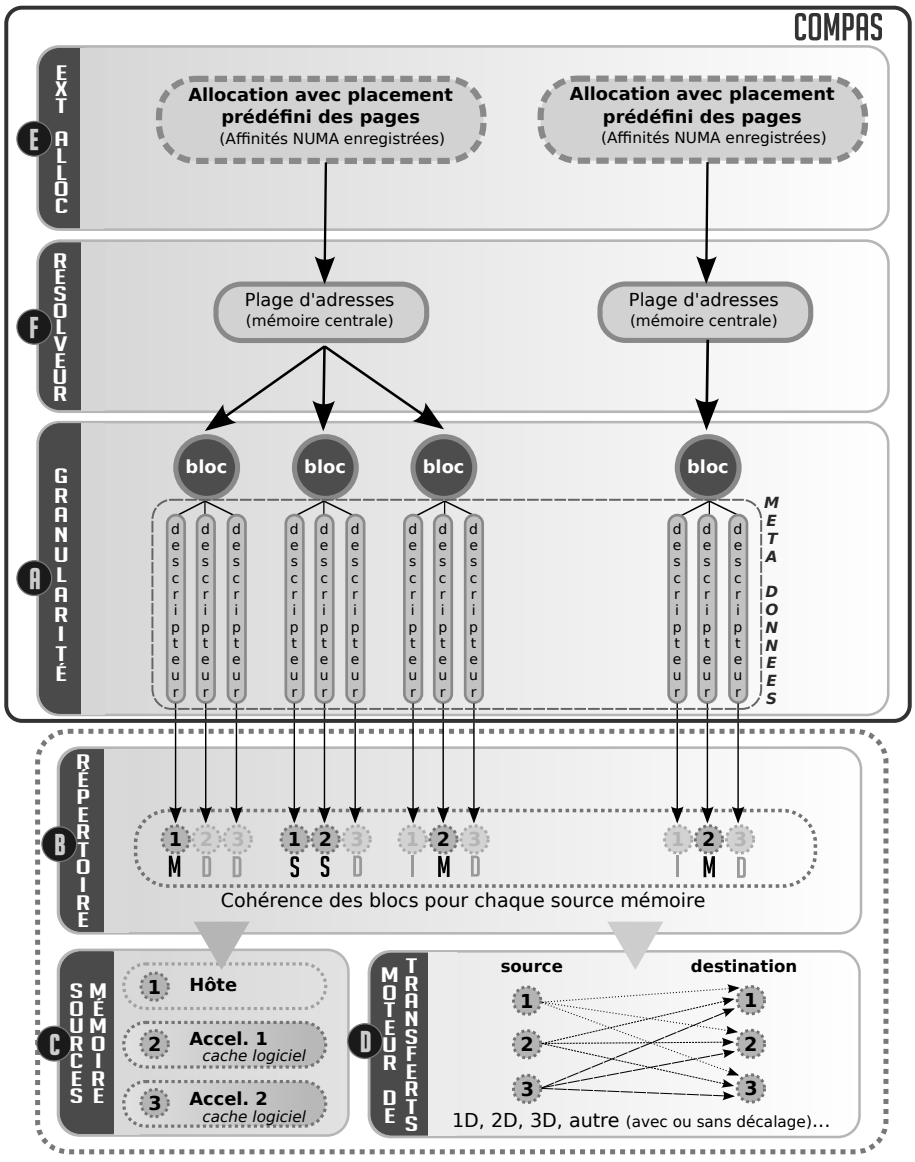


FIGURE 5.13 – Mécanismes permettant une gestion automatisée des données, avec allocations étendues et résolveur.

5.3.2 Évaluation des performances

Les performances d'appels successifs de la même bibliothèques sont étudiées dans cette partie. Des multiplications de matrices contenant des valeurs à virgule flottante (simple précision) sont choisies. La figure 5.14 présente le code renseigné par le programmeur. Les matrices sont déclarées avec ou sans répartition des pages mémoires. La granularité de la décomposition est par défaut associée au schéma de répartition des pages. Ainsi, dans ce cas précis, des blocs de cohérence de 1024×1024 valeurs sont initialisés à l'allocation. Cinq multiplications de matrices s'enchaînent en cumulant les résultats dans la matrice C. La bibliothèque hétérogène récupère les informations nécessaires pour exploiter la même granularité de cohérence que celle définie à l'allocation. Les données sont alors maintenues dans les caches logiciels de manière transparente pour le programmeur.

Code : appel de bibliothèque hétérogène à localité temporelle

```
1  int M = 2048;
2  int N = 2048;
3  int K = 2048;
4
5  /* Declaration des caracteristiques du placement des pages */
6  struct compas_pages_s pages;
7  compas_pages_init(&pages);
8  pages.geometry = COMPAS_2D;
9  pages.pattern = COMPAS_CYCLIC;
10 pages.xblocks = sizeof(float)*K/4096;
11 pages.yblocks = sizeof(float)*M/4096;
12 pages.xblocksize = (M*sizeof(float))/PSIZE;
13 pages.yblocksize = (M*sizeof(float))/PSIZE;
14 pages.elsize = sizeof(float);
15
16 /* Allocation des matrices et declaration des  */
17 /* blocs de coherence implicite a l'allocation. */
18 float *A = compas_malloc(&pages, COMPAS_PLOCKED, M*K*sizeof(float));
19 float *B = compas_malloc(&pages, COMPAS_PLOCKED, K*N*sizeof(float));
20 float *C = compas_malloc(&pages, COMPAS_PLOCKED, M*N*sizeof(float));
21
22 /* Appel successif classic de 5 multiplications de matrices */
23 for (int i=0; i<5; i++) {
24   sgemm(CblasColMajor,CblasNoTrans,CblasNoTrans, M,N,K, 1, A,M, B,K, 1, C,M);
25 }
```

FIGURE 5.14 – Appel de 5 multiplications de matrices interagissant avec la plate-forme pour le maintien des données. Après la définition du placement des pages, une granularité de cohérence par défaut est choisie, laquelle est récupérée par la bibliothèque de manière transparente pour l'utilisateur.

La figure 5.15 présente les performances obtenues avec et sans maintien des données dans les caches logiciels. L'interface MORSE est modifiée pour que cette rétention soit possible avec le noyau *SGEMM* de la bibliothèque MAGMA, dont les tâches de calcul sont prises en charge par StarPU. Les résultats sont comparés à un ordonnancement de super-tâches associées à des blocs de données pouvant être décomposés en sous-blocs de 256×256 pour les unités d'une même équipe. A partir d'un nœud de calcul de la partition *Inti hétérogène* (cf. annexe A.1), la rétention de données permet un accroissement de performances pouvant atteindre 66% avec la bibliothèque MAGMA modifiée. L'ordonnancement à multi-granularité apporte au gain supplémentaire qui

peut être amplifié par un placement initial des pages mémoires afin de mieux guider la localité spatiale. Ainsi, 1 139 GFLOP/s sont atteints, portant l'amélioration à 87% face à la bibliothèque MAGMA d'origine, sans la rétention des blocs de données.

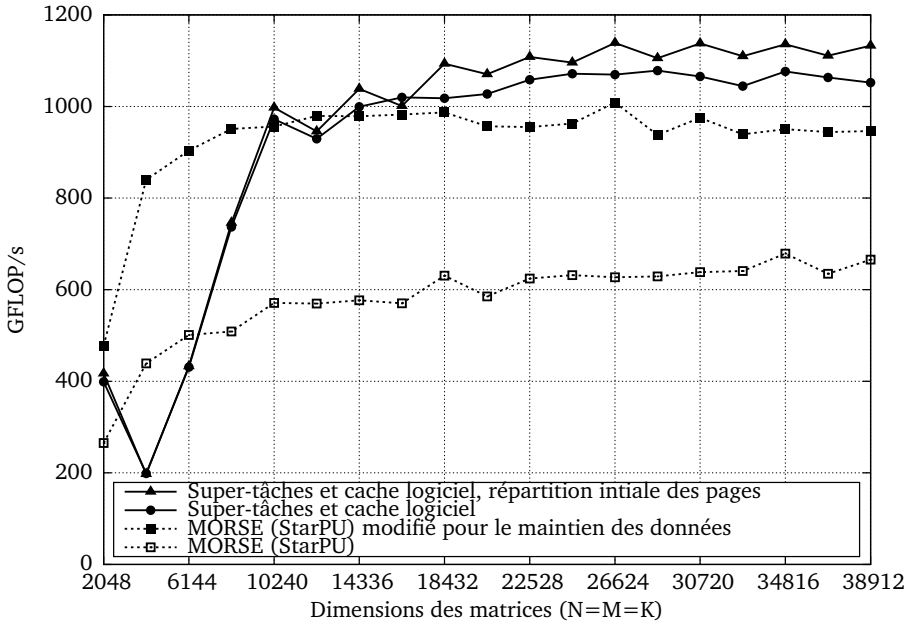


FIGURE 5.15 – *SGEMM* hétérogènes avec deux processeurs Intel Xeon et deux accélérateurs Nvidia M2050 (*Inti hétérogène*, cf. **annexe A.1**), décomposition en blocs de 1024×1024 valeurs et réutilisation des données lors d'un déchaînement de 5 multiplications de grandes matrices.

Pour résumer, cette section a présenté divers mécanismes permettant de gérer des données dans un contexte de programmation par tâches de calcul. Des transferts sont déclenchés sans l'intervention du programmeur afin de conserver des données à jour en présence de mémoires qui ne sont pas rendues cohérentes entre elles par le matériel. Des caches logiciels permettent de réduire la quantité de données véhiculées entre les différentes ces mémoires. La rétention est amplifiée en enrichissant une politique d'éviction *LRU* avec un indice facultatif permettant, dans certains codes de calcul, de mieux capturer les potentiels de réutilisation. Un couplage fort est déployé entre l'état des caches logiciels et la politique d'ordonnancement afin d'accroître davantage l'efficacité de rétention. Enfin, une proposition d'association entre une plate-forme d'exécution et des bibliothèques hétérogènes a été décrite afin de conserver plus de données dans les caches logiciels.

Chapitre 6

Ordonnancement gouverné par la localité des données

> *“ Quand il n’obéit pas au gouvernail, le navire obéit à l’écueil. ”*
>
> Anatole France, *L’Île des Pingouins*, 1908

Dans les premiers chapitres, des contraintes liées à la gestion des données ont été observées. Au travers des contributions détaillées dans les deux chapitres précédents, des mécanismes ont été proposés pour accroître la localité des données. Il s’agit désormais de les combiner dans une unique solution. Nous avons ainsi élaboré la plate-forme **H3LMS**[110]. Celle-ci se prononce “*helms*” comme le terme anglais qui désigne les gouvernails. Ce support d’exécution gouverne l’attribution de chaque super-tâche. Il évite autant que faire se peut l’écueil des accès distants en s’appuyant sur le module *COMPAS* présenté dans le **chapitre 4** (page 79).

H3LMS exploite un modèle de programmation par tâches (et super-tâches) de calcul en adoptant une approche de gestion synchrone par lots[111] telle que l’a introduit L. Valiant pour son modèle BSP[112] [238, 239]. Des décisions d’ordonnancement peuvent alors être prises à partir de ces lots de tâches. L’avantage est de détenir plus d’informations sur la position des données associées à une grande quantité de tâches exécutables. Des affinités plus poussées peuvent être calculées pour chaque unité logique afin de réduire les accès distants tout en facilitant des techniques d’équilibrage de charge hétérogènes. Un mauvais choix d’ordonnancement peut entraîner des déplacements de données coûteux, que ce soit depuis des mémoires d’un nœud NUMA distant, ou plus pénalisant, depuis des mémoires déportées, embarquées dans des accélérateurs matériels. Des heuristiques permettent une répartition des super-tâches qui priorise ces accès locaux. De plus, une collaboration entre des unités logiques favorise l’exploitation de la hiérarchie mémoire des nœuds de calcul. Les mécanismes de partage et de vol de tâches qui permettent un travail collaboratif des équipes logiques sont abordés dans ce chapitre. Le programmeur peut être chargé de fournir des informations facultatives et permettre de mieux orienter la plate-forme sur l’agressivité à choisir quant à l’équilibrage de charge.

Les codes de calcul parallèles existants reposent souvent sur des modèles de programmation de type MPI, OpenMP ou la combinaison des deux. La programmation par tâche de calcul hétérogène peut quant à elle faciliter l’exploitation de matériel spécialisé. Employer ce modèle ponctuellement dans un code de calcul existant contribuerait à faciliter un portage progressif. Dans ce cas précis, il faut favoriser une cohabitation adaptée entre tous les supports exécutifs des divers

[110] **H3LMS :** *Harnessing Hierarchy and Heterogeneity with Locality Management and Scheduling*.

[111] *Bulk synchronous*.

[112] **BSP :** *Bulk-Synchronous Parallel*.

modèles de programmation employés. En effet, une compétition entre les différentes plate-formes peut induire des ralentissements lors de l'exécution. Une intégration dans un contexte fusionné peut concourir à améliorer leur fonctionnement en réduisant ces surcoûts. L'environnement de développement *MPC*[113] (cf. chapitre 3.3.1, page 66), développé conjointement par le CEA et le laboratoire *Exascale Computing Research*, permet cela en s'appuyant sur un ordonnancement unifié.

Ce chapitre présente tout d'abord l'interface et les heuristiques adoptées pour ordonnancer des tâches et super-tâches de calcul. Les mécanismes d'équilibrage de charge hiérarchiques permettant de réduire les accès distants aux données sont ensuite décrits. Un second mode d'équilibrage est détaillé pour les tâches dont les performances sont fortement conditionnées par les mouvements de données. Enfin, des détails sont apportés sur l'intégration de la plate-forme *H3LMS* dans l'environnement multi-modèle de programmation *MPC*.

6.1 Interface de programmation

Factorisation LU d'une matrice dense de 128×128 blocs de 2048×2048 éléments

```
1  #define NB 128 /* Nombre de blocs par dimension */
2  #define BS 2048 /* Nombre d'element par dimension dans un bloc */
3  /* Donnees physiquement rangees par colonnes */
4  #define MROW BS
5  #define BROW (BS*BS*NB)
6
7  int main(int argc, char* argv[]) {
8    /* Allocation et initialisation */
9    double *A = malloc(NB*BS*NB*BS*sizeof(double));
10   init_matrix(A);
11
12   for (int kk=0; kk<NB; kk++) {
13     /* Etape 1 */
14     lu0( A[kk*MROW + kk*BS] );
15
16     /* Etape 2 */
17     for (int jj=kk+1; jj<NB; jj++) {
18       fwd (A[kk*MROW + kk*BS], A[kk*MROW + jj*BS]);
19     }
20
21     for (int ii=kk+1; ii<NB; ii++)  {
22       bdiv (A[kk*MROW + kk*BS], A[ii*MROW + kk*BS]);
23     }
24
25     /* Etape 3 */
26     for (int ii=kk+1; ii<NB; ii++) {
27       for (int jj=kk+1; jj<NB; jj++) {
28         bmod (A[ii*MROW + kk*BS], A[kk*MROW + jj*BS], A[ii*MROW + jj*BS]);
29       }
30     }
31   }
32 }
```

FIGURE 6.1 – Factorisation LU sans pivot de matrice dense par blocs. Version séquentielle.

L'interface de programmation (API[114]) des principales fonctionnalités est présentée dans cette section afin de faciliter la compréhension des mécanismes d'ordonnancement détaillés

[113] **MPC :** *Multi-Processor Computing*.

[114] **API :** *Application Programming Interface*.

plus loin dans le chapitre. La factorisation LU de matrice dense par blocs introduite dans le chapitre 3.4.1 (page 69) est étudiée en guise d'exemple. En partant d'une implémentation séquentielle dont le code source est exposé en figure 6.1, cette section détaille les modifications utiles et nécessaires pour profiter du parallélisme de tâche aiguillé par la localité des données. Dans un premier temps, les données sont allouées et réparties dans les mémoires des nœuds NUMA à partir d'un schéma prédéfini grâce à *COMPAS*. La granularité de cohérence est ensuite renseignée au moyen de ce même module. Enfin, les déclarations des tâches et super-tâches sont présentées et les barrières de synchronisation correctement positionnées avec l'interface d'*H3LMS*. Des directives sont également proposées. Elles n'ont pas été implémentées, mais participeraient à accroître la productivité des portages de codes.

6.1.1 Affinités et compléments d'information sur les données (COMPAS)

Lorsqu'il n'y a qu'un seul producteur de tâches, spécifier la répartition des pages mémoires permet de réduire les accès distants en évitant de concentrer la pression sur la mémoire attachée à un unique nœud NUMA. D'autres informations permettent de définir la granularité à laquelle la cohérence des données sera gérée tout au long de la durée de vie de l'application et d'optimiser la quantité de transferts générée. Enfin, les indices de potentiels de réutilisation permettent de mieux orienter les super-tâches pour améliorer la rétention des données relatives dans les caches logiciels. Toutes ces informations sont continuellement analysées par l'ordonnanceur lors de l'attribution des super-tâches afin de réduire les effets NUMA et NUIOA.

Répartition des pages mémoires

Cette étape, bien que facultative, est vivement recommandée. La répartition des données prédétermine à la fois la pression sur les mémoires physiques des nœuds NUMA et le comportement de l'ordonnanceur. Un schéma de distribution des pages mémoires peut être spécifié avant de procéder à l'allocation et au placement effectif de ces pages. La figure 6.2 illustre une répartition par blocs de pages. Chaque bloc est placé dans la mémoire d'un nœud NUMA de manière alternée grâce au mot clé "*cyclic*".

Interface de programmation (COMPAS)

```
8   /* Allocation d'une matrice et
       repartition des pages */
9   struct compas_pages_s pages;
10  compas_pages_init(&pages);
11  pages.geometry = COMPAS_2D;
12  pages.pattern = COMPAS_CYCLIC;
13  pages.xsize = NB*BS;
14  pages.ysize = NB*BS;
15  pages.xblocksize = BS;
16  pages.yblocksize =
        (BS*sizeof(double))/PSIZE;
17  distri.elsize =
        (BS*sizeof(double))/PSIZE;
18
19  double *A = compas_malloc(&distri,
        COMPAS_PLOCKED, NB*BS*NB*BS
        *sizeof(double));
20  init_matrix(A);
```

Proposition de directive

```
8   /* Allocation d'une matrice et
       repartition des pages */
9   #pragma compas malloc
        elsize(sizeof(double))
        size(NB*BS,NB*BS)
        bsize((BS*sizeof(double))/PSIZE,
        (BS*sizeof(double))/PSIZE)
        pattern(cyclic)
        page(locked)
10
11  double *A = malloc(NB*BS*NB*BS *
        sizeof(double));
12  init_matrix(A);
```

FIGURE 6.2 – Allocation d'une matrice en déterminant le schéma de répartition des pages mémoires associées.

Description des blocs de cohérence

Cette étape est indispensable, car elle permet à la plate-forme d'assurer la cohérence des données véhiculées entre les mémoires déportées et la mémoire centrale. La granularité des blocs de cohérence doit être renseignée. Pour cela, deux possibilités peuvent être envisagées. La figure 6.3 présente la première qui consiste à baser les blocs de cohérence sur les dimensions des blocs de répartition des page mémoires, spécifiées à l'étape précédente.

Interface de programmation (COMPAS)

```
22  /* creation des metadonnees selon la
       repartition des pages */
23  compas_data_register(A, NULL,
       COMPAS_COLMAJOR);
```

Proposition de directive

```
22  /* creation des metadonnees selon la
       repartition des pages */
23  #pragma compas register(A)
       order(colmajor)
```

FIGURE 6.3 – Création des blocs de cohérence en utilisant la même granularité que la répartition des pages mémoire.

La deuxième solution nécessite de renseigner explicitement les caractéristiques de ces blocs de cohérence. Il est aussi possible de préciser un décalage sur une dimension si la granularité des blocs ne permet pas de les aligner correctement sur les blocs de répartition de pages. La figure 6.4 illustre cette façon de procéder en spécifiant un décalage de 1024 éléments. La première solution est retenue pour la suite de cet exemple.

Interface de programmation (COMPAS)

```
22  /* creation des metadonnees selon le
       schema suivant */
23  compas_data_s data;
24  compas_data_init(&data);
25  data.geometry = 2D;
26  data.xsize = NB*BS;
27  data.ysize = NB*BS;
28  data.xblocksize = BS;
29  data.yblocksize = BS;
30  data.ystride = 1024;
31  data.elsize = sizeof(double);
32
33  compas_data_register(A, &data,
       COMPAS_COLMAJOR);
```

Proposition de directive

```
22  /* creation des metadonnees selon le
       schema suivant */
23  #pragma compas register(A)
       elsize(sizeof(double))
       size(NB*BS-BS,NB*BS-BS)
       bsize(BS,BS) stride(1024)
       order(colmajor)
```

FIGURE 6.4 – Création des blocs de cohérence en spécifiant explicitement un schéma.

Potentiel de réutilisation

L'algorithme de factorisation LU est itératif et progresse le long de la diagonale principale. Les blocs situés en haut à gauche sont moins réutilisés que ceux en bas à droite au cours de la factorisation. La figure 6.5 précise la façon de renseigner l'indice de potentiel de réutilisation pour chaque bloc de cohérence. Le *résolveur* (cf. chapitre 5.3.1, page 110) retrouve les structures des méta-données à partir d'une adresse mémoire pointant vers un élément contenu dans un bloc. Il suffit alors de renseigner une telle adresse avec la valeur de l'indice pour associer ce dernier à un bloc de cohérence. Une autre manière plus concise pourrait être employée en fournissant une formule dans une directive. Elle permettrait de calculer l'indice de réutilisation en fonction des coordonnées i et j de chaque bloc (cf. figure 6.5).

Interface de programmation (COMPAS)

```
25    /* potentiels de reutilisation */
26    for(int jp=0; jp<NB; jp++)
27      for(int ip=0; ip<NB; ip++)
28        compas_reuse_potential(&A[ip*MROW
              + jp*BS], min(jp,ip));
```

Proposition de directive

```
25    /* potentiels de reutilisation */
26    #pragma compas reuse(A,min(i,j))
```

FIGURE 6.5 – Potentiel de réutilisation pour améliorer les performances des caches logiciels.

6.1.2 Sémantiques associées aux tâches et super-tâches de calcul (H3LMS)

L'interface de création des tâches et des super-tâches est introduite dans cette sous section en s'appuyant sur l'exemple de factorisation LU de matrice dense.

Définition d'une tâche de calcul

Nous décidons d'orienter toutes les opérations *fwd* (étape 2 de la factorisation LU) sur les unités logiques CPUs. Les super-tâches ne sont donc pas nécessaires puisqu'un seul niveau de granularité est choisi. Il s'agit du mode de création classique de tout modèle par tâches de calcul. La figure 6.6 présente l'interface permettant de définir de telles tâches de calcul. Des *macros C* permettent d'alléger la façon de définir ces tâches. En particulier, une fonction interne est générée et s'assure de maintenir la cohérence des données selon les types de dépendances renseignés en déclenchant les transferts de données appropriés et de manière transparente pour le développeur. Cette écriture peut toutefois rendre plus compliquées les phases de débogage en cas d'erreur. Un jeu de *macros variadiques* permet d'allouer une structure de taille adéquate pour porter les informations liées aux tâches et les valeurs des arguments de la fonction ciblée. Ces allocations systématiques rendent la génération des tâches nettement moins légère que celle opérée par XKaapi par exemple (cf. chapitre 3.3.2, page 66). Si ces tâches embarquent suffisamment de traitements, le surcoût de création devient naturellement négligeable. Une barrière de synchronisation permet d'attendre que toutes les tâches générées aient été exécutées avant de passer à l'étape suivante.

Interface de programmation (H3LMS)

```
21    /* Etape 2 */
22    for (int jj=kk+1; jj<NB; jj++) {
23      H3LMS_BEGIN_TASK_DECLARATION
24        H3LMS_REGISTER_NB_DEP(2)
25        H3LMS_REGISTER_DEP_2D(
              &A[kk*MROW+kk*BS],
              H3LMS_IN, sizeof(double),
              BS,BS*NB,BS)
26        H3LMS_REGISTER_DEP_2D(
              &A[kk*MROW+jj*BS],
              H3LMS_INOUT,
              sizeof(double),
              BS,BS*NB,BS)
27        H3LMS_CPU_FUNCTION(fwd)
28      H3LMS_END_TASK_DECLARATION
29    }
30    ...
31    H3LMS_BARRIER
```

Proposition de directive

```
21    /* Etape 2 */
22    for (int jj=kk+1; jj<NB; jj++) {
23      #pragma h3lms task depend(in:
            A[kk*BS:kk*BS+BS-1],
            kk*BS:kk*BS+BS-1], inout:
            A[kk*BS:kk*BS+BS-1,
            jj*BS:jj*BS+BS-1])
24        fwd (A[kk*MROW + kk*BS], A[kk*
              MROW + jj*BS]);
25    }
26    ...
27    #pragma h3lms barrier
```

FIGURE 6.6 – Définition de tâches CPUs pour la fonction *fwd* de l'étape 2.

Définition d'une super-tâche de calcul

Interface de programmation (H3LMS)

```
H3LMS_BEGIN_SUPERTASK_FUNCTION(l_bmod_cpu)
  double *A1 = GETDATA(0, double*);
  double *A2 = GETDATA(1, double*);
  double *A3 = GETDATA(2, double*);

  const int subblocks = 16;
  const int subblocksize = BS/subblocks;

   for (int i=0; i<subblocks; i++) {
     double *SA1 = A1;
     double *SA2 = &A2[i*subblocksize
         *BS*NB];
     double *SA3 = &A3[i*subblocksize
         *BS*NB];
     H3LMS_BEGIN_SUBTASK_DECLARATION
        H3LMS_REGISTER_ARGS(SA1, SA2, SA3,
            subblocksize)
        H3LMS_CPU_LAUCHER_FUNCTION(
            bmod_cpu)
     H3LMS_END_SUBTASK_DECLARATION
   }
H3LMS_END_SUPERTASK_FUNCTION
...
/* Etape 3 */
for (int ii=kk+1; ii<NB; ii++) {
  for (int jj=kk+1; jj<NB; jj++) {
    H3LMS_BEGIN_SUPERTASK_DECLARATION
       H3LMS_REGISTER_NB_DEP(3)
       H3LMS_REGISTER_DEP_2D( &A[ii*MROW +
           kk*BS], IN,sizeof(double),
           BS,BS*NB,BS)
       H3LMS_REGISTER_DEP_2D( &A[kk*MROW +
           jj*BS], IN,sizeof(double),
           BS,BS*NB,BS)
       H3LMS_REGISTER_DEP_2D( &A[ii*MROW +
           jj*BS], INOUT,sizeof(double),
           BS,BS*NB,BS)
       H3LMS_CPU_LAUCHER_FUNCTION(
           l_bmod_cpu)
       H3LMS_ACCEL_LAUCHER_FUNCTION(
           l_bmod_accel)
       H3LMS_COMPUTE_RATIO(5)
    H3LMS_END_SUPERTASK_DECLARATION
  }
}

H3LMS_BARRIER
```

Proposition de directive

```
/* Etape 3 */
for (int ii=kk+1; ii<NB; ii++) {
  for (int jj=kk+1; jj<NB; jj++)
      {
    #pragma h3lms supertask
        depend(in:
        A[ii*BS:ii*BS+BS-1,
        kk*BS:kk*BS+BS-1],
        A[kk*BS:kk*BS+BS-1,
        jj*BS:jj*BS+BS-1], inout:
        A[ii*BS:ii*BS+BS-1,
        jj*BS:jj*BS+BS-1])
        accel(l_bmod_cuda)
        cratio(5)
     {
      const int subblocks = 16;
      const int subblocksize = BS
          /subblocks;

      for (int i=0; i<subblocks;
          i++) {
        double *SA1 = &A[ii*MROW
            + kk*BS];
        double *SA2 = &A[kk*MROW
            + jj*BS + i*
            subblocksize*BS*NB];
        double *SA3 = &A[ii*MROW
            + jj*BS + i*
            subblocksize*BS*NB];

        #pragma h3lms subtask
        bmod_cpu (SA1, SA2, SA3,
            subblocksize);
      }
    }
  }
}

#pragma h3lms barrier
```

FIGURE 6.7 – Définition de super-tâches pour la fonction *bmod* de l'étape 3.

Les super-tâches sont renseignées pour qu'un noyau de calcul puisse profiter d'une exécution hétérogène. Les spécialisations tardives des super-tâches permettent un équilibrage de charge entre les différentes microarchitectures. La cohérence des données est alors opérée lorsqu'une super-tâche est exécutée. Si elle est prise en charge par une unité logique associée à un cœur de CPU, elle est préférentiellement décomposée en plusieurs sous-tâches. La cohérence de chacune de ces sous-tâches n'est pas vérifiée, car elle a déjà été opérée lors de l'exécution de la super-tâche.

Dans le cas de la fonction *bmod*, chaque super-tâche est scindée en 16 sous-tâches comme le décrit le code de la **figure 6.7**. L'indice de ratio de calcul (*compute ratio*) permet d'indiquer le rapport approximatif des performances attendues entre une unité logique accélératrice et une équipe logique associée à plusieurs cœurs CPU. Dans certains cas, cet indice pourrait être inféré au cours de l'exécution. Il peut aider à mieux guider les super-tâches vers les unités de traitement logique les plus appropriées. Plus de détails seront apportés à ce sujet dans la section suivante. L'attribut de dépendance du *pragma* proposé pour la création de super-tâches est inspiré de la version temporaire (en cours d'élaboration) de la norme *OpenMP 4.0* [240]. L'expression des sous-portions de matrices ([el. de début, el. de fin en x : el. de début, el. de fin en y]) repose quant à elle sur la notation *Matlab* [241].

6.2 Affectations gouvernées par la localité des données

Lorsqu'une super-tâche est générée, il faut la rendre accessible aux unités logiques d'exécution. Cette section décrit deux modes d'affectation possibles. Le premier est orienté par la localité des données tout en permettant un équilibrage de charge hétérogène. Le second mode se focalise uniquement sur la minimisation des déplacements de données, en sacrifiant des opportunités d'équilibrage.

Les heuristiques d'attributions des super-tâches se basent sur la représentation abstraite établie automatiquement à l'initialisation de la plate-forme *H3LMS*. La construction de ce modèle abstrait dépend de la machine cible et s'appuie sur les contraintes fixées dans le **chapitre 4.2** (page 87). Cette représentation détermine le nombre de pôles, les équipes qui les composent et la disposition des listes partagées. Elle établit également la quantité de caches logiciels en dénombrant les accélérateurs embarquant leur propre mémoire. La quantité de super-tâches générées peut être impactée à la fois par la quantité de données que doit traiter le code de calcul et par la granularité des données spécifiée au module *COMPAS*.

6.2.1 Affinités standards des super-tâches

Pour ce premier mode d'affectation, chaque super-tâche est placée dans une liste partagée. Cette dernière est principalement sélectionnée en fonction de la localité des données. Ce choix s'opère grossièrement comme une règle de compilation qui privilégie le traitement localisé des données[115] [242, 243], en tenant compte également des données nécessaires en lectures seules. Ensuite, une affinité associée aux architectures accédant à la liste permet de la trier pour accroître la localité des données entre les architectures hétérogènes et tenir compte des accélérations potentielles. Les formules permettant ces prises de décision sont présentées dans les sous-sections suivantes.

Pour rappel, l'architecture du nœud de calcul *ENS-Fermi* (cf. **annexe A.6**), est constituée de deux CPUs dodéca-cœurs aux effets NUMA internes et de deux GPUs connectés au même nœud NUIOA. La **figure 6.8**, introduite dans le **chapitre 4.2** (page 87), illustre l'organisation abstraite représentant cette machine. Les exemples d'attribution des tâches reposent sur cette architecture asymétrique, car elle permet d'aborder des cas de figure particuliers.

[115] *Owner computes rule* en anglais.

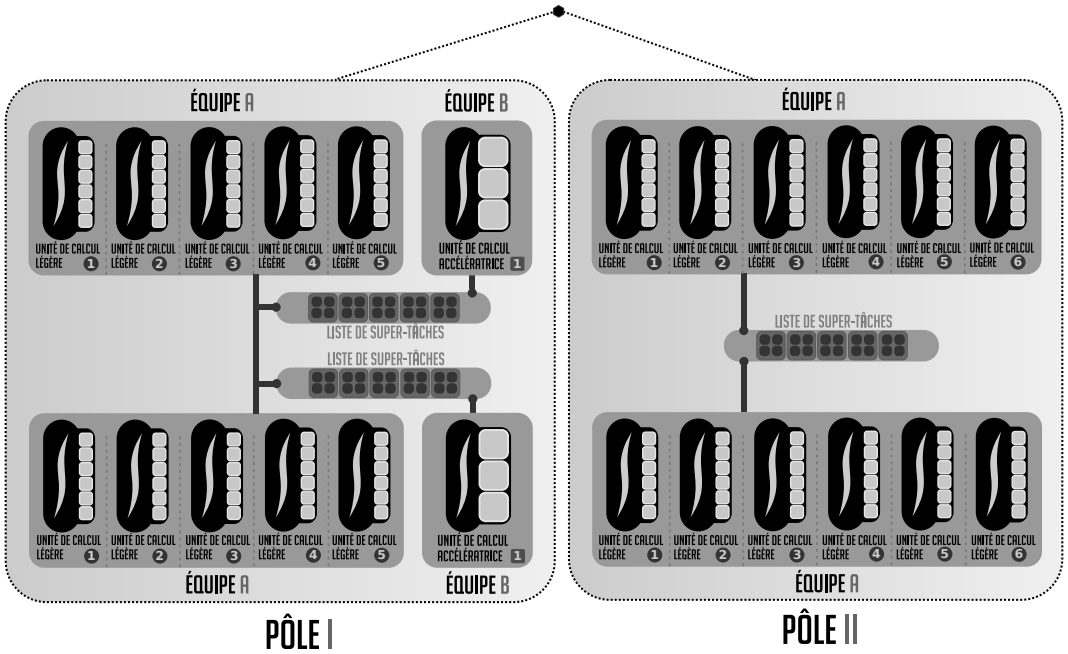


FIGURE 6.8 – Rappel de l'organisation abstraite d'un nœud de calcul asymétrique *ENS-Fermi* (cf. **annexe A.6**).

Choix du pôle et d'une liste de super-tâches

Les super-tâches contiennent idéalement une grande quantité de travail. Il vaut mieux choisir les unités d'exécution permettant de réduire la quantité d'accès distants, quitte à introduire un léger surcoût lié à des calculs d'affinités. Les mouvements de données vers les ressources de calcul accélératrices sont certainement les mécanismes les plus coûteux. Une importance particulière est portée sur des attributions de tâches réduisant ces transferts.

Lorsqu'une super-tâche est générée, deux cas de figure se posent en fonction de la présence d'au moins un de ses blocs de données dans un cache logiciel. Si c'est le cas, c'est l'indice R_1 qui est évalué. Il est calculé pour chaque cache logiciel associé à un accélérateur et détermine le matériel spécialisé qui possède la meilleure affinité en fonction des données chargées. Les blocs qui possèdent une dépendance de données en lecture et écriture (*inout*) ont plus de poids, car ils peuvent générer deux fois plus de trafic : un transfert au chargement et un autre pour rendre cohérentes les données modifiées en mémoire centrale. Ainsi, la liste de super-tâche associée à l'accélérateur qui possède l'indice le plus conséquent se voit affecter la super-tâche. En cas d'égalité, une liste est choisie au hasard. La formule suivante précise le calcul de R_1 pour un couple composé d'une super-tâche et d'un cache logiciel :

$$R_1(\text{super-tâche}, \text{cache}) = \sum_{i=1}^{B(\text{super-tâche})} S(i) \times D(i) \times L(i, \text{cache}) \tag{6.1}$$

R_1 : Indice de Répartition d'une super-tâche pour un cache logiciel
B : Ensemble des blocs rattachés à la super-tâche
S : Espace mémoire occupé par le bloc de données en octets
D : Type de dépendances de données (lecture : 1, écriture : 1, lecture-écriture : 2)
L : Présence en mémoire de la ressource de calcul déportée (0 ou 1)

En revanche, si aucun cache logiciel ne contient au moins un bloc de données, un autre indice R_2 est évalué. Celui-ci permet de diminuer les accès distants aux nœuds NUMA en identifiant un pôle adéquat. Le pôle est ainsi déterminé par cette autre formule et une liste de super-tâches est choisie au hasard dans celui-ci. La formule suivante permet d'évaluer R_2 :

$$R_2(\text{super-tâche}, \text{pôle}) = \sum_{i=1}^{B(\text{super-tâche})} S(i) \times D(i) \times N(i, \text{pôle}) \tag{6.2}$$

R_2 : Indice de Répartition d'une super-tâche pour une liste particulière
B : Ensemble des blocs rattachés à la super-tâche
S : Espace mémoire occupé par le bloc de données en octets
D : Type de dépendances de données (lecture : 1, écriture : 1, lecture-écriture : 2)
N : Présence en mémoire centrale dans le nœud NUMA associé (0 ou 1)

La figure 6.9 illustre ces choix d'attribution pour un nœud de calcul asymétrique de type *ENS-Fermi* (cf. annexe A.6) du fait de sa représentation abstraite. Une super-tâche associée à quatre blocs de données doit être placée dans une des trois listes partagées $LS1$, $LS2$ ou $LS3$. Si aucun des blocs n'est présent dans un des caches logiciels d'indice 1 ou 2, la formule R_2 est évaluée. Dans le cas contraire, c'est la formule R_1 qui permet de sélectionner une des listes *LS1* ou *LS2* qui améliore la réutilisation temporelle des données. La liste *LS3* n'est pas considérée par la formule R_1 puisqu'elle n'est associée à aucun accélérateur et donc aucun cache logiciel.

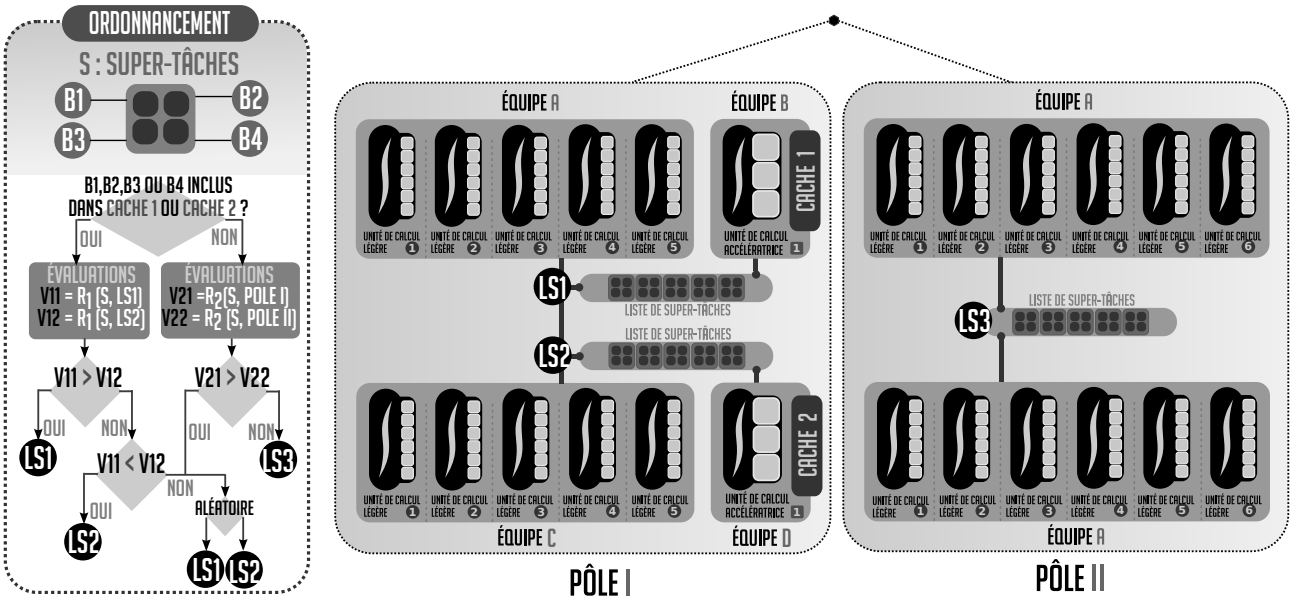


FIGURE 6.9 – Exemple d'arbre de décisions pour l'attribution d'une super-tâche en s'appuyant sur un nœud de calcul *ENS-Fermi* (cf. annexe A.6).

Calcul de l'indice d'affinité pour une unité accélératrice

Lorsqu'une liste est sélectionnée, les super-tâches sont triées par affinités à l'équipe accélératrice associée. Cela permet une meilleure isolation des données, comme décrit dans le chapitre 5.2.3 (page 108). La formule d'affinité est complétée par un facteur facultatif permettant d'apporter un poids d'importance à certaines super-tâches d'efficacité radicalement différente en fonction de l'architecture (*compute ratio*, cf. définition de l'interface en section 6.1.2, page 118). Les tâches dont les performances sont nettement plus avantageuses lorsqu'elles sont exécutées par une unité accélératrice sont attirées en priorité vers cette dernière. La formule d'affinité A est alors calculée de la manière suivante :

$$A(\text{super-tâche}, \text{liste}) = \omega \sum_{i=1}^{B(\text{super-tâche})} L(i, \text{équipe}) \times I(i, \text{cache}) \tag{6.3}$$

A : Affinité pour une équipe déportée
ω : Indice d'intensité (compute ratio). Rapport de performances de l'équipe accélératrice par celles de l'équipe locale (facteur facultatif renseigné à la déclaration de la super-tâche)
B : Ensemble des blocs rattachés à la super-tâche
L : Présence en mémoire de la ressource de calcul déportée (0 ou 1)
I : Indice de rétention d'un bloc lié au cache logiciel considéré

Lorsqu'un nœud de calcul nécessite une quantité importante de listes partagées pour mieux capturer les contraintes de localité, l'évaluation de ces paramètres pourrait impacter les performances globales de l'exécution. Une solution serait de générer des tâches légères dédiées à ces évaluations. Ces indices seraient alors calculés en parallèle par plusieurs unités logiques afin de diminuer le surcoût de ces heuristiques. Pour les évaluations présentées dans les sections suivantes, peu de listes partagées sont employées. Ainsi, ce surcoût est en pratique négligeable face à la quantité de calculs embarqués dans chaque super-tâche.

6.2.2 Attribution de super-tâches spécifiques à équilibrage de charge restreint

Les performances des accélérateurs sont souvent dépréciées par des transferts de données coûteux. Ces mouvements sont amplifiés par des aller-retour causés par des phases de calcul déportées intensives et des phases de traitements moins conséquentes effectuées par les processeurs généralistes. En permettant une exécution déportée de ces derniers traitements qui ne sont *a priori* pas adaptés pour les accélérateurs, la quantité de données transférées peut être réduite. Le déport de ces opérations n'est donc plus motivé par une accélération des calculs, qui peut d'ailleurs s'exécuter plus lentement, mais mise sur une meilleure rétention des données dans les caches logiciels. Un équilibrage de charge dynamique n'est donc plus judicieux si celui-ci déclenche d'importants mouvements de données. Ces déplacements peuvent même, par effet boule de neige, engendrer une quantité de transferts bien plus conséquente pour les tâches suivantes et réduire à néant les efforts de déport.

La plate-forme permet la définition de tâches guidées uniquement par la localité (cf. figure 6.10) afin de maximiser les réutilisations locales des données et empêcher un rééquilibrage dynamique. Ces tâches sont directement assignées à l'équipe appropriée dans des listes de tâches dédiées qui permettent tout de même un équilibrage de charge entre unités de l'équipe. Ce fonctionnement est proche de l'attribution d'une simple tâche de calcul, mais permet en plus

de choisir dynamiquement une architecture différente. La plate-forme gère ainsi deux types de super-tâches. Les premières sont orientées vers les équipes qui maximisent la localité tout en permettant des mécanismes d'équilibrages dynamiques et hétérogènes. Les secondes sont assignées une fois pour toutes afin de minimiser les déplacements de données.

Interface de programmation (H3LMS)

```
1 H3LMS_BEGIN_SUPERTASK_DECLARATION
2   ...
3  H3LMS_LOCALITY_ONLY
4 H3LMS_END_SUPERTASK_DECLARATION
```

Proposition de directive

```
1 #pragma h3lms supertask depend
      (...) locality
```

FIGURE 6.10 – Définition de l'attribut de localité pour forcer une exécution de super-tâche maximisant la localité.

6.3 Équilibrage de charge dynamique hiérarchique

Lorsque les super-tâches sont positionnées dans les listes partagées, des mécanismes d'équilibrage de charge hiérarchiques et dynamiques entrent en scène. Cette section présente l'enchaînement des opérations de partage et de vol de tâches de la plate-forme *H3LMS*.

6.3.1 Décomposition des super-tâches et partage de travail

Un représentant de chaque équipe tente tout d'abord de récupérer une super-tâche dans une liste, laquelle est éventuellement partagée avec une autre architecture (cf. **figure 6.11 étape 1A**). Cette restriction empêche que plusieurs unités logiques de la même équipe récupèrent plusieurs super-tâches en même temps et réduisent l'intérêt de la multi-granularité. Lorsqu'une super-tâche est acquise par une unité logique locale, cette dernière la décompose en sous-tâches qui sont réparties tour à tour[116] entre toutes les unités logiques de la même équipe (cf. **figure 6.11 étape 1B**). Une nouvelle tâche ne peut être saisie que s'il n'y a plus d'opportunité d'exécution au sein de la même équipe. Pour cela, des mécanismes de vols locaux permettent de rééquilibrer la charge de travail si la répartition n'était pas assez équitable. La première unité logique qui ne possède plus de tâche à exécuter et qui n'est pas parvenue à en voler une dans son équipe devient un représentant et essaye de récupérer une nouvelle super-tâche. Au contraire, une super-tâche acquise par une unité accélératrice de type *GPU* ne réalise généralement pas une phase de décomposition (cf. **figure 6.11 étape 2**), puisque les unités logiques de ce type sont souvent les seules constituantes de leur équipe associée du fait de la contrainte imposée sur le partage de mémoire physique (cf. **chapitre 4.2**, page 87). C'est en réalité le noyau de calcul du modèle de programmation sous-jacent (*OpenCL*, *CUDA*, etc.) qui se charge de décomposer le travail entre les processeurs de flux. Avec un accélérateur de type *MIC* (*Intel Xeon Phi*), la décomposition peut être soit envisagée à ce niveau dans *H3LMS*, soit comme pour les GPUs, contenue dans un autre modèle (OpenCL, OpenMP, etc.).

[116] Mode *round-robin* en anglais.

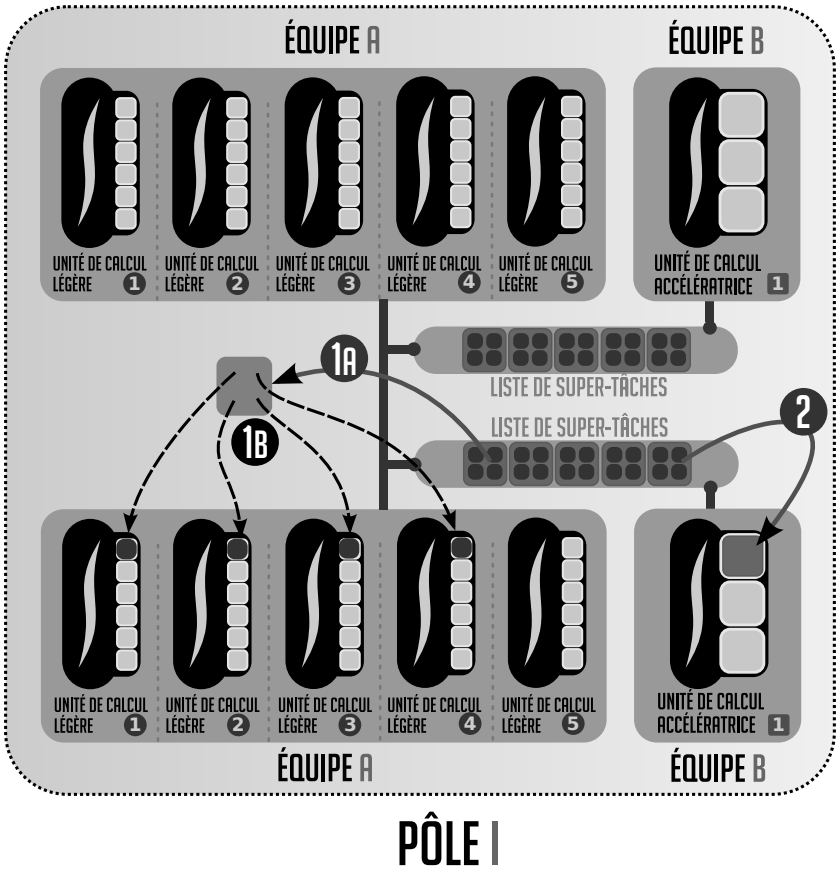


FIGURE 6.11 – Décomposition des super-tâches et partage de travail dans un pôle du nœud de calcul asymétrique *ENS-Fermi* (cf. **annexe A.6**).

6.3.2 Vol collaboratif hiérarchique

La répartition des super-tâches peut être plus ou moins déséquilibrée à cause d'une mauvaise répartition initiale des données ou bien suite à des irrégularités de calcul. Certaines équipes de travail sont donc amenées à achever les traitements associés par affinités plus tôt que d'autres. Des mécanismes de vols hiérarchiques sont employés pour ajuster cette charge de travail. Cet équilibrage dynamique privilégie d'abord des vols locaux. Ainsi, un représentant d'une équipe en attente de traitement va d'abord essayer de récupérer des super-tâches dans des listes contenues dans le même pôle. Si cette tentative est infructueuse, l'expérience est renouvelée pour des listes contenues dans d'autres pôles. Les tentatives de vols lointains sont effectuées par ordre de *facteurs NUMA* croissants afin de réduire les coûts d'accès aux données.

6.4 Exploitation du SMT par l'ordonnanceur

Le pilotage de certains accélérateurs tels que les GPUs nécessite de dédier un cœur de calcul CPU par unité accélératrice. Ces cœurs servent alors exclusivement à initier les transferts de données et à déclencher l'exécution des noyaux de calcul déportés (cf. **chapitre 4.2.1**, page 88). Dans le cadre d'une exécution hétérogène, ils se trouvent donc sous exploités. Réunir une activité de pilotage et d'exécution de noyaux de calcul directement sur le même cœur rend d'organisation de l'ordonnanceur délicate. Il s'agit de prioriser la commande du GPU afin de ne pas pénaliser les performances de ce dernier par un manque de réactivité. Une mauvaise évaluation peut rapidement entraîner une baisse d'efficacité d'exploitation de cet accélérateur matériel.

Certains processeurs généralistes possèdent des unités SMT (cf. chapitre 2.1, page 35) qui leur confèrent la capacité de gérer matériellement deux flots d'instructions simultanément pour chaque cœur de calcul. Cette section explore l'opportunité d'exploiter le SMT pour piloter des accélérateurs et récupérer la puissance de calcul des cœurs qui étaient exclusivement dédiés à leur pilotage. Cette solution est plus simple à mettre en œuvre qu'un arbitrage pris en charge directement par l'ordonnanceur de la plate-forme. Pour ce faire, la représentation abstraite de la machine cible a tout de même été légèrement modifiée. La figure 6.12 présente les modifications apportées à la gestion d'un nœud de calcul *Inti hétérogène* (cf. annexe A.1) afin d'exploiter la fonctionnalité SMT et de tirer parti de tous les cœurs CPU pour du calcul.

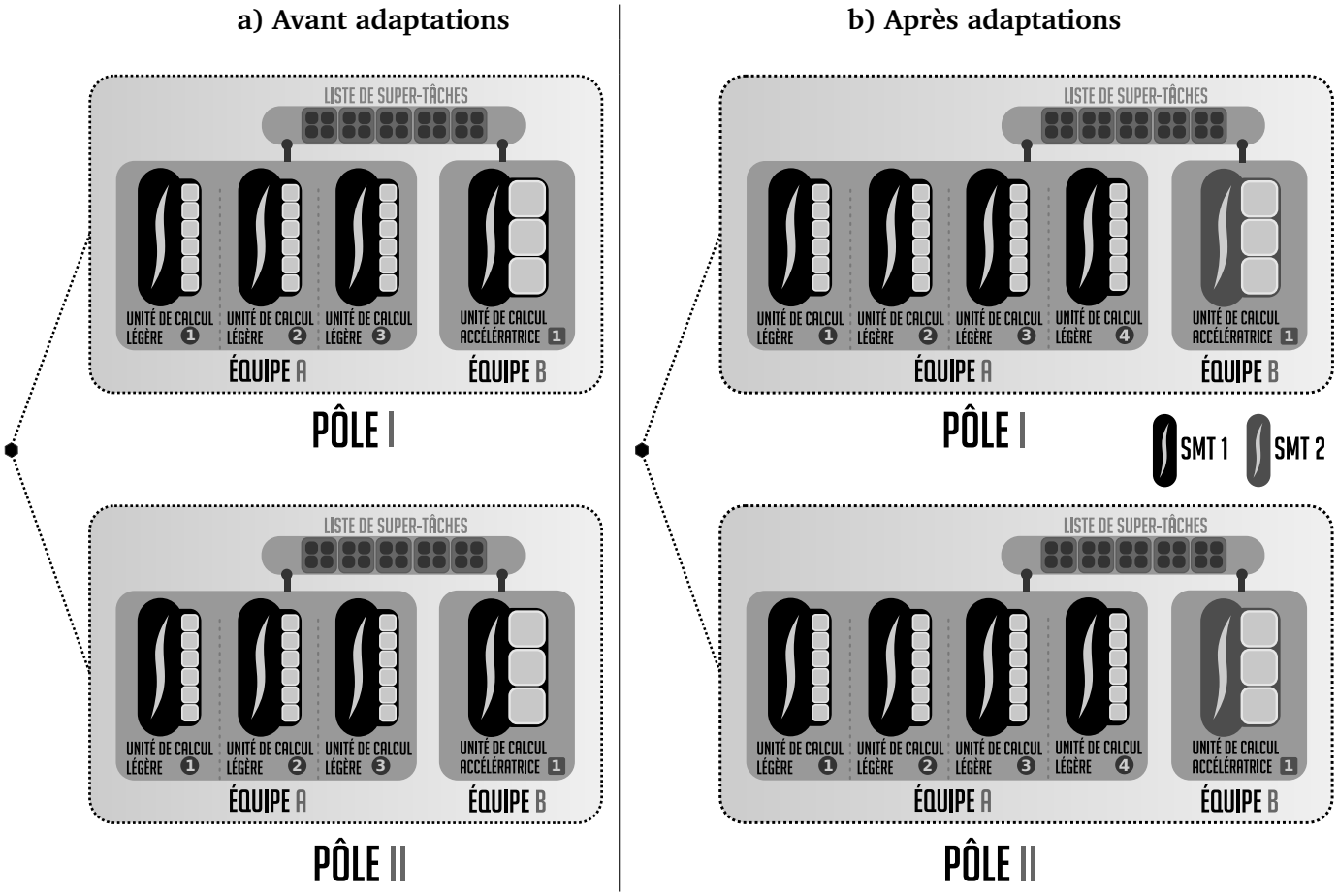


FIGURE 6.12 – Modification de la représentation abstraite d'un nœud de calcul *Inti hétérogène* (cf. annexe A.1) pour le support du SMT.

Les performances sont évaluées sur un nœud de calcul *Inti hétérogène* (cf. annexe A.1) à partir du test de multiplication de matrices déjà employé dans les chapitres précédents (cf. chapitre 5.3.2, page 111). Une série de mesures sont effectuées en activant et désactivant matériellement le SMT dans le BIOS[117]. Plusieurs modes d'exécution sont ainsi comparés en exploitant un ou deux GPUs. Une dimension de $20\,480 \times 20\,480$ est choisie pour les matrices et la taille des blocs de celle-ci est fixée à $1\,024 \times 1\,024$ éléments. La rétention des données est activée entre cinq appels de la bibliothèque, ainsi que la répartition équilibrée des pages mémoires dans les différents nœuds NUMA. Les moyennes et les écarts types sont mesurés à partir de 25 lancements.

[117] **BIOS :** *Basic Input Output System*.

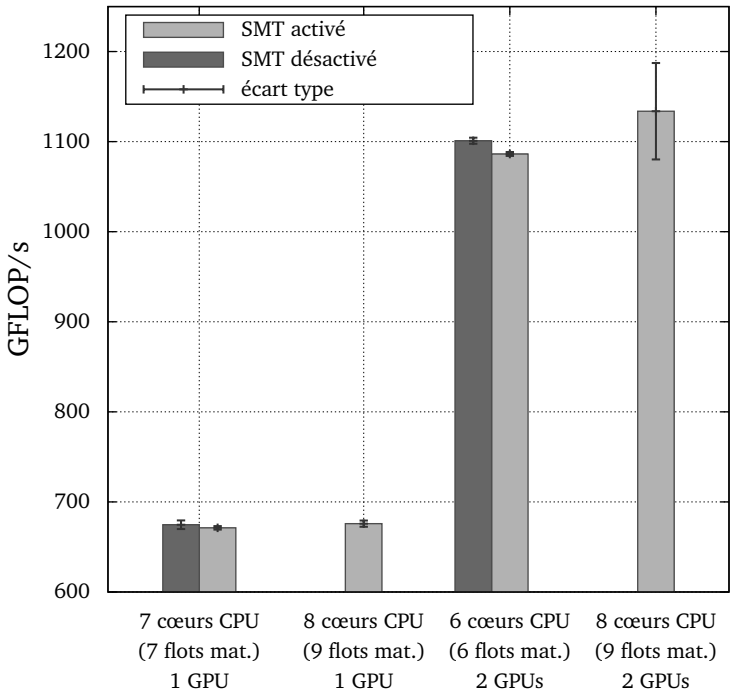


FIGURE 6.13 – Évaluation du comportement de l'ordonnancement lors de l'exploitation du SMT pour le pilotage de GPUs. SGEMM hétérogènes sur des matrices $20\,480 \times 20\,480$ avec deux processeurs Intel Xeon et deux accélérateurs Nvidia M2050 (*Inti hétérogène*, cf. **annexe A.1**). Chaque bloc contient 1024×1024 valeurs. Cinq multiplications sont effectuées à la suite pour permettre la réutilisation des données.

La simple activation du SMT entraîne une légère baisse de performance de 0.5% en exploitant 1 GPU et 7 cœurs de calcul CPU (7 cœurs CPU, 7 flots matériels, 1 GPU). En employant le SMT pour à la fois piloter un GPU et effectuer des opérations de calcul à partir du même cœur CPU (8 cœurs CPU, 9 flots matériels, 1 GPU), un très faible gain de 0.2% est constaté par rapport aux résultats où le SMT est désactivé et le cœur est uniquement dédié au contrôle de l'accélérateur. Le SMT ne semble pas suffisamment intéressant dans un contexte mono accélérateur. Lorsque 2 GPUs sont exploités conjointement à 6 cœurs de calcul CPU (6 cœurs CPU, 6 flots matériels, 2 GPUs), une dégradation de 1.3% est constatée avec le SMT activé. En revanche, lorsque les 2 cœurs qui étaient dédiés au pilotage effectuent aussi des calculs grâce au SMT (8 cœurs CPU, 10 flots matériels, 2 GPUs), un gain de 3% ($32{,}82$ GFLOP/s) est obtenu, mais au prix d'une plus grande instabilité des temps d' exécution.

En résumé, le SMT semble apporter des gains lorsqu'au moins deux GPUs sont exploités. La quasi-totalité de la puissance de calcul, qui était alors inexploitée du fait du contrôle des accélérateurs, peut ainsi être rétablie au profit des traitements. Cela entraîne, en contrepartie, une plus grande instabilité des durées de calcul. Les éléments d'information que nous disposons ne nous permettent pas de conclure sur la source de ces fluctuations. Ces dernières pourraient être induites par une moins bonne réactivité locale du pilotage des GPUs, des effets du SMT sur les mémoires caches de données ou le TLB[118]. Des expériences supplémentaires en présence de plus de deux accélérateurs permettraient de confirmer l'utilité du SMT dans un contexte d'ordonnancement hétérogène et de fournir davantage d'éléments d'explication sur la cause des variations constatées.

[118] **TLB :** *Translation Lookaside Buffer*.

6.5 Intégration à l'environnement de développement MPC

L'environnement de développement *MPC* vise à faciliter la cohabitation de plusieurs modèles de programmation et peut contribuer à diminuer l'empreinte mémoire des exécutions. Comme déjà évoqué dans le **chapitre 3.3.1** (page 66), cet environnement fournit une implémentation de la norme *MPI 1.3* et de la norme *OpenMP 2.5*, lesquelles reposent sur des *processus légers mixtes* (cf. **chapitre 3.2.1**, page 60). Par exemple, depuis un même processus MPC, une instance MPI peut être démarrée par nœud NUMA, chacune embarquant une gestion distincte d'unités logiques de traitement *OpenMP* en mode *compact* afin d'accroître la localité comme l'illustre la **figure 6.14.**

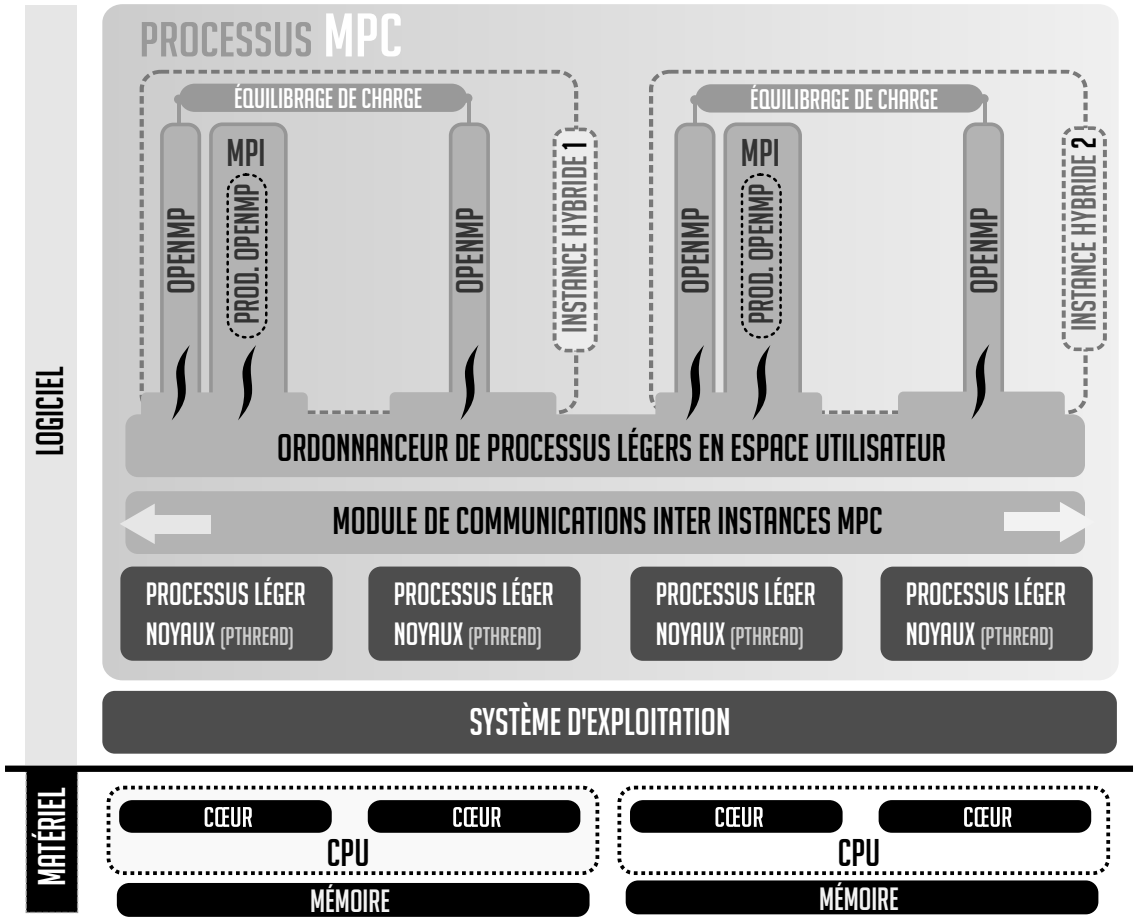


FIGURE 6.14 – L'environnement de développement *MPC*. Fonctionnement en mode hybride *MPI* (deux instances) et *OpenMP* (deux unités de travail par instance MPI).

6.5.1 Mise en œuvre

La plate-forme d'exécution *H3LMS* a été intégrée à *MPC*. Il est ainsi possible de recourir à des tâches de calcul focalisées sur la localité des données pour minimiser les accès lents aux mémoires déportées et aux nœuds NUMA distants. Ce support exécutif étoffe ainsi l'environnement de développement MPC et permettant par la même occasion un support des accélérateurs matériels. La sémantique de *H3LMS* s'en trouve allégée. Les phases d'initialisation et de finalisation du support exécutif sont intégrées aux mécanismes de MPC. Cela évite aux développeurs de renseigner explicitement les fonctions associées.

Pour une telle intégration, le support exécutif a été adapté pour autoriser un fonctionnement non bloquant des unités logiques de calcul. Ces dernières doivent être en mesure de libérer rapidement, et au moment approprié, les ressources de calcul pour faciliter le passage d'un modèle de programmation à un autre. Les processus légers en espace utilisateur de *MPC* sont non préemptif, cette libération doit donc être déclenchée à des moments bien précis, par exemple après plusieurs tentatives de vols de tâches infructueuses. Les prochaines évaluation présentées dans ce manuscrit s'appuieront sur cette version d'*H3LMS* incorporée à *MPC*, démontrant le bon fonctionnement de ce couplage.

6.5.2 Discussion

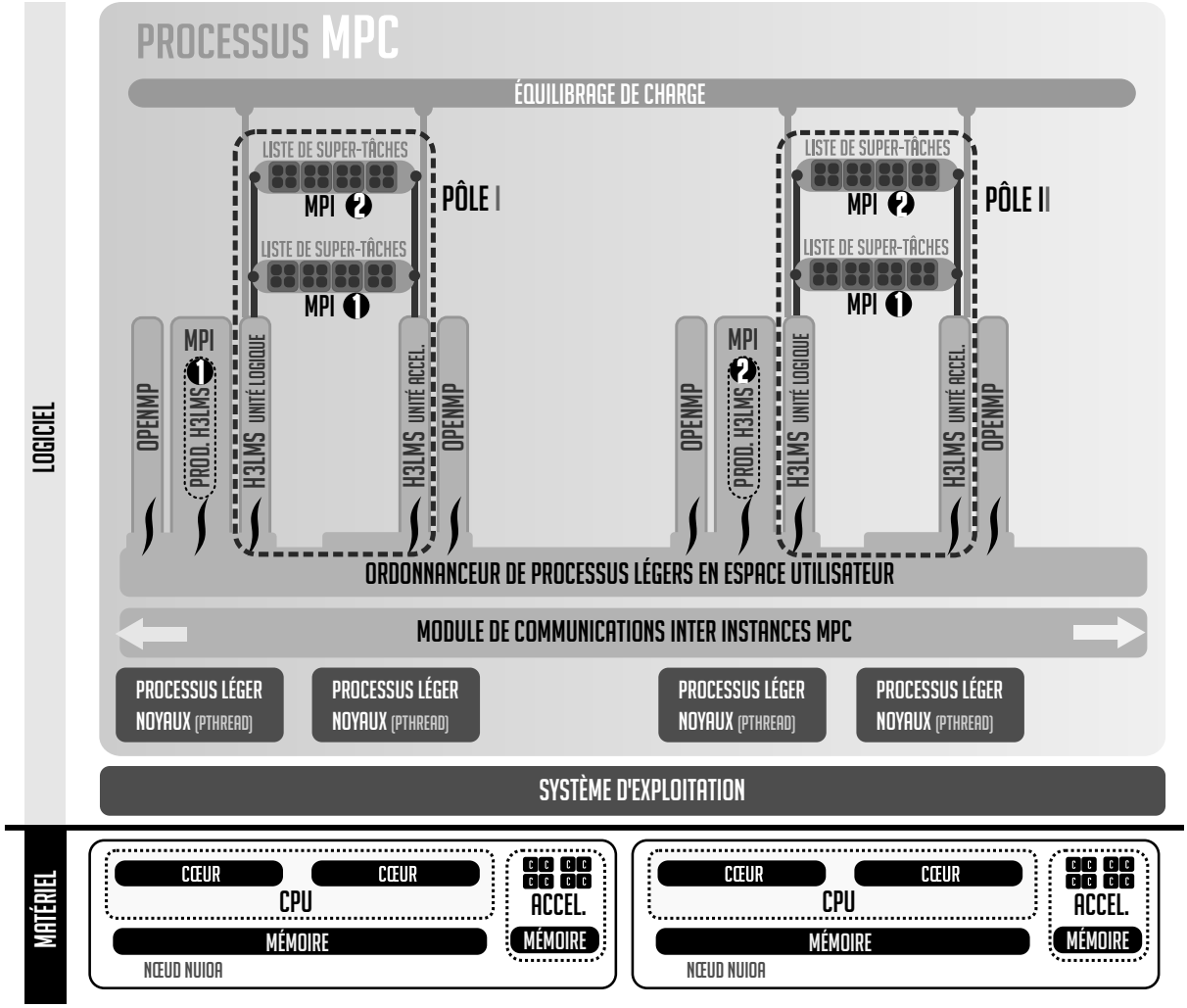


FIGURE 6.15 – *MPC* modifié, intégrant la plate-forme *H3LMS*. Un équilibrage de charge transversal est alors possible entre les instances MPI.

L'intégration de la plate-forme *H3LMS* dans l'environnement de développement *MPC* est nécessaire et prépare la suite des travaux en permettant des exécutions hybrides et collaboratives des tâches de calcul avec d'autres modèles de programmation. En effet, chaque modèle peut profiter des informations possédées par les supports exécutifs des autres afin d'améliorer la gestion des ressources mémoires et de calcul. En particulier, les tâches de calcul générées à partir de deux instances *MPI* différentes pourraient profiter d'un équilibrage de charge en exploitant les mêmes unités logiques de traitement. Comme représenté par la figure 6.15, cette

implémentation multiplie le nombre de listes de super-tâches par le nombre de processus légers de niveau noyau comportant au moins une unité productrice (MPI par exemple). Cela permet de limiter les ralentissements en séparant de mécanismes d'équilibrage dynamique réalisés sur des flots distincts d'opérations. L'évaluation de l'exploitation collaborative de plusieurs modèles de programmation s'inscrit dans le cadre de futurs travaux.

Enfin, l'association d'instances *MPI* et de tâche de calcul pourrait servir à plus facilement apporter des informations sur la répartition des données. Si au moins une instance est employée par nœud NUMA, la localité naturelle du modèle MPI peut être exploitée par la plate-forme *H3LMS* en évitant de recourir à l'interface de répartition des pages mémoire qui peut être complexe à exploiter dans certains cas de figure.

6.6 Résumé des étapes de portage de codes existants

Cette section résume les trois principales étapes utiles pour adapter et optimiser un code existant avec H3LMS. La première consiste en l'identification des sections coûteuses en temps de calcul et qui valent la peine d'être accélérées. Puis, ces sections ainsi ciblées sont portées avec H3LMS. Enfin, les performances de ces portions considérées peuvent être améliorées selon différents axes, en fonction de leur comportement. La **figure 6.16** illustre l'enchaînement de ces différentes étapes.

6.6.1 Identification des sections d'intérêt

Les sections les plus coûteuses en temps de calcul sont identifiées grâce à une instrumentation du code ou à l'aide d'un profileur tel que *Vampir* [244], *Sclalasca* [245], *Tau* [246], *Paraver* [247] ou encore l'outil *MALP*[119] [248] embarqué dans l'environnement de développement MPC. En s'appuyant sur la loi d'Amdahl (cf. **chapitre 3.1.1**, page 58), l'intérêt d'accélérer chacune des portions de code peut être évalué en estimant le gain maximal espéré pour chacune d'entre elles. Typiquement, les fonctions n'occupant que quelques pour cent de l'exécution globale sont ignorées, tant qu'elles n'impactent pas les performances des sections associées à des temps d'exécution plus conséquents. La **figure 6.16, 1-Identification** décrit cette sélection.

6.6.2 Portage avec H3LMS

La deuxième étape s'intéresse au portage d'une section sélectionnée pour être prise en charge par la plate-forme d'exécution H3LMS (cf. **figure 6.16, 2-Portage**). Cette portion de code pourra ainsi profiter d'une exécution parallèle sur processeurs généralistes et éventuellement d'accélérations matérielles. Si les traitements considérés sont déjà pris en charge par une bibliothèque basée sur H3LMS, tels que certaines opérations de type BLAS (cf. **chapitre 5.3.2**, page 111), la modification du code peut se restreindre à l'ajout d'un appel de fonction en lieu et place de la portion de code existante. En revanche, si aucune bibliothèque ne peut se substituer au code de ladite section, un portage manuel peut être envisagé. Dans un premier temps, il faut analyser les données exploitées et générées par la section. Il est nécessaire que ces données soient enregistrées dans le module COMPAS (cf. **chapitre 4.3**, page 90) et associées à une granularité de cohérence si cela n'a pas déjà été réalisé lors du portage d'une autre section impactée par les mêmes données. Cette phase est indispensable pour gérer automatiquement les déplacements de données entre les mémoires disjointes. Les super-tâches sont ensuite déclarées en employant l'interface d'H3LMS (cf. **chapitre 6.1.2**, page 118) ainsi que la décomposition en sous-tâches pour les unités légères (cf. **chapitre 4.2**, page 87). Cette dernière description de sous-décomposition

[119] **MALP :** *Multi Applications Online Profiling*.

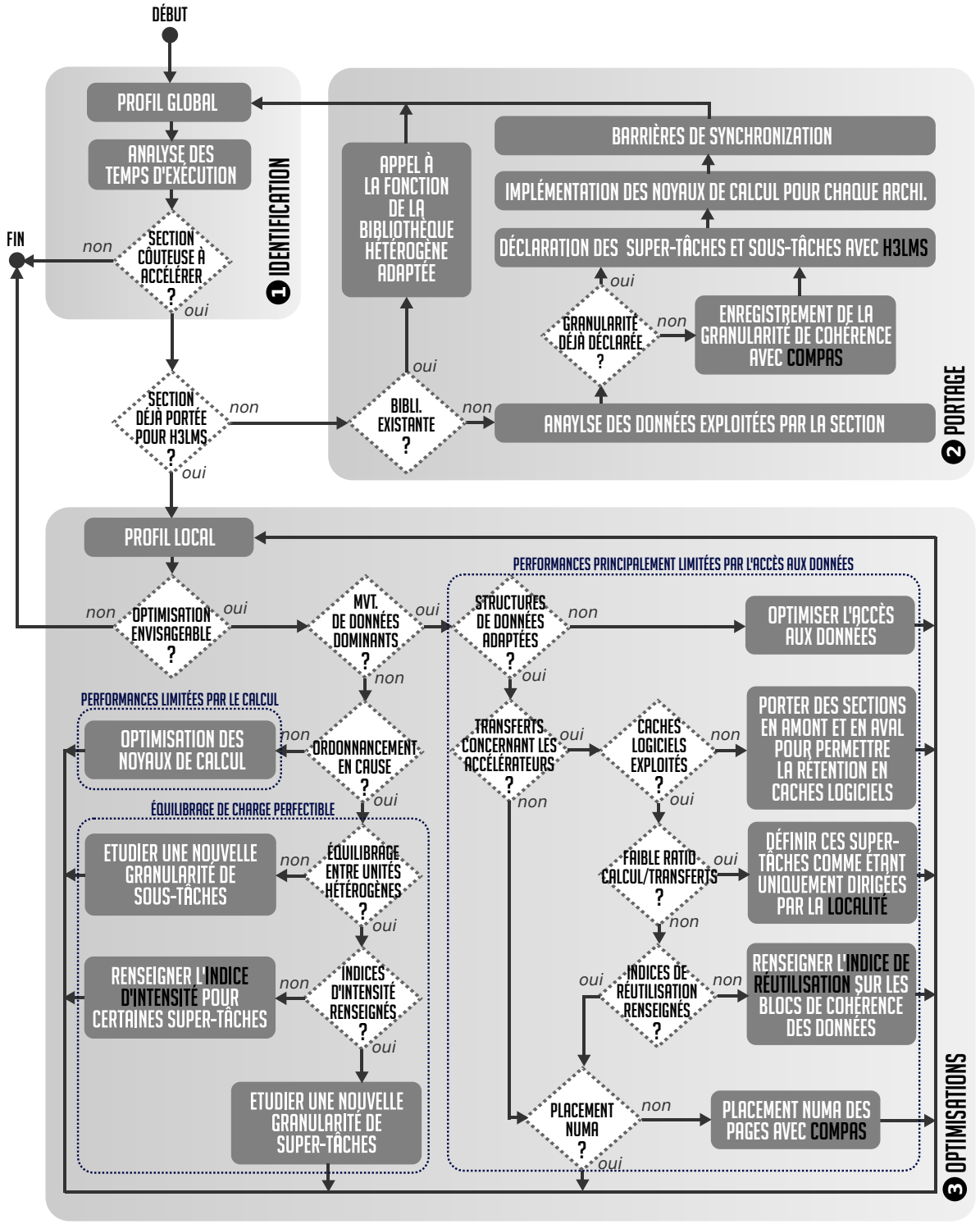


FIGURE 6.16 – Trois étapes pour le portage efficace de codes existants : l'identification des sections coûteuses, le portage de ces sections et leurs optimisations.

peut être ajustée en fonction des opérations réalisées, du moment qu'elle est inférieure ou égale à la granularité d'un bloc de cohérence. Il s'agit ensuite d'implémenter les noyaux de calcul pour chacune des architectures à exploiter et les associer aux super-taches précédemment décrites. Enfin, les barrières de synchronisation doivent être positionnées aux emplacements appropriés, tout comme les instructions permettant de forcer la cohérence des données en mémoire centrale avant d'atteindre une section dont les calculs ne sont pas pris en charge par la plate-forme H3LMS.

6.6.3 Optimisations

La dernière étape, bien que facultative, peut servir à améliorer les performances des portions de codes portées pour H3LMS. Un profilage local est nécessaire pour déterminer la source des ralentissements. Pour cela, des outils d'analyse dynamique tels que *NVIDIA Visual Profiler* [249] (métriques sur les noyaux CUDA, quantité et temps des transferts de données) et *Intel Vtune Amplifier* [250] (défauts de cache et autres comportements de mécanismes de bas niveaux) ou d'analyse statique comme *MAQAO*[120] [251] (analyse le code binaire et fournit des pistes d'améliorations selon des prédictions de performance) peuvent être employés. La **figure 6.16, 3-Optimisations** illustrent les améliorations possibles selon la cause des ralentissements. Celle-ci peut être de trois natures différentes : l'accès aux données, les opérations de calcul ou l'équilibrage de charge. Les sous-sections suivantes décrivent des modifications, lesquelles peuvent améliorer le comportement du code face à chacune de ces limitations.

Performances principalement limitées par l'accès aux données

Il s'agit dans ce cas précis de limiter les accès distants et de favoriser l'exploitation des mémoires locales. Plusieurs axes d'améliorations sont envisageables :

1. Employer des structures de données plus adaptées. Par exemple, convertir les tableaux de structures en structures de tableaux et permettre des accès réguliers aux données. En particulier, cela facilite l'exploitation d'accélérateurs massivement parallèles en favorisant des accès coalescés (cf. **chapitre 2.4.3**, page 51). Les structures et la granularité des sous-tâches doivent éviter les accès induisant du faux partage (cf. **chapitre 2.2.2**, page 39). Des sous-opérations par accès morcelés peuvent être envisagées afin d'accroître la localité des données pour des portions de codes fortement optimisées (cf. **chapitre 2.2.1**, page 37).
2. Permettre une exploitation des caches logiciels. Pour cela, il faut porter en super-tâches les portions de codes en amont et en aval qui manipulent les mêmes données que la section considérée. Cela rend possible une rétention des données dans les caches logiciels entre ces sections.
3. Identifier les super-tâches caractérisées par un faible ratio calculs/transferts et adapter leur mode d'ordonnancement pour éviter de déplacer une grande quantité de données lors d'un équilibrage de charge dynamique. Ces super-tâches sont alors définies comme étant *locales*
4. Renseigner les *indices de potentiel de réutilisation* sur les blocs de cohérence grâce au module *COMPAS* afin d'accroître l'efficacité de rétention des caches logiciels.
5. Lorsqu'un seul producteur est employé, imposer une affinité des pages avec *COMPAS* afin de répartir les données sur les différents nœuds NUMA. Cela permet de profiter de la bande passante de tous les processeurs (cf. **chapitre 2.3.2**, page 46) et d'accroître la localité en aiguillant l'ordonnanceur pour préférer les accès locaux. Cette fonctionnalité peut réduire à la fois les effets NUMA (cf. **chapitre 2.3.1**, page 42) et NUIOA (cf. **chapitre 2.4**, page 46).

Équilibrage de charge perfectible

Pour certaines opérations, l'ordonnancement peut être amélioré en renseignant des informations supplémentaires ou en ajustant la granularité des tâches :

[120] **MAQAO :** *Modular Assembly Quality Analyzer and Optimizer*.

1. Un équilibrage de charge insuffisant entre unités légères peut apparaître si les sous-décomposions ne génèrent pas assez de sous-tâches. En ajustant la granularité des sous-tâches pour un grain plus fin, l'efficacité d'exploitation de ces unités légères peut être améliorée.
2. Lorsque l'équilibrage hétérogène est en cause, le fait de préciser les éventuels *indices d'intensité* (cf. **chapitre 6.2.1**, page 122) fournit des renseignements plus précis à l'ordonnanceur. Ainsi, les super-tâches posséderont une affinité particulière pour l'architecture la mieux adaptée.
3. En dernier recours, des nouvelles granularités des super-tâches peuvent être étudiées.

Performances limitées par le calcul

Il s'agit dans ce cas d'apporter des optimisations usuellement employées pour des noyaux renfermant une grande quantité de calcul, tel qu'éviter la redondance des opérations ou limiter l'emploi d'opérations coûteuses lorsque c'est possible. Des optimisations de plus bas niveau peuvent être également envisagées si le gain attendu peut être suffisamment intéressant. En particulier, le programmeur peut vérifier qu'une vectorisation adéquate des opérations est réalisée à la compilation.

Pour résumer, ce chapitre a présenté la plate-forme d'exécution *H3LMS*, laquelle intègre les contributions détaillées dans les chapitres précédents. Son interface de programmation a tout d'abord été explicitée sur un exemple concret de factorisation LU de matrice dense. *H3LMS* s'appuie sur la représentation abstraite de la machine cible ainsi que du module *COMPAS* afin d'aiguiller l'exécution des super-tâches en fonction de l'architecture du nœud de calcul et de la localité des données. En particulier, deux modes d'attribution des super-tâches ont été décrits en fonction du type des opérations embarquées dans les super-tâches. Les heuristiques permettant ces aiguillages ont été précisées en s'appuient sur le nœud de calcul asymétrique *ENS-Fermi*, architecture la plus atypique que nous ayons à disposition. Les mécanismes d'équilibrage de charge hiérarchiques ont également été présentés. Ceux-ci permettent de faire face à des déséquilibres d'ordonnancement, tout en conservant une certaine localité des données. Enfin, l'intégration d'*H3LMS* dans l'environnement de développement *MPC* a été abordée. Elle offre une meilleure cohabitation d'*H3LMS* avec d'autres modèles de programmation.

Chapitre 7

Évaluations

> *“ L'observation recueille les faits, la réflexion les combine ; l'expérience vérifie le résultat de la combinaison. ”*
>
> Denis Diderot, *Pensées sur l'Interprétation de la Nature*, 1753

Les **chapitres 1** (page 17) et **2** (page 33) ont discuté des contraintes liées à l'exploitation du matériel. En particularité, des résultats de mesures ont été recueillis et justifient de concentrer des efforts sur la façon dont les données des programmes sont accédées. Dans un contexte hétérogène, d'autres difficultés apparaissent quant à la gestion des mémoires déportées. Dans les **chapitres 4** (page 79) et **5** (page 97), des mécanismes ont ensuite été proposés pour maintenir cohérentes des données stockées dans des mémoires séparées tout en amplifiant leurs localités spatiale et temporelle. Ils sont alors agrégés dans le support exécutif *H3LMS*, introduit dans le **chapitre 6** (page 113). Ce dernier est finalement intégré à l'environnement de programmation *MPC* afin de faciliter la cohabitation avec d'autres modèles de programmation. Il s'agit désormais d'évaluer expérimentalement les performances de ces combinaisons. Pour ce faire, les performances du couple *H3LMS-MPC* sont évaluées à partir de deux applications aux comportements radicalement opposés. L'une est contrainte par les capacités de calcul[121] de la machine mettant au premier plan l'ordonnancement hétérogène. La seconde est fortement limitée par l'accès aux données[122], reposant davantage sur les mécanismes permettant d'amplifier la localité.

Les premières évaluations concernent le programme de performance *High Performance Linpack*. Celui-ci est dérivé de la bibliothèque *Linpack* et est principalement limité par la puissance délivrée par les ressources de calcul lorsqu'il manipule des cas tests suffisamment grands. La version officielle de l'algorithme relayée par le site web du classement *TOP500* est élaborée pour des machines à ressources de calcul homogènes. Le code source a été légèrement adapté afin de recourir à des accélérateurs tout en minimisant la quantité de lignes modifiées.

Les secondes appréciations se focalisent ensuite sur un code de calcul développé au *CEA* appelé *PN*. Ce code de simulation, dont l'objectif sera présenté plus loin dans ce chapitre, est progressivement adapté afin bénéficier d'accélérations de matériels spécialisés. Contrairement au test *Linpack*, ce programme offre un aperçu plus réaliste des problématiques rencontrées dans certaines applications de simulation. En particulier, les performances de ce programme sont fortement contraintes par les transferts de données. Des adaptations sont alors successivement apportées afin de réduire l'impact des mouvements des données sur le temps d'exécution.

[121] *Compute bound*.

[122] *I/O bound*.

7.1 High Performance Linpack

Le test de performance *HPL*[123] résout un système linéaire dense $Ax = b$ en inversant la matrice A avec une factorisation optimisée. Comme évoqué dans le **chapitre 1** (page 17), il est usuellement employé pour comparer les capacités de calcul délivrées par des supercalculateurs pour le classement *TOP500*. C'est un code régulier qui fonctionne en double précision et dont le temps d'exécution est dominé par des opérations de type *BLAS 3*. Les paramètres permettant d'atteindre les plus hautes performances sont déterminés par expérimentation. Ils concernent la taille de la matrice, la dimension des blocs, la façon dont les données sont stockées (alignement et ordre) ainsi que des variantes algorithmiques. J. Kurzak et J. Dongarra présentent une version modifiée de l'implémentation *HPL* afin d'exploiter les processeurs *SPE* d'un *CELL* [252]. Ils fournissent des efforts conséquents dédiés à cette application. Fatica et al. proposent une version hétérogène *CPUs* et *GPUs* à partir d'une répartition plus ou moins statique des opérations contenues dans les multiplications de matrices [253]. Aucun mécanisme ne permet l'équilibrage de charge au sein d'une même itération de l'algorithme. Cependant, en fonction des périodes d'inactivité constatées sur les différentes unités de calcul, la répartition est réajustée à l'itération suivante.

L'évaluation présentée dans cette section a pour objectif de valider le fonctionnement du support exécutif *H3LMS* dans un code de calcul existant en minimisant les modifications. Elle permet un équilibrage de charge entre unités de traitement issues de la même architecture, et entre unités de natures différentes (CPUs et accélérateurs) pour des noyaux de calcul spécifiques. Pour cela, le code optimisé *HPL 2.1* [254] est adapté pour faire appel à la bibliothèque de multiplications de matrices hétérogènes. Les portions de code à l'extérieur des appels de type *BLAS 3* sont exécutées de manière statique sur un cœur de calcul, sans qu'elles soient prises en charge par la plate-forme d'exécution. Les données sont rendues cohérentes en mémoire centrale entre chaque lot d'opérations de type *BLAS 3*. Il n'est donc pas pertinent d'étudier le mode d'exécution maximisant la localité des données.

7.1.1 Modifications

Ce portage requiert la modification de seulement 3 lignes de code pour une prise en charge hétérogène. Il s'agit premièrement d'adapter l'allocation de la matrice et permettre l'emploi de *pages punaisées*. Ainsi, la bande passante des transferts entre la mémoire centrale et les accélérateurs déportés est améliorée et évite de recourir à des copies intermédiaires déclenchées par la bibliothèque *CUDA* (cf. **chapitre 2.4.1**, page 48). La **figure 7.1** précise l'adaptation réalisée. Aucun schéma de décomposition n'est spécifié car la modification minimale de ce programme ne permet pas de conserver des données dans les caches logiciels associées aux accélérateurs déportés. Le premier argument "*NULL*" est ainsi renseigné.

testing/ptest/HPL_pdtest.c - avant modif.

```
164  vptr = (void*)malloc(
165           ((size_t)(ALGO->align)+
166           (size_t)(mat.ld+1)*
167           (size_t)(mat.nq))*
168            sizeof(double));
```

testing/ptest/HPL_pdtest.c - après modif.

```
164  vptr = (void*)(compas_malloc(NULL,
              COMPAS_PLOCKED,
165           ((size_t)(ALGO->align)+
166           (size_t)(mat.ld+1)*
167           (size_t)(mat.nq))*
168            sizeof(double));
```

FIGURE 7.1 – Modification de l'allocation pour recourir à des pages punaisées en mémoire.

[123] **HPL :** *High Performance Linpack*.

Les appels aux fonctions de type type *BLAS 3* sont ensuite remplacés par des fonctions synchrones associées la bibliothèque hétérogène basée sur *H3LMS*. Puisqu'aucune description de granularité des données n'a été fournie à l'allocation, une interface étendue de la multiplication de matrice permet de spécifier la taille des blocs et des sous-blocs pour un meilleur contrôle sur la granularité des super-taches et des tâches comme le spécifie la figure 7.2. Aucune autre modification n'est nécessaire. Dans ce cas, les super-taches manipulent de blocs de $4096 \times m$ où m est la quantité de données modifiée à chaque itération dans une colonne de la matrice. Les tâches internes s'opèrent quant à elles sur des sous-blocs de $256 \times m$. L'appel spécifique de multiplications de matrices gère les flux *CUDA* afin recouvrir au maximum les transferts de données par du calcul, et ce dans les deux sens de manière *"pipelinée"*. Puisqu'aucune affinité des données n'est fournie, les super-tâches sont réparties de manière aléatoire entre les pôles.

include/hpl_blas.h - avant modif.

```
164 #define HPL_dgemm cblas_dgemm

165 #define HPL_dtrsm cblas_dtrsm
```

include/hpl_blas.h - après modif.

```
164 #define HPL_dgemm (...)
        h3lms_blas_dgemm_ext_sync(__VA_ARGS__,
         4096,-1,256,-1)
165 #define HPL_dtrsm h3lms_blas_dtrsm_sync
```

FIGURE 7.2 – Modification des appels aux fonctions *BLAS 3*.

La bibliothèque de calcul n'exploite pas les capacités des caches logiciels, car la plate-forme n'a pas connaissance du reste du code. La cohérence est donc assurée à la fin de chaque appel en mémoire centrale, comportement souhaité par les versions synchrones des appels de bibliothèque suffixés par le mot clé *_sync*. La multiplication de matrices à multi niveaux de granularité repose sur les noyaux optimisés des bibliothèques *Intel MKL 12* et *Nvidia CUBLAS 4.2*.

7.1.2 Résultats

L'application *Linpack* est exécutée sur un nœud de calcul hétérogène de *Tera-100 hétérogène* (cf. annexe A.2) composé de deux CPUs *Intel Xeon* quadri-cœurs, de deux GPUs *Nvidia Tesla M2090* et de 24 Go de mémoire centrale. Afin de démontrer l'efficacité de la solution, quatre scénarii sont comparés :

1. Exécution sur un cœur de calcul processeur généraliste. Les noyaux de type *BLAS 3* s'appuient sur la version séquentielle de la bibliothèque *Intel MKL 12*.

   ```
   OMP_NUM_THREADS=1 ./xhpl
   ```

2. Exécution parallèle homogène sur huit cœurs de calcul *Intel Xeon*, en s'appuyant sur la version parallèle de la bibliothèque *Intel MKL 12* (*Intel OpenMP*).

   ```
   OMP_NUM_THREADS=8 ./xhpl
   ```

3. Exécution parallèle hétérogène sur huit cœurs de calcul *Intel Xeon*, en s'appuyant sur la plate-forme d'exécution et sa bibliothèque de multiplication de matrices à multi-granularité. Les super-tâches manipulent de blocs de $4096 \times m$ où m est la quantité de données modifiée à chaque itération dans une colonne de la matrice. Les super-tâches sont décomposées en tâches de dimension $256 \times m$.

   ```
   mpcrun -N=1 -n=1 -c=8 ./xhpl
   ```

4 Exécution parallèle hétérogène sur deux accélérateurs *Nvidia Telsa* et six cœurs de calcul *Intel Xeon* (les deux autres cœurs servent à piloter les deux accélérateurs), en s'appuyant sur la plate-forme d'exécution et sa bibliothèque à multi granularité. Chaque super-tâche est décomposée en quatre sous-blocs lorsqu'elles sont traitées par des accélérateurs et permettent ainsi le recouvrement par du calcul des sous-blocs transférés.

```
mpcrun -N=1 -n=1 -c=8 –accel=2 ./xhpl
```

La taille de la matrice est de $46\,080 \times 46\,080$ et est choisie pour occuper au maximum la mémoire centrale sans que le système effectue de coûteuses *paginations disque* [124]. La dimension des blocs de la matrice est fixée à 512 et est un compromis de performances entre *CPUs* et *GPUs*. Les éléments sont stockés colonne par colonne[125]. Tous les paramètres employés pour ces évaluations sont résumés dans la **figure 7.3a**. La **figure 7.3b** révèle les performances obtenues pour les scénarii détaillés précédemment.

a) Paramètres

N	46080
NB	512
P	1
Q	1
méthodes	WC10L2L2

b) Performances mesurées en double précision

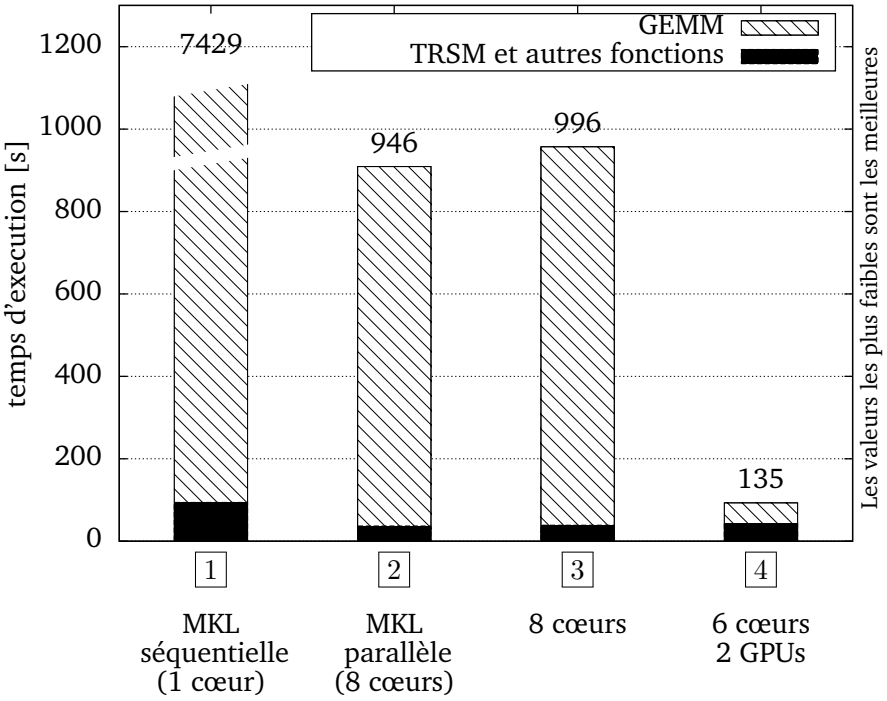


FIGURE 7.3 – *HPL 2.1* exécuté sur un nœud de calcul *Tera-100 hétérogène* (cf. **annexe A.2**) **a)** Paramètres d'exécution. **b)** Performances en double précision, atteintes selon le mode d'exécution.

[124] **swap :** *mécanise de la mémoire virtuelle qui permet d'étendre la mémoire utilisable à des disques de stockage lorsque la mémoire centrale est pleine.*

[125] *Column-major* en anglais

Bien que l'emploi de la multi-granularité entraîne une légère dégradation des performances due à un rendement moins élevé par noyau et par le surcoût de l'ordonnancement, elle permet d'atteindre des performances plutôt proches de celles atteintes avec la bibliothèque *MKL* en mode parallèle. Le programme s'exécute en 996 secondes ($62{,}11$ GFLOP/s), contre 946 secondes ($68{,}94$ GFLOP/s) pour la solution purement *Intel MKL*. L'accélération est de $7{,}46\times$ par rapport à l'exécution séquentielle, soit une efficacité parallèle de $93{,}25\%$. Une exploitation de la totalité des ressources de calcul (mis à part les deux cœurs nécessaires pour le pilotage des accélérateurs) réduit le temps de calcul à 135 secondes ($482{,}4$ GFLOP/s) ce qui représente une accélération de $7{,}8\times$ par rapport à l'exécution parallèle homogène et de $55{,}03\times$ face au traitement séquentiel.

7.1.3 Discussion des résultats et améliorations possibles

Pour résumer, en modifiant uniquement 3 lignes de code, le test *Linpack* peut profiter d'une accélération substantielle des matériels spécialisés. Du fait des surcoût induits par l'ordonnanceur, et puisqu'il s'agit d'un programme de calcul régulier sans important déséquilibre, les performances ne peuvent rivaliser avec une optimisation statique dédiée à une architecture donnée.

Les performances en mode hétérogène sont environ 30% inférieures à celles publiées pour des versions optimisées de manière *statique* par Fatica et al. [255] avec une matrice de dimension plus conséquente. Cela semble décevant aux premiers abords, mais cet écart ne peut être entièrement dû à un surcoût de l'ordonnanceur. Les performances du programme *HPL* sont en réalité fortement limitées par la taille du problème. Plus la matrice est conséquente, plus les calculs des multiplications *GEMM* occupent une proportion conséquente du temps de traitement global. Cette fonction présente une efficacité bien supérieure aux autres opérations et est en grande partie responsable des performances du test. Les dernières mesures en mode hétérogène indiquent que la part des multiplications n'occupe plus qu'environ 50% du temps d'exécution. Ainsi, cela fait baisser les performances globales. Augmenter la taille de la matrice permettrait d'accroître cette proportion et d'élever la moyenne de GFLOP/s. Dans la littérature, les performances supérieures de 30% sont obtenues à partir d'une plus grande matrice en ayant recours à un nœud de calcul avec bien plus de mémoire centrale : 96 Go permet un travail sur une matrice de $108\,032 \times 108\,032$ contre un maximum de $46\,080 \times 46\,080$ avec les 24 Go disponibles sur le nœud de calcul employé pour nos évaluations.

D'autres améliorations peuvent être envisagées, telles qu'une meilleure parallélisation du noyau *TRSM* ou un couplage à une exécution en mode *MPI* afin d'apporter du parallélisme sur les parties conservées séquentielles entre les appels de type *BLAS 3*.

7.2 Mini-application PN

PN est une *mini-application* du CEA qui a pour objectif de résoudre l'équation linéaire du transport des particules. La méthode de résolution déterministe retenue consiste à projeter l'opérateur de transport sur une base d'harmonique sphérique tronquée [256]. Ainsi, une équation dépendant de six variables distinctes (trois d'espace, une de temps et deux variables angulaires) est remplacée par un modèle de $(N+1)^2$ équations ne dépendant plus que de l'espace et du temps. L'application combine les modèles de programmation *OpenMP* et *MPI* et repose en partie sur la bibliothèque optimisée d'algèbre linéaire *Intel MKL* pour les phases de calcul matriciel. Cette section décrit les adaptations utiles pour recourir à la plate-forme d'exécution afin d'exploiter des ressources de calcul hétérogènes composées de processeurs graphiques et de processeurs généralistes multi-cœurs. Il s'agit d'apporter des accélérations substantielles et de réduire les efforts de portage du code en permettant des améliorations incrémentales.

7.2.1 Analyse et modifications de l'algorithme

Cette sous-section a pour vocation d'étudier le code de simulation PN afin de se focaliser sur les fonctions qui occupent le plus de temps. Des adaptations seront ensuite réalisées afin de faciliter le recours à des super-tâches de calcul hétérogènes.

Profil d'exécution séquentielle

L'algorithme opère sur un maillage cartésien à deux dimensions x et z et effectue plusieurs traitements qui sont répétés à chaque pas de boucle en temps. Il est composé d'un enchaînement de quatre fonctions principales : *flux numérique x*, *flux numérique z*, *émission-absorption* et *volumes finis*. La **figure 7.4** présente les proportions des temps passés dans chaque des fonctions pour chaque itération en temps, lors d'une exécution séquentielle sur un cœur de calcul d'un nœud hétérogène de *Tera-100 hétérogène* (cf. **annexe A.2**). Pour cette étude du profil d'exécution, $N = 15$ pour la décomposition des harmoniques sphériques (résolution angulaire) et elle est appliquée à un maillage cartésien de 1536×1536 (résolution spatiale $x = z = 1536$) avec quelques pas de temps.

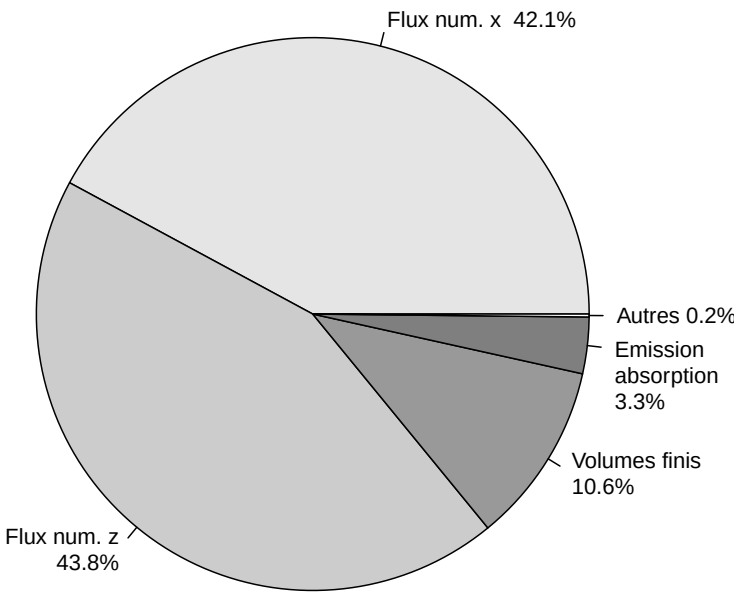


FIGURE 7.4 – Proportions du temps passé dans chaque fonction.

Les fonctions *flux numérique x* et *flux numérique z* représentent à elles seules près de 86% du temps d'exécution séquentiel. Les modifications qui suivent se focalisent alors sur ces deux fonctions qui occupent la plus grande proportion du temps. Elles sont légèrement modifiées afin de faciliter la création de tâches de calcul et améliorer la localité spatiale.

Adaptations

Les fonctions *flux numérique x* et *flux numérique z* effectuent globalement les mêmes opérations, mais sur des dimensions différentes, dont les données modifiées sont localisées dans des matrices différentes. Les temps d'exécution sont disparates, une différence de 4% peut être constaté. La disposition des données selon l'axe z ne permet pas d'effectuer des traitements sur des données consécutives en mémoire ce qui mène a une légère dégradation des performances réduisant la localité spatiale. La disposition des données contenues dans cette matrice est modifiée afin d'égaler les temps de calcul des deux fonctions flux numérique. Cette transformation facilitant également la création de tâches de calcul puisqu'elle retire de multiples dépendances de données. Cela allégera ainsi les efforts liés à la résolution des dépendances par la plate-forme d'exécution.

Les deux fonctions *flux numérique x et z* sont indépendantes et peuvent être calculées en parallèle. Les deux appels sont fusionnés dans une nouvelle fonction générique appelée simplement *flux numériques*. Les données associées à chacune des dimensions x et z sont placées dans les mémoires de nœuds NUMA distincts. Ces transformations permettent à la fois de regrouper les opérations équivalentes entre des barrières de synchronisation et de répartir les données sur des nœuds NUMA distincts afin d'accroître la localité spatiale et de réduire les accès distants.

Fonction *Flux numériques*

La fonction *flux numériques* nouvellement créée peut être décomposée en quatre étapes distinctes comportant les opérations suivantes :

- étape 1 : Deux multiplications indépendantes de grandes matrices.

- étape 2 : Quatre multiplications indépendantes de petites matrices.

- étape 3 : Plusieurs tâches de calcul définies à la main.

- étape 4 : Deux multiplications indépendantes de grandes matrices.

Les multiplications de petites matrices (étape 2) ne sont jamais effectuées par des accélérateurs. En revanche, elles peuvent être exécutées en parallèle avec les premières multiplications de grandes matrices (étape 1) puisqu'il n'existe aucune dépendance entre les étapes 1 et 2. Les étapes 3 et 4 exploitent les résultats générés par les étapes qui les précèdent. Lorsque la fonction *flux numériques* se termine à chaque pas de temps, les données sont transférées en mémoire centrale pour assurer leur cohérence. Les autres traitements qui ne sont pas pris en charge par la plate-forme, dont *émission-absorption* et *volumes finis,* peuvent donc s'exécuter sur des données à jour en mémoire centrale.

La figure 7.5 révèle les opérations et la quantité de données manipulées pour un maillage de dimensions $x \times z$, ainsi que la position des points de synchronisation. Ces derniers sont imposés par le modèle de tâches synchrones par lots pour respecter les dépendances entre chaque étapes, à la façon dont des tâches de calcul seraient synchronisées avec la norme *OpenMP 3.0* (cf. chapitre 3.3.1, page 65).

Pseudo code : fonction *flux numériques* modifiée

```
1  /* Etape 1 : grandes multiplications de matrices */
2  GEMM [lecture:A_{X(136×136)}, B_{X(x×z×136)}] [ecriture:C_{X(x×z×136)}]
3  GEMM [lecture:A_{Z(136×136)}, B_{Z(x×z×136)}] [ecriture:C_{Z(x×z×136)}]
4
5  /* Etape 2 : petites multiplications de matrices */
6  GEMM [lecture:A_{X(136×136)}, BL_{X(x×136)}] [ecriture:CL_{X(x×136)}]
7  GEMM [lecture:A_{Z(136×136)}, BL_{Z(z×136)}] [ecriture:CL_{Z(z×136)}]
8  GEMM [lecture:A_{X(136×136)}, BR_{X(x×136)}] [ecriture:CR_{X(x×136)}]
9  GEMM [lecture:A_{Z(136×136)}, BR_{Z(z×136)}] [ecriture:CR_{Z(z×136)}]
10 BARRIERE
11
12 /* Etape 3 : taches */
13 SUPER-TACHE [lecture:D_{X(1024)}, C_{X(x×z×136)}] [ecriture:E_{X(x×z×136)}]
14 SUPER-TACHE [lecture:D_{Z(1024)}, C_{Z(x×z×136)}] [ecriture:E_{Z(x×z×136)}]
15 BARRIERE
16 SUPER-TACHE [lecture:D_{X(1024)}, CL_{X(x×136)}] [lecture-ecriture:E_{X(x×z×136)}]
17 SUPER-TACHE [lecture:D_{Z(1024)}, CL_{Z(z×136)}] [lecture-ecriture:E_{Z(x×z×136)}]
18 BARRIERE
19 SUPER-TACHE [lecture:D_{X(1024)}, CR_{X(x×136)}] [lecture-ecriture:E_{X(x×z×136)}]
20 SUPER-TACHE [lecture:D_{Z(1024)}, CR_{Z(z×136)}] [lecture-ecriture:E_{Z(x×z×136)}]
21 BARRIERE
22
23 /* Etape 4 : grandes multiplications de matrices */
24 GEMM [lecture:F_{X(136×136)}, E_{X(x×z×136)}] [ecriture:G_{X(x×z×136)}]
25 GEMM [lecture:F_{Z(136×136)}, E_{Z(x×z×136)}] [ecriture:G_{Z(x×z×136)}]
26 BARRIERE
27
28 EVACUATION-CACHES
```

FIGURE 7.5 – Fonction *flux numériques* sur un maillage $x \times z$.

7.2.2 Expérimentations

Un nœud hétérogène du supercalculateur *Tera-100* (cf. annexe A.2) a été exploité pour les évaluations qui vont suivre. Les matrices sont décomposées en blocs de dimensions 1536×136. La figure 7.6 présente les modifications apportées aux allocations des matrices Bx et Bz en utilisant le module *COMPAS*. Les autres matrices sont allouées de manière similaire. Enfin, le couple *H3LMS-MPC* a été employé pour mesurer l'efficacité de la solution.

Répartitions des pages, allocations et blocs de cohérence (COMPAS).

```
1  /* Declaration des caracteristiques du placement des pages */
2  struct compas_pages_s pages;
3  compas_pages_init(&pages);
4  pages.geometry = COMPAS_1D;
5  pages.pattern = COMPAS_NODE;
6  pages.node = compas_get_firstnode();
7  pages.xblocks = 1;
8  pages.xblocksize = (sizeof(double)*1536*1536*136)/PSIZE;
9  pages.elsize = sizeof(double);
10
11 /* Allocation des matrices selon la distribution des pages */
12 double *Bx = compas_malloc(&pages,COMPAS_PLOCKED,sizeof(double)
       *1536*1536*136);
13
14 pages.node = compas_get_nextnode(pages.node);
15 double *Bz = compas_malloc(&pages,COMPAS_PLOCKED,sizeof(double)
       *1536*1536*136);
16
17 /* Declaration des blocs de coherence. */
18 struct compas_data_s data;
19 compas_data_init(&data);
20 data.geometry = COMPAS_2D;
21 data.xblocks = 1;
22 data.yblocks = 1536;
23 data.xblocksize = 136;
24 data.yblocksize = 1536;
25 data.elsize = sizeof(double);
26
27 compas_data_register(&data, Bx, COMPAS_COLMAJOR);
28 compas_data_register(&data, Bz, COMPAS_COLMAJOR);
```

FIGURE 7.6 – Modification des allocations et ajout de code (module *COMPAS*).

Bibliothèque hétérogène

La première approche consiste à seulement employer la bibliothèque hétérogène pour limiter les efforts de portage. La figure 7.7 détaille la modification mineure effectuée sur l'étape 1 de la fonction *flux numériques*. Une adaptation similaire est réalisée sur l'étape 4. Les tâches de calcul de l'étape 3 sont adaptées et déclarées avec l'interface de *H3LMS*.

étape 1 - avant modifications

```
120 clas_dgemm(CblasColMajor,
        CblasNoTrans, CblasNoTrans, 136,
        1536*1536, 136, 1., Ax, 136, Bx,
        136, 0., Cx, 136);
```

étape 1 - après modifications (H3LMS)

```
120 h3lms_blas_dgemm_sync(CblasColMajor,
        CblasNoTrans, CblasNoTrans, 136,
        1536*1536, 136, 1., Ax, 136, Bx,
        136, 0., Cx, 136);
```

FIGURE 7.7 – Modification de l'appel de la fonction *DGEMM* (bibliothèque basée sur *H3LMS*), étape 1.

Trois scénarii sont alors comparés :

$A1$ Séquentiel sur un cœur de calcul généraliste.

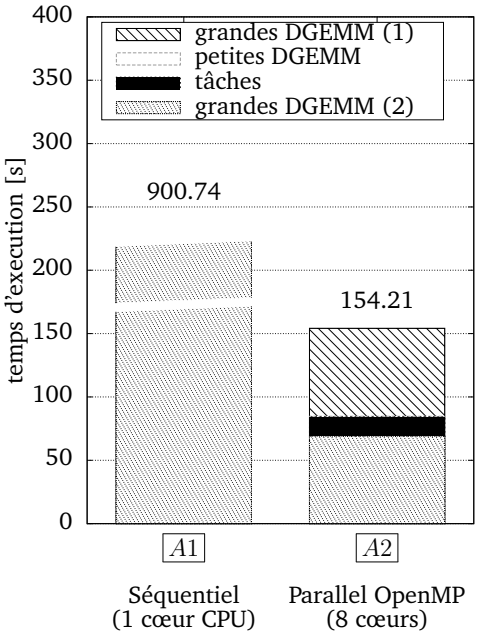
$A2$ Parallèle sur tous les cœurs généralistes disponibles.

$B1$ Hétérogène (théorique) en ne tenant pas compte des transferts de données.

$B2$ Hétérogène uniquement pour les grandes multiplications, homogène parallèle pour le reste. À l'issu de chaque étape hétérogène, les données sont évacuées des caches logiciels.

Les **figures 7.8a et 7.8b** présentent les résultats de ces premiers modes d'exécution. Cette simulation occupe environ 15 Go de mémoire. Les résultats comprennent les temps cumulés sur l'ensemble des pas de temps et pour chacune des étapes du calcul de la fonction *Flux numériques*. Seuls les meilleurs temps obtenus sur une vingtaine d'exécutions sont retenus. Le scénario purement homogène et parallèle ($A2$) apporte une accélération de $5{,}84\times$ par rapport à une exécution séquentielle. En employant la bibliothèque hétérogène de multiplication de matrices sur les étapes 1 et 4, les deux accélérateurs *Nvidia Telsa* sont sollicités en plus des 8 cœurs CPU (dont 2 dédiés à leur pilotage) et permettent de passer de 154,21 secondes à 84,29 secondes ($B2$). Les résultats du scénario $B1$, où les transferts ont été volontairement omis, révèlent que les performances sont fortement limitées par les mouvements de données. En effet, en mode hétérogène pour le scénario $B2$, la part de transfert non recouverte par du calcul représente 61% du temps d'exécution.

a) Exécutions sur ressources homogènes



b) Exécutions sur ressources hétérogènes

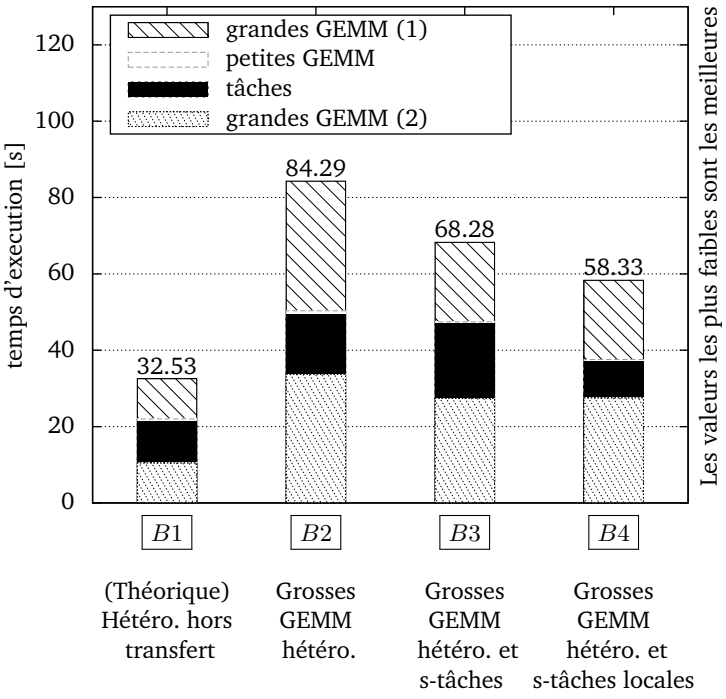


FIGURE 7.8 – Temps d'exécution de la fonction *flux numériques* sur un nœud de calcul *Tera-100 hétérogène* (cf. **annexe A.2**) avec un maillage 1536×1536 (double précision).

Portage des tâches de calcul sur accélérateurs

Les performances hétérogènes du programme PN sont fortement limitées par les transferts de données. En autorisant également une exécution hétérogène sur l'étape 3, des données (Ex et Ez) peuvent être conservées pendant toute la durée de vie d'un appel de la fonction *flux numériques*. Le nouveau scénario suivant est alors étudié :

$B3$ Hétérogène pour les grandes multiplications et les tâches de calcul définies à la main. Des données peuvent être conservées dans les caches logiciels jusqu'à la dernière étape.

Les appels à la bibliothèque hétérogène sont modifiés pour permettre de réutiliser et de conserver des données en caches logiciels après leur exécution (étape 1 et étape 4). Une instruction doit être renseignée à la fin de la fonction *flux numériques* afin d'évacuer ces données et les rendre cohérentes pour la suite des opérations qui ne sont pas gérées par le support exécutif *H3LMS*. La figure 7.9 illustre ces modifications pour l'étape 4 et la figure 7.8b présente les résultats obtenus avec le scénario $B3$. Ces modifications permettent d'apporter un gain en performance de 19% en passant de 84,29 à 68,28 secondes.

étape 4 - avant modifications

```
120 h3lms_blas_dgemm_sync(CblasColMajor,
        CblasNoTrans, CblasNoTrans, 136,
        1536*1536, 136, 1., Fx, 136, Ex,
        136, 0., Gx, 136);
```

étape 4 - après modifications (H3LMS)

```
120 h3lms_blas_dgemm(CblasColMajor,
        CblasNoTrans, CblasNoTrans, 136,
        1536*1536, 136, 1., Fx, 136, Ex,
        136, 0., Gx, 136);
121 h3lms_flush_data();
```

FIGURE 7.9 – Modification de l'appel de la fonction *DGEMM* (bibliothèque basée sur *H3LMS*), étape 4.

Adaptation des tâches de la troisième étape

L'étape 3 est constituée de plusieurs créations de tâches séparées par deux barrières de synchronisation. Autoriser un équilibrage de charge agressif peut mener à une plus grande quantité de données transférées pour les dernières sous-étapes. Les performances de l'étape 3 sont en réalité dirigées essentiellement par les mouvements de données et non pas par le calcul. Un dernier scénario est évalué :

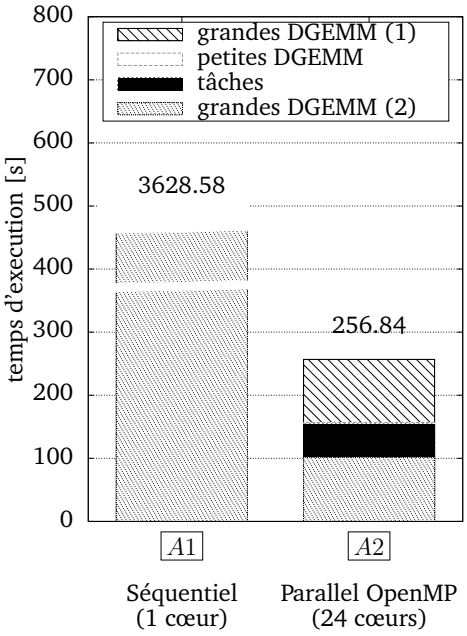
$B4$ Hétérogène pour les grandes multiplications et les tâches de calcul définies à la main. Ces dernières sont exécutées sur la ressource de calcul qui minimise les mouvements de données sans équilibrage de charge afin de réduire la quantité de transferts.

En conditionnant l'exécution de ces tâches uniquement par la localité, c'est-à-dire en empêchant un équilibrage de charge dynamique, un nouveau gain peut être constaté. La part de transferts est ainsi ramenée à 42%. Finalement, entre le scénario $A2$ et $B4$, une accélération de 2,64× est obtenue.

Évaluation sur un autre nœud de calcul

Les figures 7.10a et 7.10b présentent cette fois-ci les résultats des modes d'exécution avec un maillage 768×768 ($x = z = 768$) à partir du nœud de calcul *ENS-Fermi* (cf. annexe A.6). Cette simulation occupe environ 4 Go de mémoire.

a) Exécutions sur ressources homogènes



b) Exécutions sur ressources hétérogènes

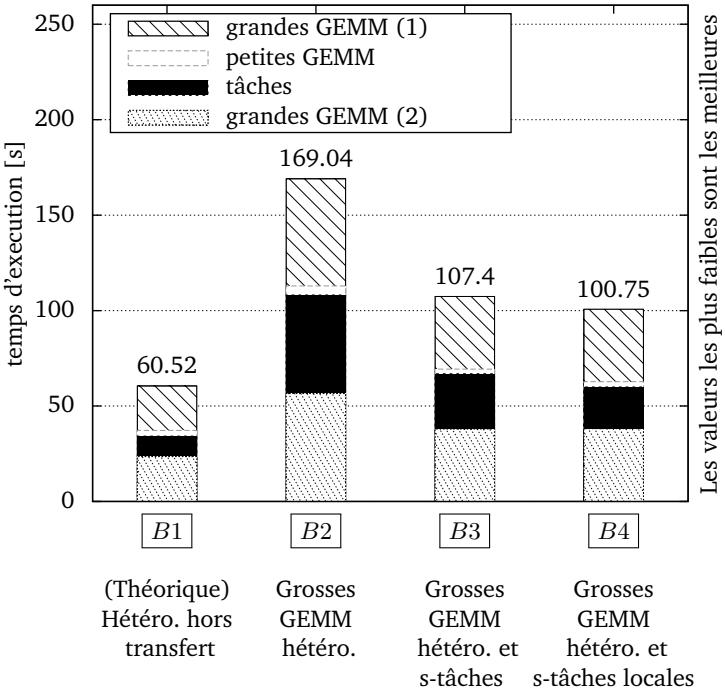


FIGURE 7.10 – Temps d'exécution de la fonction *flux numériques* sur la machine *ENS-Fermi* (cf. annexe A.6) avec un maillage 768×768 (double précision).

Les 24 cœurs de calcul permettent d'atteindre une accélération de $14{,}13\times$ en passant du scénario $A1$ à $A2$. Ajuster la façon dont les tâches de *l'étape 3* sont ordonnancées permet de passer le temps d'exécution de $107{,}4$ à $100{,}75$ secondes. Avec une telle dimension de maillage, la capacité de rétention des caches logiciels est accrue puisqu'ils peuvent se focaliser sur un jeu de données plus restreint. Par rapport au nœud de calcul de *Tera-100 hétérogène* (cf. annexe A.2), ce potentiel accentué semble contrebalancé par un rapport de performances plus faibles entre les processeurs généralistes et les accélérateurs. Enfin, entre le scénario $A2$ et $B4$, l'emploi des processeurs graphiques permet d'accélérer l'exécution de $2{,}55\times$.

7.2.3 Résultats multi nœuds de calcul

Cette sous-section vise à démontrer que recourir à la plate-forme ne perturbe pas le passage à l'échelle des applications avec un autre modèle de programmation tel qu'MPI. La figure 7.11 expose les performances obtenues en étudiant passage à l'échelle au sens faible (*weak scaling* chapitre 3.1.1, page 58) de la fonction *flux numériques*. Cette fonction ne contient pas de communication inter nœuds. 1 à 128 nœuds de calcul sont utilisés et les dimensions du maillage sont augmentées proportionnellement de manière à conserver la même quantité de calcul par nœud. Pour chaque scénario, la courbe est parfaitement plane ce qui confirme un passage à l'échelle au sens faible sans perte de performance pour la fonction étudiée.

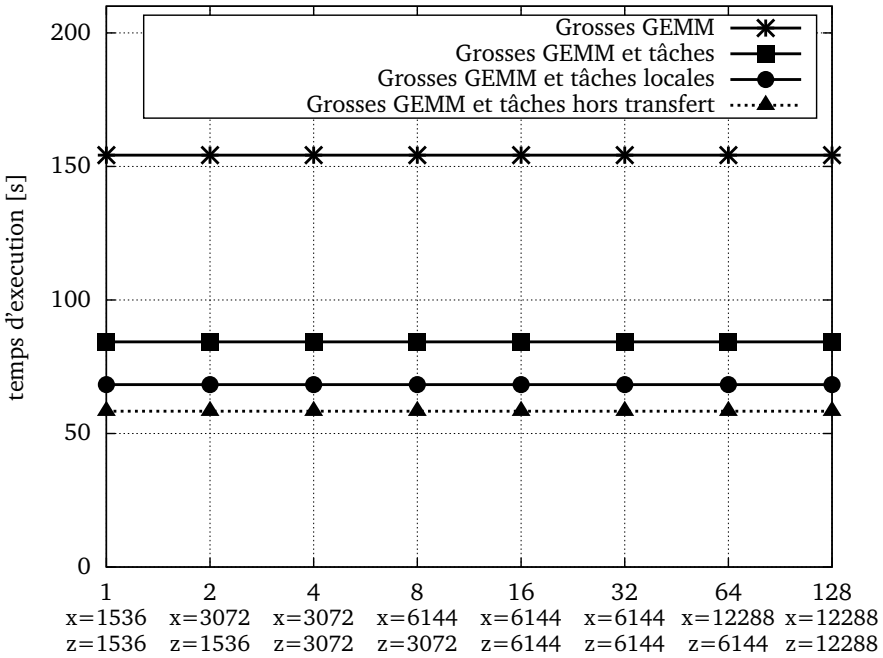


FIGURE 7.11 – Temps d'exécution de la fonction *flux numériques* en faisant varier le maillage et le nombre nœuds de calcul hétérogènes de *Tera-100 hétérogène* (cf. **annexe A.2**) (passage à l'échelle *faible*).

7.2.4 Discussion des résultats et améliorations possibles

À la lumière des ces résultats, les phases de calcul les plus intenses peuvent être accélérées par une adaptation hétérogène des opérations associées. Les autres phases de traitement sont alors conservées parallèles sur processeurs multi-cœurs. En ayant recours à une bibliothèque de calcul hétérogène pour les noyaux d'algèbre linéaire optimisés, le portage représente un faible coût de développement.

Les performances globales peuvent être néanmoins limitées par les transferts de données causés par des aller-retour entre les mémoires déportées et la mémoire centrale. Cette contrainte est en partie remédiée en portant sur toutes les architectures des noyaux de calcul qui ne profite pas d'accélération notable des calculs. Cela présente l'avantage de réduire la quantité de données véhiculées en permettant leur rétention dans les caches logiciels. Cette adaptation peut être réalisée dans un second temps, démontrant la possibilité d'un portage pas à pas vers une exécution hétérogène plus optimisée.

La quantité de transferts reste tout de même plus importante qu'espérée. Quelques transferts sont déclenchés par un équilibrage de charge trop agressif. Même si un tel choix d'ordonnancement accélère localement les traitements, le déplacement des données peut entraîner des dégradations de performances pour des traitements suivants. Il est ainsi important de différencier ces tâches particulières qui n'embarquent généralement pas une grande quantité de calculs. En forçant leur exécution par des unités qui permettent de minimiser les transferts de données, une nouvelle amélioration du temps de calcul peut être observée. La pression est désormais davantage placée sur la qualité des noyaux de calcul.

Des améliorations peuvent être envisagées, notamment pour faciliter la déclaration des tâches de calcul. Une gestion des dépendances de données permettrait d'alléger la syntaxe en évitant de positionner des instructions de synchronisation entre les tâches. Cette fonctionnalité n'empêcherait cependant pas de renseigner les instructions déclenchant l'évacuation les données contenues dans les caches logiciels.

En résumé, le portage du test de performance *HPL* a été réalisé en modifiant seulement trois lignes de code. La version à exécution homogène de la bibliothèque d'algèbre linéaire basée sur H3LMS est légèrement en retrait par rapport à version parallèle de la bibliothèque Intel MKL. L'exécution hétérogène, quant à elle, apporte une accélération de $7{,}8\times$ par rapport à la version parallèle homogène. Cependant, ces performances ne peuvent être aisément comparées aux résultats publiés par Fatica et al. puisque le nœud de calcul employé ne possède pas suffisamment de mémoire pour évaluer un cas test de dimension semblable.

La *mini-application* PN démontre qu'un portage progressif peut être envisagé. Dans un premier temps, un simple appel à la bibliothèque hétérogène apporte une accélération de $1{,}26\times$ sur la fonction *flux numériques* avec un nœud de calcul hétérogène du supercalculateur *Tera-100* comparé à une exécution n'exploitant que les cœurs des CPUs. Après avoir adapté des portions de code afin de conserver des données dans les caches logiciels entre les appels des multiplications de matrices, l'amélioration est portée à $1{,}83\times$. Enfin, en forçant une exécution minimisant les transferts de données pour certaines super-taches qui contiennent peu de d'opérations, l'accélération atteint les $2{,}64\times$.

Troisième partie

Synthèse et perspectives

Synthèse

Ce manuscrit a présenté des contributions permettant d'accentuer la localité des données dans des nœuds de calcul hétérogènes à partir d'un modèle de programmation par tâches de calcul. En particulier, la localité spatiale a été renforcée pour mieux exploiter la mémoire centrale, laquelle s'articule autour d'une imbrication de plus en plus marquée de diverses mémoires. La localité temporelle est quant à elle amplifiée en développant et en améliorant l'efficacité de caches logiciels déployés pour les accélérateurs matériels possédant leur propre mémoire.

La programmation par tâches de calcul apporte de surcroît des atouts indéniables. Elle permet un équilibrage de charge dynamique entre ressources de calcul hétérogènes à condition d'implémenter les noyaux d'exécution adaptés à chaque architecture. Les programmes ainsi élaborés peuvent plus facilement faire face à des traitements irréguliers et à la complexification des machines qui impliquent davantage d'imprévisibilités lors des phases de calculs. De plus, grâce à une sémantique adaptée, la déclaration des tâches de calcul permet également de recourir à des mécanismes de gestion automatisée de la majorité des opérations de cohérence entre les mémoires disjointes, et de décharger les développeurs de cette tâche fastidieuse et source d'erreurs.

Les contributions présentées tout au long de ce document ont été rassemblées dans la plate-forme d'exécution ***H3LMS***[126], laquelle s'appuie sur le module ***COMPAS***[127] afin d'aiguiller l'attribution des tâches vers les diverses unités de calcul en fonction de la localité de données. Elle permet un portage pas à pas, en se focalisant sur certaines portions des codes existants. Elle offre alors une solution plus réaliste en améliorant la productivité de ce processus d'évolution. La plate-forme ainsi élaborée a été finalement intégrée à un environnement de développement, lequel favorise un ordonnancement unifié de divers supports exécutifs associés à d'autres modèles de programmation, tout en réduisant les surcoûts liés à leur cohabitation.

Résumé des contributions

Certaines des ces contributions ont été publiées et présentées à MUTIPROG'12 [11].

— En termes d'amélioration de la localité spatiale, des **tâches de calcul décomposables** ajustent dynamiquement la charge de travail en fonction des unités de calcul ciblées. Les *super-tâches* facilitent les transferts groupés vers des mémoires déportées en renfermant une plus grande quantité d'opérations. Celles-ci peuvent être réparties à *grain fin* entre les unités de plus faible capacité de calcul, et permettre ainsi un équilibrage de charge plus adéquat. Cette approche offre ainsi une meilleure exploitation d'unités hétérogènes tout en profitant également de la puissance de calcul des unités les plus lentes. Plus de 20% de gain sont obtenus par rapport aux solutions de référence sur une multiplication de matrices avec des ressources de calcul hétérogènes.

[126] **H3LMS :** *Harnessing Hierarchy and Heterogeneity with Locality Management and Scheduling*

[127] **COMPAS :** *Coordinate and Organize Memory Placement and Allocation for Scheduler*.

— Une **coopération hiérarchique** des unités de traitement, s'appuyant sur les contraintes architecturales des machines actuelles, permet de limiter les accès distants. Elle est associée à un placement ainsi qu'à une décomposition des données adaptée pour mieux exploiter la localité spatiale. Cela permet d'accroître la localité spatiale dans un contexte hétérogène en limitant les effets d'accès non uniformes à la mémoire. Plus de 5% de gains supplémentaires s'ajoutent ainsi aux performances obtenues par l'approche à multi-granularité sur une multiplication de matrices.

— L'élaboration du module **COMPAS** permet de fixer une granularité de cohérence et de **répartir des données** en mémoire selon un schéma défini. Il réduit la quantité d'accès distants (NUMA) dans un nœud de calcul en aiguillant l'ordonnancement des tâches de calcul selon la localité des données.

— En matière de **gestion automatisée** de la cohérence des données, la décomposition en tâches offre plus de souplesse d'exécution. Les opérations ainsi morcelées impliquent de rapatrier progressivement les données dont la totalité ne pouvait être contenue dans des mémoires déportées. Une **rétention accrue des données** dans les mémoires des accélérateurs, couplée à un ordonnancement tenant compte de leur position, permet de réduire le volume de transferts. De plus, un mécanisme d'éviction étendant la politique *LRU* sert à mieux capturer le potentiel de réutilisation des données et à économiser 10% de la quantité de données véhiculées sur une factorisation LU dense. Sur un programme de factorisation LU creux, ces fonctionnalités permettent d'atteindre des performances supérieures à la somme de celles relevées individuellement pour chaque architecture (sur-linéaire hétérogène). Cela s'explique par la focalisation des accélérateurs sur des jeux de données plus réduits, ce qui améliore ainsi l'efficacité de leur cache logiciel en diminuant la quantité globale des données utiles pour chaque ressource de calcul.

— Une proposition d'interface autorise la **réutilisation de données préchargées** entre des appels à une bibliothèque de calculs optimisés et d'autres traitements. Une rétention de données dans les caches logiciels est donc possible et évite d'être contraint à évacuer toutes les données retenues cohérentes dans les caches logiciels. Cela passe par une mémorisation des granularités employées et par des mécanismes permettant de retrouver la cohérence de chaque bloc de données. Un gain atteignant 66% est obtenu en intégrant cette fonctionnalité à la bibliothèque *Magma-StarPU* sur un micro-test enchaînant plusieurs appels de multiplication hétérogènes.

— Une **double politique d'ordonnancement** permet d'adapter l'agressivité de l'équilibrage de charge en fonction du type de tâches. Une réduction de 15% du temps d'exécution est constatée en l'employant sur la mini-application *PN*. Une exécution minimisant les mouvements de données est ainsi imposée aux super-tâches qui comportent un faible ratio $\frac{\text{quantité de calculs}}{\text{transferts de données}}$.

— La **conception d'un support exécutif H3LMS** agrégeant les fonctionnalités précédentes et intégré à l'environnement de programmation *MPC*[128] facilite les interactions avec d'autres modèles de programmation tel que *MPI*. Cela permet de réduire davantage les lignes de codes à saisir dans un programme existant, en masquant les phases d'initialisation et de finalisation de la plate-forme au développeur. En ne rajoutant que trois lignes de code, deux accélérateurs sont exploités dans un contexte d'équilibrage de charge dynamique et hétérogène sur le noyau *DGEMM* du test *Linpack*. Une accélération de $7{,}8\times$ est atteinte sur l'ensemble du test par rapport à une exécution parallèle sans exploiter les deux GPUs associés aux nœuds de calcul employé.

[128] **MPC :** *Multi-Processor Computing*.

Perspectives

La poursuite des travaux présentés dans ce manuscrit permettrait d'enrichir les fonctionnalités de la plate-forme *H3LMS* selon différents axes, lesquels ouvriraient potentiellement la voie à de nouvelles contributions. Un première section aborde quelques pistes d'améliorations pouvant être envisagées à court terme. Puis, ce document se clôturera sur des considérations plus générales portant sur les futurs enjeux auxquels les supports d'exécution devront vraisemblablement répondre.

1 Pistes d'améliorations à court terme

Des évolutions de trois natures peuvent être entreprises. En premier lieu, un enrichissement du support exécutif servirait à supporter plusieurs types d'accélérateurs. Des améliorations contribueraient aussi à faciliter le portage de codes existants. Enfin, des perfectionnements permettraient d'accroître les performances et diminuer les surcoûts liés à l'ordonnancement des tâches de calcul.

1.1 Support étendu à plusieurs types d'accélérateurs

Seuls les accélérateurs compatibles avec le modèle de programmation *CUDA* ont été jusque-là supportés par la plate-forme *H3LMS*. Depuis 2010, les accélérateurs *GPUs Nvidia* représentent la majorité des processeurs spécialisés qui équipent les supercalculateurs du CEA. En effet, les processeurs *Cell* étaient en déclin après l'aveu de Davide Turek, vice président de *Exascale Systems* chez *IBM*, de la suspension des évolutions de la puce [257]. L'entreprise *AMD* n'a, quant à elle, commercialisé des cartes capables de répondre réellement aux besoins du calcul à haute performance qu'à partir de juin 2012, en embarquant des mémoires à correction d'erreurs avec les produits *AMD FirePro W8000* et *AMD FirePro W9000* [258]. Depuis la fin 2012, les accélérateurs de type *MIC* d'*Intel* (*Xeon Phi*) ont fait leur apparition en tant que concurrents de poids. Ainsi, en 2013, il est désormais pertinent d' étendre le support de *H3LMS* aux accélérateurs *AMD* et *Intel*.

La prise en charge du standard *OpenCL*, proche du modèle *CUDA* et termes de sémantique bien que plus verbeuse, permettrait d'étendre aisément le support aux accélérateurs déportés *MICs* et aux *GPUs AMD*. Plus généralement, cette extension ouvrirait la voie à une compatibilité théorique de tout autre accélérateur supportant la norme et pourrait profiter des mêmes contributions que celles détaillées dans ce manuscrit. Des utilisations plus avancées sont également envisageables avec un accélérateur *MIC*. Ils embarquent un système d'exploitation complet et peuvent donc fonctionner de manière autonome comme un nœud de calcul à part entière. Il est alors possible, en déployant *H3LMS* sur ce type d'accélérateur, de permettre un fonctionnement par tâches et super-taches de calcul. Une gestion automatisée des transferts avec la mémoire centrale peut s'appuyer sur l'interface de communication d'*Intel SCIF*[129] [259]. Des super-tâches contribueraient à amplifier la localité des données au sein des *Xeon Phi*. Ceux-ci ne présentent aucune contrainte d'accès non uniforme à leur mémoire embarquée (*NUMA*) puisqu'un réseau

[129] **SCIF :** *Symmetric Communications Interface*.

de mémoires locales interconnectées en anneaux masque ces effets. Néanmoins, chaque cœur peut manipuler simultanément quatre processus d'exécution matériels différents (*SMT*) afin d'améliorer la gestion des ressources de calcul vis à vis des accès à la mémoire (cf. **chapitre 2.1**, page 35). Une super-tâche pourrait donc générer des sous-tâches destinées aux multiples voies *SMT* d'un cœur de calcul afin d'optimiser la localité spatiale des mémoires locales communes. Un nœud de calcul hétérogène composé de *CPUs* et de plusieurs accélérateurs *Intel* pourrait alors exécuter des super-tâches à deux niveaux de granularité via des instances d'*H3LMS* imbriquées. Une instance principale serait déployée pour la gestion hétérogène des *CPUs* et des *MICs*, et d'autres, secondaires, seraient destinées à manipuler une double granularité au sein de chaque accélérateur.

1.2 Aide au portage de codes existants

L'exploitation de ressources de calcul hétérogènes est complexe et l'adaptation de codes existants peut demander des efforts considérables. La plate-forme *H3LMS* atténue la quantité de travail des développeurs en se chargeant de divers mécanismes tels que l'équilibrage de charge dynamique et hétérogène, le maintien de la cohérence, et la rétention de blocs de données dans des caches logiciels. D'autres améliorations permettraient de faciliter davantage le portage de codes de calcul existants. En particulier, la sémantique des mécanismes associés aux tâches de calcul pour architectures hétérogènes peut être allégée par les mécanismes détaillées dans cette sous-section.

Dépendances de données

Étendre la plate-forme *H3LMS* afin de résoudre les dépendances de données dynamiquement permettrait de limiter le recours à des barrières de synchronisation. Employée dans *StarPU*, *XKaapi* ou *StarSs*, cette fonctionnalité peut améliorer l'équilibrage de charge dans certains cas de figure. Elle accroît le degré de parallélisme au cours de l'exécution en générant des tâches dès qu'elles sont prêtes à être exécutées. Par exemple, pour la factorisation LU de matrices par blocs (cf. **chapitre 3.4.1**, page 69), des tâches de l'étape 3 pourraient s'exécuter en même temps que des tâches de l'étape 2 issues d'un autre pas d'itération. L'intégration de cette fonctionnalité dans *H3LMS* suppose toutefois d'établir un compromis sur la granularité de la libération des tâches lorsque les dépendances sont résolues. En effet, des mécanismes présentés dans ce manuscrit reposent sur une approche *synchrone par lot*, laquelle facilite une prise de décision plus efficace lorsqu'une quantité suffisante de tâches est considérée. En particulier, l'indice d'affinité permet de séparer les super-tâches dans les listes partagées pour accentuer la localité temporelle des caches logiciels associés aux mémoires déportées (cf. **chapitre 5.2.3**, page 108). Rendre accessible trop tôt des super-tâches rajoutées au fur et à mesure dans une liste partagée ne permettrait pas d'aboutir à une aussi bonne séparation qu'une liste triée par lots.

Mémoires déportées partagées virtuellement

Gelado et al. présentent GMAC[130] [260], une solution de gestion logicielle de la cohérence des données pour des mémoires déportées. Celle-ci donne l'illusion que toutes les mémoires sont associées à un même espace partagé (*DSM*[131]). Il s'agit dans ce cas précis d'une version asymétrique[132] où la cohérence n'est assurée que lorsque les *CPUs* accèdent aux données. La récupération des données par les accélérateurs requiert toujours d'initier explicitement des transferts. Couplé à un support exécutif tel que *H3LMS*, les développeurs ne seraient plus contraints de déclencher les mouvements de données vers les mémoires déportées. Le recours à une *ADSM* présenterait un atout encore plus intéressant puisqu'il ne serait plus nécessaire d'évacuer systématiquement les données conservées dans les caches logiciels avant que le programme n'accède

[130] **GMAC :** *Global Memory for ACcelerators*.
[131] **DSM :** *Distributed Shared Memory*.
[132] **ADSM :** *Asymmetric Distributed Shared Memory*.

à des phases de calcul qui ne seraient plus gérées par la plate-forme. La cohérence des données est effectuée de manière tardive lorsque les blocs sont modifiés en mémoire centrale. De plus, cette gestion permettrait de collecter des informations utiles aux développeurs pour identifier les portions de code qui amplifieraient la rétention des données si elles étaient déportées sur les accélérateurs. Le principal intérêt de cette approche est de faciliter l'adaptation de codes existants en ne se souciant plus du tout de la cohérence des données. Les performances de telles approches sont néanmoins sujettes à controverses. Elles reposent sur des mécanismes de protection de pages mémoire qui nécessitent l'intervention du noyau du système d'exploitation. Ils peuvent sévèrement dégrader les performances lorsqu'ils sont sollicités trop fréquemment. En particulier, l'accès à n'importe quel bloc de données situés sur la même page mémoire qu'un bloc non cohérent déclencherait une interruption. Employer ces mécanismes avec les fonctionnalités de répartition de pages et de granulartiés appropriées des blocs de cohérence contribuerait à diminuer le taux de fausses alertes. Ainsi, une telle approche pourrait, dans certains cas, profiter d'accélérations grâce à une rétention de données accrue dans les caches logiciels pendant toute la durée de vie de l'application.

Déclaration par directive

Implémenter les directives proposées dans le **chapitre 6** (page 113) servirait à accroître la productivité en allégeant la quantité de code à renseigner. Les appels à l'interface de programmation du support exécutif seraient insérés à la compilation au moyen, par exemple, d'un greffon du compilateur *GCC*[133] [261]. De plus, lorsque les directives sont ignorées le fonctionnement séquentiel est toujours assuré de manière similaire à *OpenMP*, contribuant ainsi à faciliter un portage progressif des codes existants et leur débogage. Des directives de normes émergentes telles que *OpenMP 4.0* et *OpenAcc* (cf. **chapitre 3.2.4**, page 63) pourraient également être supportées en apportant éventuellement des extensions afin d'être en mesure d'opérer des exécutions à équilibrages de charge hétérogènes.

1.3 Amélioration des performances

Interaction avec d'autres modèles de programmation

Grâce à l'intégration dans *MPC*, une collaboration entre les supports exécutifs de plusieurs modèles de programmation peut être établie. Les possibilités de ces interactions seront étudiées, en particulier, lorsque les modèles *MPI* et *OpenMP* sont combinés à *H3LMS*. Ce dernier offre une gestion globale et unifiée des ressources de calcul pour le modèle de programmation *MPI* (cf. **chapitre 6.5**, page 127). Ainsi, lorsque des tâches de calcul sont générées depuis une instance *MPI*, elles peuvent être traitées par plusieurs unités logiques dont des unités associées à des accélérateurs. En particulier, certaines copies de données plus ou moins conséquentes qui sont induites par l'envoi et la réception de messages peuvent être accélérées si elle sont décomposées en plusieurs tâches. Elles profiteraient ainsi de la bande passante combinée de plusieurs cœurs de calcul (cf. **chapitre 2.3.2**, page 46). Les super-tâches peuvent également bénéficier de plusieurs instances *MPI*. En effet, un tel mode de fonctionnement multi producteurs éviterait de renseigner les informations parfois contraignantes mais nécessaires à la répartition des pages mémoires. Leur placement serait forcé sur le nœud NUMA de l'instance *MPI* associée, allégeant ainsi la sémantique décrite dans le **chapitre 6** (page 113).

Caches logiciels pour CPUs

Pour le moment, les caches logiciels sont employés qu'avec des accélérateurs matériels. Étendre leur recours aux CPUs permettrait de renforcer davantage la localité, au prix d'une plus ou moins grande augmentation de la consommation mémoire qui serait causée par la

[133] **GCC :** *GCC, the GNU Compiler Collection*.

duplication de certains blocs de données. Cette réplication présenterait l'intérêt de rapprocher les blocs les plus utilisés en réduisant théoriquement à la fois les effets NUMA et les effets NUIOA. En particulier, cette fonctionnalité permettrait d'affiner la localité NUMA à une granularité plus fine qu'une taille de page mémoire. L'association de tels caches logiciels à des pôles de la représentation abstraite de la machine cible pourrait alors réduire les mouvements de données entre les CPUs. Dans un contexte mono producteur, le placement initial des pages mémoires deviendrait moins critique.

2 Réflexion finale sur les futurs enjeux des supports d'exécution

Les plate-formes d'exécution glanent des informations d'ordre algorithmique, et relatives à la cohérence de données et à la manière dont elles sont accédées. Ces renseignements précieux pourraient être utiles pour agir de manière adéquate lors d'un dysfonctionnement matériel ou pour optimiser de façon générale la consommation électrique d'un système complet en minimisant les déplacements de données.

Le *temps moyen entre pannes* (MTBF[134]) indique la fiabilité des composants ou d'un système complet. Il a été démontré que le taux de panne était davantage corrélé à la taille de la machine qu'à la technologie choisie [262]. Avec les prochaines générations de machines, la probabilité qu'une défaillance matérielle survienne lors d'un long calcul est accrue. Par exemple, le test de performance *Linpack* nécessite environ une trentaine d'heures pour évaluer les performances d'une machine *pétaflopique*. A l'échelle *exaflopique*, cette évaluation s'étalerait sur quasiment une semaine. Sans mécanisme permettant de réagir dynamiquement suite à un dysfonctionnement matériel, les chances d'aboutir à une résultat s'amenuisent.

Les supercalculateurs devancent considérablement les capacités de calcul d'un être humain et les progrès se poursuivent pour fournir toujours plus de performance. Le projet HPB[135] [263] est un consortium de recherche autour de la neuroscience, financé à hauteur d'un milliard d'euros par l'union européenne sur une période de 10 ans à partir de 2013. Il ambitionne de simuler un cerveau humain en temps réel afin d'en saisir certaines complexités pour déboucher sur des avancées dans le domaine médical, mais également d'ouvrir la voie à de nouvelles ressources de calcul pour le champ du calcul à haute performance [264]. Henry Markram, directeur du projet à l'EPFL[136], estime que les machines *exaflopiques* pourront répondre à ce problème de simulation [265]. Cette imitation numérique nécessite une très grande capacité de calcul et un surcoût d'adaptation pour approcher le fonctionnement complexe d'un cerveau humain. Les prévisions les plus optimistes tablent sur une consommation de 20 à 30 MW d'ici 2018 pour les premières machines *exaflopiques* [266]. Les cerveaux, quant à eux, sont des systèmes hautement parallèles qui peuvent se reconfigurer dynamiquement et recourir à des mécanismes de transductions pour véhiculer des informations. Ils ne consommeraient qu'entre 20 à 40 Watts ce qui ferait d'eux des systèmes un million de fois plus énergiquement efficace que les ressources nécessaires pour les simuler. Pour faire écho au paragraphe introductif de cette thèse, les transistors qui composent en quantité faramineuse chaque machine ont bel et bien dépassé la quantité de neurones contenu dans un cerveau humain. Mais des marges de progrès considérables sont envisageables pour une gestion plus efficace de leur énergie. Ces efforts pourraient être en partie fournis par des plate-formes d'exécution capables de s'adapter dynamiquement, afin de minimiser les déplacements de données et de limiter la surconsommation induite.

[134] **MTBF :** *MeanTime Between Failure*.
[135] **HBP :** *Human Brain Project*.
[136] **EPFL :** *École Polytechnique Fédérale de Lausanne*.

Annexe A

Détails techniques des nœuds de calcul

Inti hétérogène

Tera-100 hétérogène

Curie hétérogène

Curie large

Cirrus hétérogène

ENS-Fermi

Station hétérogène

A.1 INTI HÉTÉROGÈNE

Inti est un prototype du supercalculateur du CEA-DAM Tera-100 installé en 2010. Sa partition hétérogène comprend 8 nœuds de calcul embarquant chacun **24 Go** de mémoire centrale, deux processeurs **Intel Xeon Nehalem quadri-cœurs** et deux accélérateurs **Nvidia Tesla M2050**.

Logiciel

O.S.	Bullx Linux 6.1
Compliateurs	GCC 4.5 - ICC 13.0
Nvidia CUDA	4.1
Intel MKL	13.10
Nvidia CUBLAS	4.1

Matériel

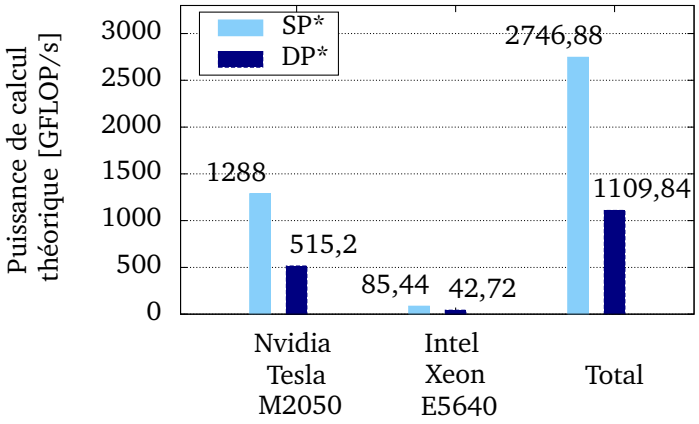
Système

Proc. généralistes	2× CPUs
Cœurs généralistes	8
Accélérateurs	2× GPUs
Mémoire centrale	24 Go DDR3
Nœuds NUMA	2
Réseau	Infiniband

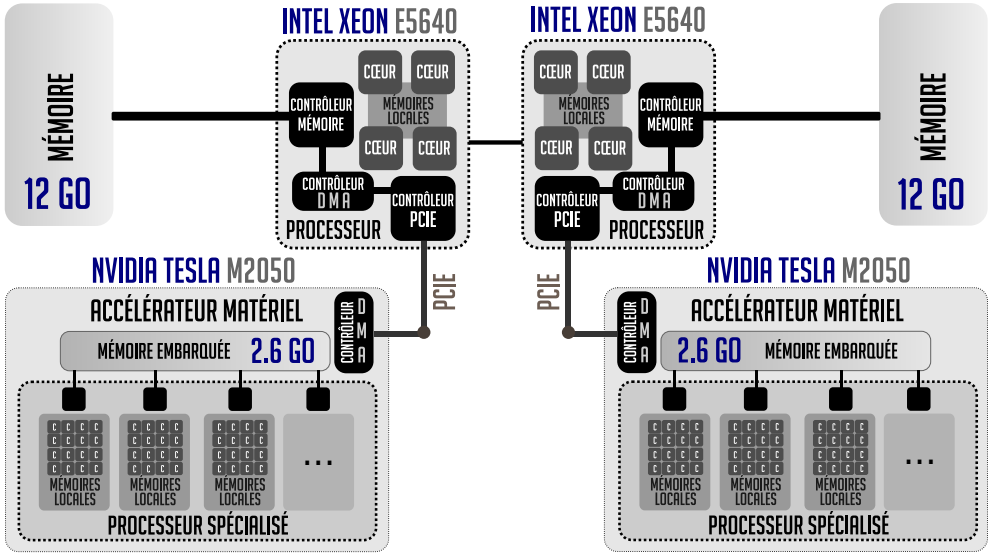
Processeurs

	CPU	GPU
Marque	*Intel*	*Nvidia*
Modèle	Xeon E5640	Tesla M2050
Architecture	Nehalem	Fermi
Date de sortie	2010	2010
Cœurs	4	448
Fréquence	2,67 GHz	1,15 GHz
Cache partagé	L3 (12 Mo)	L2 (768 Ko)
Mémoire associée	12 Go DDR3	2,6 Go GDDR5

Performances



***DP** : Double Précision, **SP** : Simple Précision



A.2 TERA-100 HÉTÉROGÈNE

Tera-100 est un supercalculateur déployé par la société Bull pour le CEA-DAM. En novembre 2010, il est classé 6e au *TOP500* en atteignant une puissance de traitement de 1,05 PFLOP/s. Sa partition hétérogène, installée en 2011, comprend 192 nœuds de calcul embarquant chacun **24 Go** de mémoire centrale, deux processeurs **Intel Xeon Nehalem quadri-cœurs** et deux accélérateurs **Nvidia Tesla M2090**.

Logiciel

O.S.	Bullx Linux 6.1
Compliateurs	GCC 4.5 - ICC 13.0
Nvidia CUDA	4.1
Intel MKL	13.10
Nvidia CUBLAS	4.1

Matériel

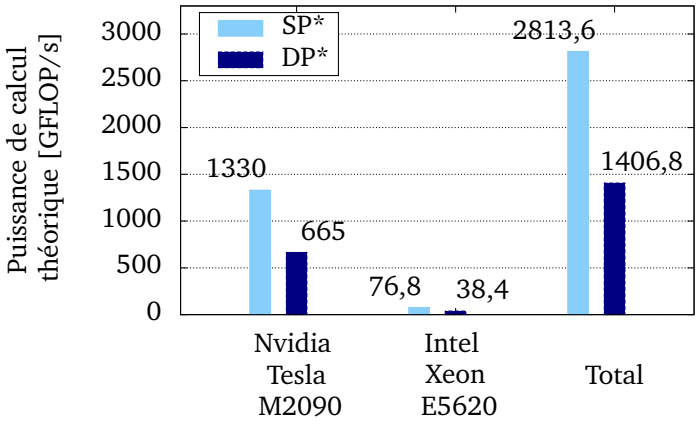
Système

Proc. généralistes	2× CPUs
Cœurs généralistes	8
Accélérateurs	2× GPUs
Mémoire centrale	24 Go DDR3
Nœuds NUMA	2
Réseau	Infiniband

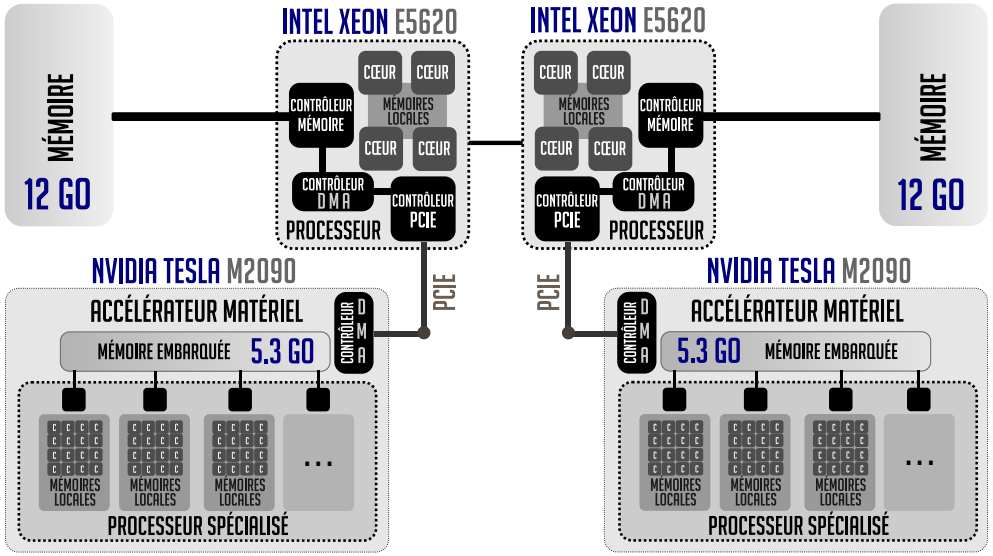
Processeurs

	CPU	GPU
Marque	*Intel*	*Nvidia*
Modèle	Xeon E5620	Tesla M2090
Architecture	Nehalem	Fermi
Date de sortie	2010	2011
Cœurs	4	512
Fréquence	2,40 GHz	1,30 GHz
Cache partagé	L3 (12 Mo)	L2 (768 Ko)
Mémoire associée	12 Go DDR3	5,25 Go GDDR5

Performances



***DP** : Double Précision, **SP** : Simple Précision



A.3 CURIE HÉTÉROGÈNE

Curie est un supercalculateur déployé par la société Bull pour GENCI au TGCC. En novembre 2012, il est classé 9e au *TOP500* en atteignant une puissance de traitement de 1,359 PFLOP/s. Sa partition hétérogène comprend 144 nœuds de calcul embarquant chacun **24 Go** de mémoire centrale, deux processeurs **Intel Xeon Nehalem quadri-cœurs** et deux accélérateurs **Nvidia Tesla M2090**.

Logiciel

O.S.	Bullx Linux 6.1
Compliateurs	GCC 4.5 - ICC 13.0
Nvidia CUDA	4.1
Intel MKL	13.10
Nvidia CUBLAS	4.1

Matériel

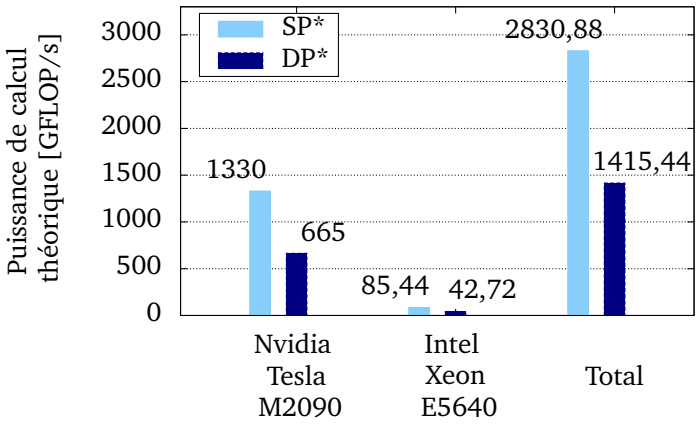
Système

Proc. généralistes	2× CPUs
Cœurs généralistes	8
Accélérateurs	2× GPUs
Mémoire centrale	24 Go DDR3
Nœuds NUMA	2
Réseau	Infiniband

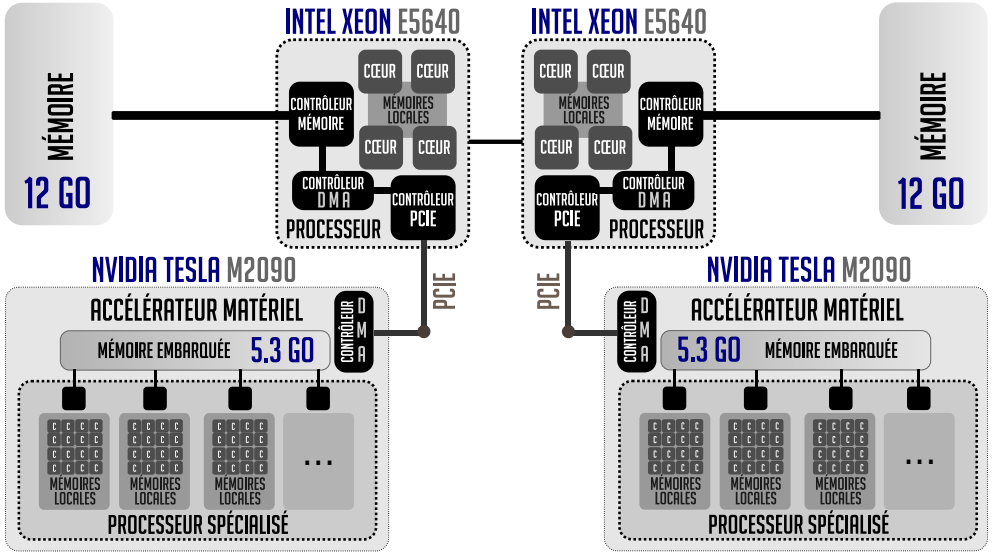
Processeurs

	CPU	GPU
Marque	*Intel*	*Nvidia*
Modèle	Xeon E5640	Tesla M2090
Architecture	Nehalem	Fermi
Date de sortie	2010	2011
Cœurs	4	512
Fréquence	2,67 GHz	1,30 GHz
Cache partagé	L3 (12 Mo)	L2 (768 Ko)
Mémoire associée	12 Go DDR3	5,25 Go GDDR5

Performances



***DP** : Double Précision, **SP** : Simple Précision



A.4 CURIE LARGE

Curie est un supercalculateur déployé par la société Bull pour GENCI au TGCC. En novembre 2012, il est classé 9e au *TOP500* en atteignant une puissance de traitement de 1,359 PFLOP/s. Sa partition *large* est constituée de 90 nœuds de calcul *Bull Novascale Bullion* élaborés autour d'un assemblage de 4 cartes-mères grâce à l'interconnexion *Bull Coherence System*. Chaque nœud de calcul embarque **512 Go** de mémoire centrale et seize processeurs **Intel Xeon Nehalem octo-cœurs**.

Logiciel

O.S.	Bullx Linux 6.1
Compliateurs	GCC 4.5 - ICC 13.0
Intel MKL	13.10

Matériel

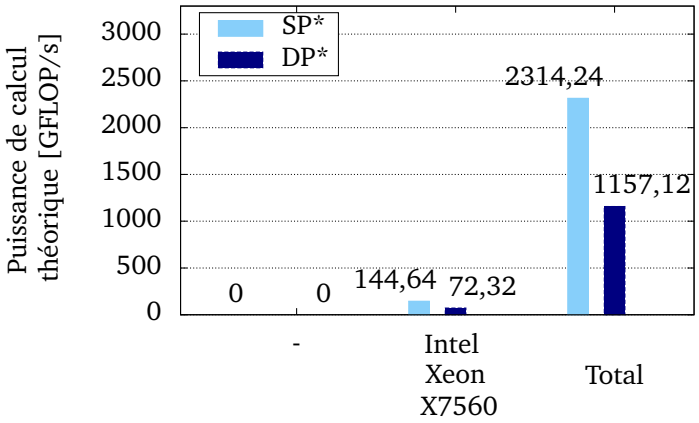
Système

Proc. généralistes	16× CPUs
Cœurs généralistes	128
Accélérateurs	-
Mémoire centrale	512 Go DDR3
Nœuds NUMA	16
Réseau	Infiniband

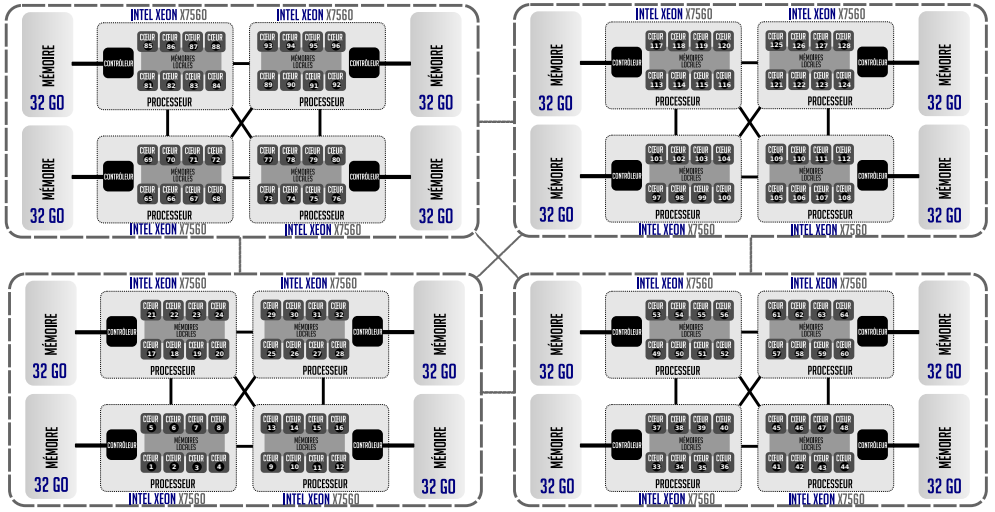
Processeurs

	CPU	GPU
Marque	*Intel*	-
Modèle	Xeon X7560	-
Architecture	Nehalem	-
Date de sortie	2010	-
Cœurs	8	-
Fréquence	2,26 GHz	-
Cache partagé	L3 (24 Mo)	-
Mémoire associée	32 Go DDR3	-

Performances



*__DP__ : Double Précision, __SP__ : Simple Précision



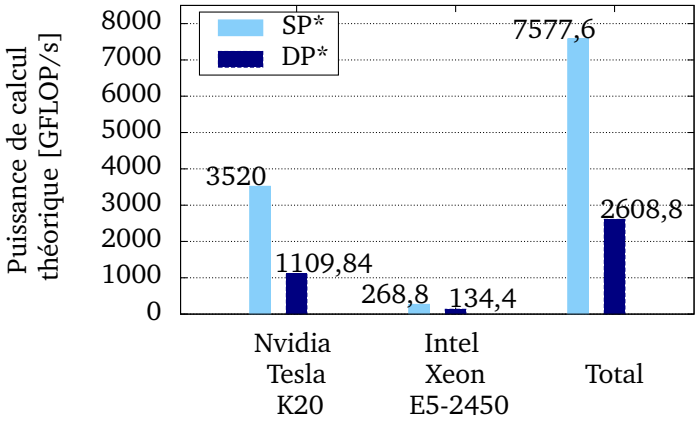
A.5 CIRRUS HÉTÉROGÈNE

Cirrus est un prototype installé pour les besoins du projet PERFCLOUD. Ce dernier est un projet coopératif issu du Fond pour la Société Numérique. Il associe partenaires industriels et académiques afin de concevoir, puis mettre à disposition du marché, les briques de base nécessaires aux nouvelles générations de centres de calcul pour le HPC. Sa partition hétérogène GPUs comprend 2 nœuds de calcul embarquant chacun **48 Go** de mémoire centrale, deux processeurs **Intel Xeon Nehalem octo-cœurs** et deux accélérateurs **Nvidia Tesla K20**.

Logiciel

O.S.	Bullx Linux 6.1
Compliateurs	GCC 4.6 - ICC 13.1
Nvidia CUDA	5.0
Intel MKL	13.1
Nvidia CUBLAS	5.0

Performances



***DP** : Double Précision, **SP** : Simple Précision

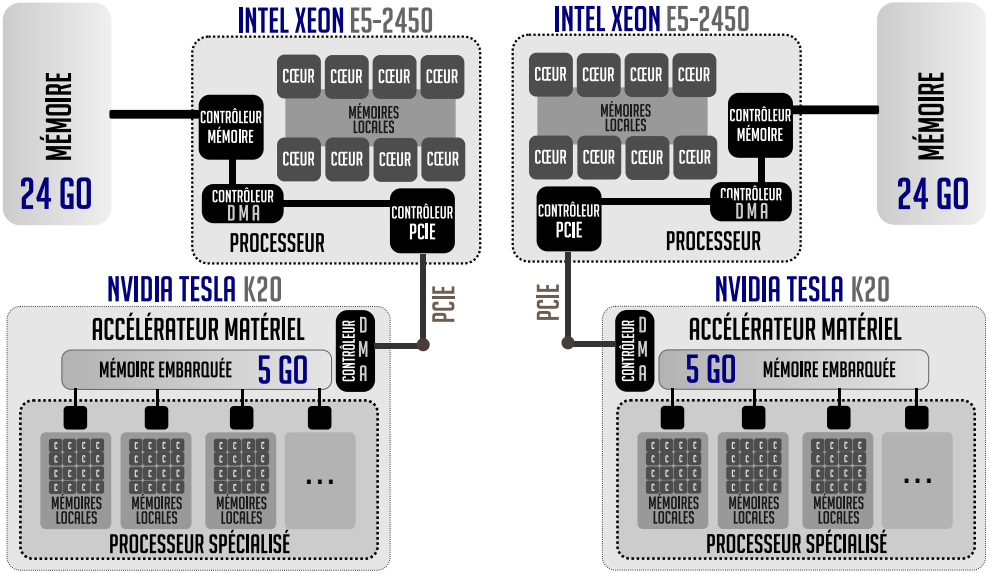
Matériel

Système

Proc. généralistes	2× CPUs
Cœurs généralistes	8
Accélérateurs	2× GPUs
Mémoire centrale	48 Go DDR3
Nœuds NUMA	2
Réseau	Infiniband

Processeurs

	CPU	GPU
Marque	*Intel*	*Nvidia*
Modèle	Xeon E5-2450	Tesla K20
Architecture	Sandy Bridge	Kepler
Date de sortie	2012	2012
Cœurs	8	2 496
Fréquence	2,10 GHz	745 MHz
Cache partagé	L3 (20 Mo)	L2 (1,25 Mo)
Mémoire associée	24 Go DDR3	5 Go GDDR5



A.6 ENS-FERMI

ENS-FERMI est un nœud de calcul embarquant **16 Go** de mémoire centrale, deux **AMD Opteron dodéca-cœurs**, chacun associés à deux niveaux d'accès NUMA, et deux accélérateurs **Nvidia Geforce GTX 470** connectés aux même nœud NUIOA.

Logiciel

O.S.	Ubuntu 11.04
Compliateurs	GCC 4.6 - ICC 10
Nvidia CUDA	4.0
Intel MKL	10
Nvidia CUBLAS	4.0

Matériel

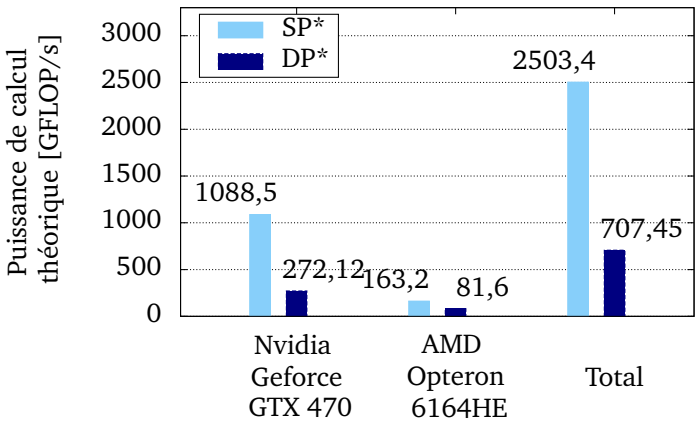
Système

Proc. généralistes	2× CPUs
Cœurs généralistes	24
Accélérateurs	2× GPUs
Mémoire centrale	16 Go DDR3
Nœuds NUMA	2 (+2)
Réseau	Infiniband

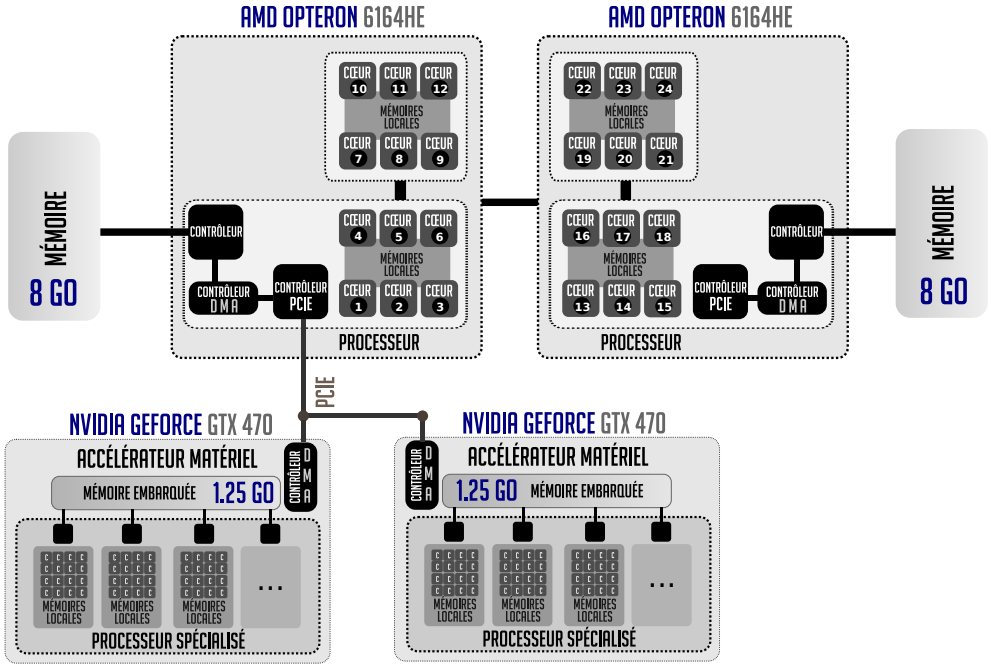
Processeurs

	CPU	GPU
Marque	*Intel*	*Nvidia*
Modèle	Opteron 6164HE	Geforce GTX 470
Architecture	Magny Cours	Fermi
Date de sortie	2010	2010
Cœurs	12	448
Fréquence	1,7 GHz	1,215 GHz
Cache partagé	L3 (12 Mo)	L2 (768 Ko)
Mémoire associée	8 Go DDR3	1,25 Go GDDR5

Performances



*DP : Double Précision, SP : Simple Précision



A.7 STATION HÉTÉROGÈNE

La station hétérogène embarque **16 Go** de mémoire centrale, deux processeurs **Intel Xeon quadri-cœurs** et deux accélérateurs **Nvidia Geforce GTX 480**.

Logiciel

O.S.	Ubuntu 11.04
Compliateurs	GCC 4.6 - ICC 10
Nvidia CUDA	4.0
Intel MKL	10.2.6
Nvidia CUBLAS	4.0

Matériel

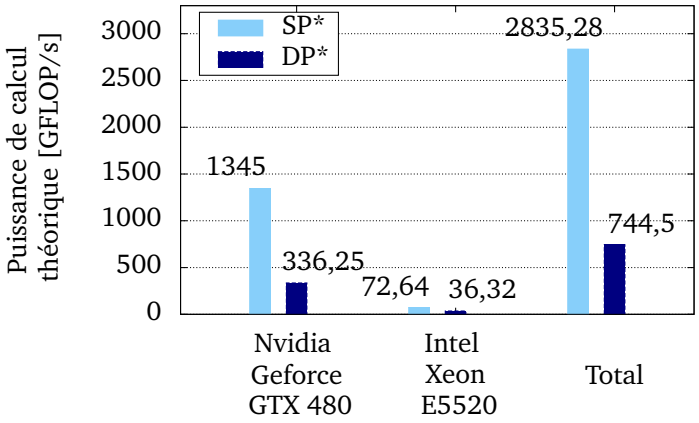
Système

Proc. généralistes	2× CPUs
Cœurs généralistes	8
Accélérateurs	2× GPUs
Mémoire centrale	16 Go DDR3
Nœuds NUMA	2
Réseau	-

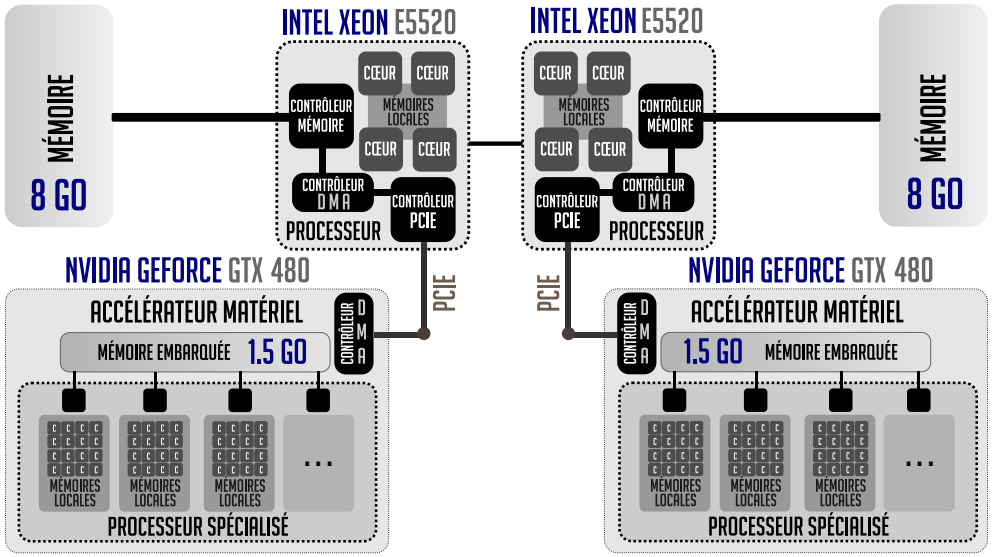
Processeurs

	CPU	GPU
Marque	*Intel*	*Nvidia*
Modèle	Xeon E5520	Geforce GTX 480
Architecture	Nehalem	Fermi
Date de sortie	2009	2010
Cœurs	4	480
Fréquence	2,27 GHz	1,4 GHz
Cache partagé	L3 (8 Mo)	L2 (768 Ko)
Mémoire associée	8 Go DDR3	1,5 Go GDDR5

Performances



***DP** : Double Précision, **SP** : Simple Précision



Bibliographie

[1] Murray Campbell, A Joseph Hoane Jr, and Feng-hsiung Hsu. Deep blue. *Artificial intelligence*, 134(1) :57–83, 2002. pages 7

[2] Oak Ridge Claims No. 1 Position on Latest TOP500 List with Titan, November 12, 2012. http://www.top500.org/blog/lists/2012/11/press-release. pages 7

[3] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965. pages 7

[4] G. Mourlevat. *Les machines arithmétiques de Blaise Pascal*. Mémoires de l'Académie des Sciences, Belles Lettres et Arts de Clermont-Ferrand : Académie des Sciences, Belles-Lettres et Arts. La Française d'Edition et d'Imprimerie, 1988. pages 7

[5] D. Swade. *Charles Babbage and his calculating engines*. Science Museum, 1991. pages 7

[6] Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42 :230–265, 1936. pages 7

[7] R. Rojas. Konrad zuse's legacy : the architecture of the z1 and z3. *Annals of the History of Computing, IEEE*, 19(2) :5–16, 1997. pages 7

[8] Ieee standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–58, 2008. pages 7

[9] *The Computer from Pascal to von Neumann*. Princeton University Press, 2008. pages 8

[10] G. Gallavotti. *The Fermi-Pasta-Ulam Problem : A Status Report*. Lecture Notes in Physics. Springer, 2008. pages 8

[11] Jean-Yves Vet, Patrick Carribault, and Albert Cohen. Multigrain affinity for heterogeneous work stealing. In *MULTIPROG-2012*, 2012. pages 9, 149

[12] TOP500. Top500 Supercomputing Sites. http://www.top500.org. pages 17, 26

[13] Jack J Dongarra. *LINPACK users' guide*. Number 8. Siam, 1979. pages 17

[14] ELEAnoR Chu and ALAn GEoRGE. Gaussian elimination with partial pivoting and load balancing on a multiprocessor. *Parallel Computing*, 5(1) :65–74, 1987. pages 17

[15] T.E. Anderson, D.E. Culler, and D.A. Patterson. A case for now (networks of workstations). *Micro, IEEE*, 15(1) :54–64, 1995. pages 18

[16] Thomas Sterling, Donald J. Becker, Daniel Savarese, John E. Dorband, Udaya A. Ranawake, and Charles V. Packer. Beowulf : A parallel workstation for scientific computation. In *In Proceedings of the 24th International Conference on Parallel Processing*, pages 11–14. CRC Press, 1995. pages 18

[17] DEUS consortium – Dark Energy Universe Simulations. http://www.deus-consortium.org. pages 19

[18] Le Calcul Haute Performance, La Recherche N393, page 12, janvier 2006. pages 19

[19] Michael J Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, 100(9) :948–960, 1972. pages 20

[20] FH Sumner, G Haley, and ECY Chen. The central control unit of the" atlas" computer. In *IFIP Congress*, pages 657–663, 1962. pages 20

[21] Donald MacKenzie. The influence of the los alamos and livermore national laboratories on the development of supercomputing. *Annals of the History of Computing*, 13(2) :179–201, 1991. pages 20

[22] CORE MEMORY. Parallel operation in the control data 6600. *Readings in computer architecture*, page 32, 2000. pages 20

[23] Roger Espasa, Mateo Valero, and James E Smith. Vector architectures : past, present and future. In *Proceedings of the 12th international conference on Supercomputing*, pages 425–432. ACM, 1998. pages 20

[24] Charles J. Purcell. The control data star-100 : performance measurements. In *Proceedings of the May 6-10, 1974, national computer conference and exposition*, AFIPS '74, pages 385–387, New York, NY, USA, 1974. ACM. pages 20

[25] W. J. Watson. The ti asc : a highly modular and flexible super computer architecture. In *Proceedings of the December 5-7, 1972, fall joint computer conference, part I*, AFIPS '72 (Fall, part I), pages 221–228, New York, NY, USA, 1972. ACM. pages 20

[26] Richard M Russell. The cray-1 computer system. *Communications of the ACM*, 21(1) :63–72, 1978. pages 20

[27] George H Goble and Michael H Marsh. A dual processor vax 11/780. In *ACM SIGARCH Computer Architecture News*, volume 10, pages 291–298. IEEE Computer Society Press, 1982. pages 21

[28] John L Larson. Multitasking on the cray x-mp-2 multiprocessor. *Computer*, 17(7) :62–69, 1984. pages 21

[29] W Daniel Hillis and Lewis W Tucker. The cm-5 connection machine : a scalable supercomputer. *Communications of the ACM*, 36(11) :31–40, 1993. pages 21

[30] Wu-chun Feng, M Warren, and Eric Weigle. The bladed beowulf : A cost-effective alternative to traditional beowulfs. In *Cluster Computing, 2002. Proceedings. 2002 IEEE International Conference on*, pages 245–254. IEEE, 2002. pages 21

[31] Robert H Dennard, Fritz H Gaensslen, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *Solid-State Circuits, IEEE Journal of*, 9(5) :256–268, 1974. pages 22

[32] David Lammers. Intel cancels tejas, moves to dual-core designs. *EETimes, May 7th*, 2004. pages 23

[33] Daniel L Slotnick, W Carl Borck, and Robert C McReynolds. The solomon computer. In *Proceedings of the December 4-6, 1962, fall joint computer conference*, pages 97–107. ACM, 1962. pages 23

[34] CPU, GPU and MIC hardware characteristics over time. http://www.karlrupp.net/blog/. pages 24, 47, 57

[35] Cray inc. cray XD1 FPGA Development. http://www.cray.com. pages 24

[36] Ling Zhuo and Viktor K Prasanna. High performance linear algebra operations on reconfigurable systems. In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, pages 2–2. IEEE, 2005. pages 24

[37] Robert Baxter, Stephen Booth, Mark Bull, Geoff Cawood, James Perry, Mark Parsons, Alan Simpson, Arthur Trew, Andrew McCormick, Graham Smart, et al. Maxwell a 64 fpga supercomputer. In *Adaptive Hardware and Systems, 2007. AHS 2007. Second NASA/ESA Conference on*, pages 287–294. IEEE, 2007. pages 24

[38] Clearspeed inc. http://www.clearspeed.com. pages 25

[39] Dac Pham, Shigehiro Asano, Mark Bolliger, Michael N Day, H Peter Hofstee, C Johns, J Kahle, Atsushi Kameyama, John Keaty, Yoshio Masubuchi, et al. The design and implementation of a first-generation cell processor. In *Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 IEEE International*, pages 184–592. IEEE, 2005. pages 25

[40] Ken Koch. Roadrunner platform overview. *Los Alamos National Laboratory*, 2008. pages 25

[41] Ricardo Marroquim and André Maximo. Introduction to gpu programming with glsl. In *Computer Graphics and Image Processing (SIBGRAPI TUTORIALS), 2009 Tutorials of the XXII Brazilian Symposium on*, pages 3–16. IEEE, 2009. pages 25

[42] John D Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E Lefohn, and Timothy J Purcell. A survey of general-purpose computation on graphics hardware. In *Computer graphics forum*, volume 26, pages 80–113. Wiley Online Library, 2007. pages 26

[43] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5) :879–899, 2008. pages 26

[44] Nvidia corp. fermi compute architecture white paper. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf. pages 26

[45] Craig M Wittenbrink, Emmett Kilgariff, and Arjun Prabhu. Fermi gf100 gpu architecture. *Micro, IEEE*, 31(2) :50–59, 2011. pages 26

[46] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, et al. Larrabee : a many-core x86 architecture for visual computing. In *ACM Transactions on Graphics (TOG)*, volume 27, page 18. ACM, 2008. pages 26

[47] Timothy G. Mattson, Rob Van der Wijngaart, and Michael Frumkin. Programming the intel 80-core network-on-a-chip terascale processor. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 38 :1–38 :11, Piscataway, NJ, USA, 2008. IEEE Press. pages 26

[48] J.S. Vetter. *Contemporary High Performance Computing : From Petascale Toward Exascale*. Chapman and Hall/CRC Computational Science Series. Taylor & Francis Group, 2013. pages 27

[49] Rick Merritt. Cpu designers debate multi-core future. *EETimes Online, February*, 2008. pages 29

[50] Chuck moore's biography, colorforth. http://www.colorforth.com/bio.html. pages 29

[51] Tilera : About us. tilera corporation. http://www.tilera.com/about_tilera. pages 29

[52] Sony computer entertainment inc. introduces playstation 4 (ps4). http://www.scei.co.jp/corporate/release/pdf/130221a_e.pdf, february 2013. pages 29

[53] Bill Dally. Project denver processor to usher in new era of computing. http://blogs.nvidia.com/blog/2011/01/05/project-denver-processor-to-usher-in-new-era-of-computing/, 2011. pages 29

[54] Rajeeb Hazra. Driving industrial innovation on the path to exascale : From vision to reality, isc'13, June 2013. pages 30

[55] Kalray's mppa (multi-purpose processor array) - programmable manycore processor. http://www.kalray.eu/products/mppa-manycore/mppa-256/. pages 30

[56] Mateusz Berezecki, Eitan Frachtenberg, Mike Paleczny, and Kenneth Steele. Power and performance evaluation of memcached on the tilepro64 architecture. *Sustainable Computing : Informatics and Systems*, 2(2) :81–90, 2012. pages 30

[57] The mit angstrom project : Universal technologies for exascale computing. http://projects.csail.mit.edu/angstrom/. pages 30

[58] Eric Smalley. "mit genius stuffs 100 processors into single chip". wired magazine, January 2007. pages 30

[59] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition : A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006. pages 33

[60] Gharachorloo Adve. Shared memory consistency models : a tutorial. pages 34

[61] McKee Wulf. Hitting the memory wall : Implications of the obvious. 1994. pages 34

[62] Lei Chai, Qi Gao, and Dhabaleswar K. Panda. Understanding the impact of multi-core architecture in cluster computing : A case study with intel dual-core system. In *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, CCGRID '07, pages 471–478, Washington, DC, USA, 2007. IEEE Computer Society. pages 34

[63] Ulrich Drepper. What every programmer should know about memory, 2007. pages 35

[64] F. Dahlgren and J. Torrellas. *Cache-Only Memory Architectures*. Computer, vol. 32. 1999. pages 36

[65] R. Yang, J. Antony, P. P. Janes, and A. P. Rendell. Memory and thread placement effects as a function of cache usage : A study of the gaussian chemistry code on the sunfire x4600 m2. In *Proceedings of the The International Symposium on Parallel Architectures, Algorithms, and Networks*, ISPAN '08, pages 31–36, Washington, DC, USA, 2008. IEEE Computer Society. pages 37

[66] I. Kadayif and M. Kandemir. Data space-oriented tiling for enhancing locality. *ACM Trans. Embed. Comput. Syst.*, 4(2) :388–414, May 2005. pages 37

[67] Fengguang Song, S. Moore, and J. Dongarra. L2 cache modeling for scientific applications on chip multiprocessors. In *Parallel Processing, 2007. ICPP 2007. International Conference on*, pages 51–51, 2007. pages 37

[68] Fengguang Song, Shirley Moore, and Jack Dongarra. Modeling of l2 cache behavior for thread-parallel scientific programs on chip multi-processors. Technical report, In UT Computer Science, 2006. pages 37

[69] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, pages 285–, Washington, DC, USA, 1999. IEEE Computer Society. pages 38

[70] Intel Math Kernel Library (Intel MKL). http://software.intel.com/en-us/intel-mkl. pages 38

[71] AMD Core Math Library (ACML). http://developer.amd.com/tools-and-sdks/cpu-development/amd-core-math-library-acml/. pages 38

[72] Marc Tchiboukdjian, Vincent Danjean, Thierry Gautier, Fabien Mentec, and Bruno Raffin. A work stealing scheduler for parallel loops on shared cache multicores. In *Euro-Par 2010 Parallel Processing Workshops*, volume 6586 of *Lecture Notes in Computer Science*, pages 99–107. Springer Berlin Heidelberg, 2011. pages 40

[73] W. Stallings. *Computer Organization And Architecture : Designing For Performance*. The William Stallings books on computer and data communications technology. Pearson Prentice Hallï, 2006. pages 41

[74] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M.S. Lam. The stanford dash multiprocessor. *Computer*, 25(3) :63–79, 1992. pages 42

[75] R. E. Kessler and J.L. Schwarzmeier. Cray t3d : a new dimension for cray research. In *Compcon Spring '93, Digest of Papers.*, pages 176–182, 1993. pages 42

[76] James Laudon and Daniel Lenoski. The sgi origin : a ccnuma highly scalable server. *SIGARCH Comput. Archit. News*, 25(2) :241–251, May 1997. pages 42

[77] Lenoski. *Scalable Shared-Memory MultiProcessing*. 1997. pages 42

[78] HyperTransport I/O Link Specification. http://www.hypertransport.org. pages 43

[79] HyperTransport Technology I/O Link, A High Bandwidth I/O Architecture. Technical report, AMD INC., 2001. pages 43

[80] INTEL CORP. Intel quickpath architecture : A new system architecture for unleashing the performance of future generations of intel multi-core microprocessors. Technical report, 2008. pages 43

[81] Antony Joseph, Janes Pete P., and Rendell Alistair P. Exploring thread and memory placement on numa architectures : Solaris and linux, ultrasparc/f ireplane and opteron/hypertransport. 2006. pages 43

[82] Timothy Brecht. On the importance of parallel application placement in numa multiprocessors. In *USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems - Volume 4*, Sedms'93, pages 1–1, Berkeley, CA, USA, 1993. USENIX Association. pages 43

[83] Joseph Antony, Pete P. Janes, and Alistair P. Rendell. Exploring thread and memory placement on numa architectures : solaris and linux, ultrasparc/fireplane and opteron/hypertransport. In *Proceedings of the 13th international conference on High Performance Computing*, HiPC'06, pages 338–352, Berlin, Heidelberg, 2006. Springer-Verlag. pages 43

[84] David A. Patterson and John L. Hennessy. *Computer Organization and Design (4th Edition)*. The Morgan Kaufmann Series in Computer Architecture and Design. Academic Press, 2009. pages 43

[85] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the dash multiprocessor. *SIGARCH Comput. Archit. News*, 18(3a) :148–159, May 1990. pages 43

[86] C.N. Keltcher, K.J. McGrath, A. Ahmed, and P. Conway. The amd opteron processor for multiprocessor servers. *Micro, IEEE*, 23(2) :66–76, 2003. pages 43

[87] Pat Conway, Nathan Kalyanasundharam, Gregg Donley, Kevin Lepak, and Bill Hughes. Cache hierarchy and memory subsystem of the amd opteron processor. *IEEE Micro*, 30(2) :16–29, March 2010. pages 43

[88] INTEL CORP. First the tick, now the tock : Next generation intel microarchitecture (nehalem). Technical report, 2008. pages 44

[89] KEVIN J. BARKER, KEI DAVIS, ADOLFY HOISIE, DARREN J. KERBYSON, MIKE LANG, SCOTT PAKIN, and JOSE CARLOS SANCHO. A performance evaluation of the nehalem quad-core processor for scientific computing. *Parallel Processing Letters*, 18(04) :453–469, 2008. pages 44

[90] AMD Accelerated Processing Units. http://www.amd.com/us/products/technologies/apu/Pages/apu.aspx. pages 46

[91] Stéphanie Moreaud, Brice Goglin, and Raymond Namyst. Adaptive MPI multirail tuning for non-uniform input/output access. EuroMPI'10. pages 48

[92] Allison Proffitt. Exascale challenges, opportunities. http://www.bio-itworld.com, June 2013. pages 53

[93] Avinash Sodani and Chief Architect MIC Processor. Race to exascale : Opportunities and challenges. In *Keynote at the Annual IEEE/ACM 44th Annual International Symposium on Microarchitecture*, 2011. pages 53

[94] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Hideki Saito, Rakesh Krishnaiyer, Mikhail Smelyanskiy, Milind Girkar, and Pradeep Dubey. Can traditional programming bridge the ninja performance gap for parallel computing applications ? *SIGARCH Comput. Archit. News*, 40(3) :440–451, June 2012. pages 54

[95] H. Sutter. The free lunch is over : A fundamental turn toward concurrency in software, Dr. Dobb's Journal. http://www.top500.org, 2005. pages 57, 60

[96] Ambrose Bierce. *The devil's dictionary*. Courier Dover Publications, 1993. pages 58

[97] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM. pages 58

[98] John L. Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31 :532–533, 1988. pages 59

[99] Dennis M. Ritchie. The evolution of the unix time-sharing system. *Communications of the ACM*, 1984. pages 60

[100] Patrick Deiber. Les processus légers (threads), 1999. pages 60

[101] Ieee standard for information technology - portable operating system interface (posix). system interfaces. *IEEE Std 1003.1, 2004 Edition. The Open Group Technical Standard. Base Specifications, Issue 6. Includes IEEE Std 1003.1-2001, IEEE Std 1003.1-2001/Cor 1-2002 and IEEE Std 1003.1-2001/Cor 2-2004. Syst*, 2004. pages 60

[102] Xavier Leroy. The linux threads library, 1999. pages 60

[103] Ulrich Drepper and Ingo Molnar. The Native POSIX Thread Library for Linux, 2002. pages 60

[104] David R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997. pages 60

[105] RALF S. ENGELSCHALL. Gnu portable threads (pth), 1999. pages 60

[106] Daniel J. Berg. Java threads - a white paper. Technical report, Sun Micro-systems, 1996. pages 60

[107] Raymond Namyst and Jean-François Méhaut. *Marcel : Une bibliothèque de processus légers*. LIFL, Univ. Sciences et Techn. Lille, 1995. pages 60

[108] Raymond Namyst. *PM2 : un environnement pour une conception portable et une exécution efficace des applications parallèles irrégulières*. PhD thesis, Univ. de Lille 1, January 1997. pages 60

[109] V. S. Sunderam. Pvm : a framework for parallel distributed computing. *Concurrency : Pract. Exper.*, 2(4) :315–339, November 1990. pages 61

[110] The message passing interface (mpi) standard. http://www.mcs.anl.gov/research/projects/mpi/. pages 61

[111] Mpi-2 : Extensions to the message-passing interface. http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html. pages 61

[112] Mpi 3.0 standardization effort. http://meetings.mpi-forum.org/MPI_3.0_main_page.php. pages 61

[113] Mpich2 : High-performance and widely portable mpi. http://www.mcs.anl.gov/research/projects/mpich2/. pages 61

[114] William Gropp. Mpich2 : A new start for mpi implementations. In *Proceedings of the 9th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 7–, London, UK, UK, 2002. Springer-Verlag. pages 61

[115] Openmpi : Open source high performance computing. http ://www.open-mpi.org/. pages 61

[116] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI : Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004. pages 61

[117] Bullx cluster suite - application developer's guide. 2.1-mpibull2. pages 61

[118] T. Shanley, J. Winkles, and Inc MindShare. *InfiniBand Network Architecture*. PC System Architecture Series. ADDISON WESLEY Publishing Company Incorporated, 2003. pages 61

[119] Intel mpi library. http://software.intel.com/en-us/intel-mpi-library. pages 61

[120] Using the intel mpi library on intel xeon phi coprocessor systems. pages 61

[121] An overview of programming for intel xeon processors and intel xeon phi coprocessors. pages 61

[122] Pgas : Partitioned global address space. http://www.pgas.org. pages 62

[123] Robert W Numrich and John Reid. Co-array fortran for parallel programming. In *ACM Sigplan Fortran Forum*, volume 17, pages 1–31. ACM, 1998. pages 62

[124] William W Carlson, Jesse M Draper, David E Culler, Kathy Yelick, Eugene Brooks, and Karen Warren. *Introduction to UPC and language specification*. Center for Computing Sciences, Institute for Defense Analyses, 1999. pages 62

[125] Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. *UPC : distributed shared memory programming*, volume 40. Wiley-Interscience, 2005. pages 62

[126] Berkeley unified parallel c project. http://upc.lbl.gov. pages 62

[127] Gnu unified parallel c. http://www.gccupc.org. pages 62

[128] Vijay Saraswat, George Almasi, Ganesh Bikshandi, Calin Cascaval, David Cunningham, David Grove, Sreedhar Kodali, Igor Peshansky, and Olivier Tardieu. The asynchronous partitioned global address space model. In *Proceedings of The First Workshop on Advances in Message Passing*, 2010. pages 62

[129] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10 : an object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10) :519–538, October 2005. pages 62

[130] Bradford L Chamberlain, David Callahan, and Hans P Zima. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, 21(3) :291–312, 2007. pages 62

[131] William J. Dally, Ujval J. Kapasi, Brucek Khailany, Jung Ho Ahn, and Abhishek Das. Stream processors : Progammability and efficiency. *Queue*, 2(1) :52–62, March 2004. pages 62

[132] NVIDIA Corporation. *NVIDIA CUDA, Compute Unified Device Architecture Programming Guide, v5.0*, 2012. pages 62

[133] Shane Ryoo, Christopher I Rodrigues, Sara S Baghsorkhi, Sam S Stone, David B Kirk, and Wen-mei W Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82. ACM, 2008. pages 62

[134] PGI CUDA C/C++ for Multi-core x86. http://www.pgroup.com/resources/cuda-x86.htm. pages 63

[135] John A. Stratton, Sam S. Stone, and Wen-Mei W. Hwu. Languages and compilers for parallel computing. chapter MCUDA : An Efficient Implementation of CUDA Kernels for Multi-core CPUs, pages 16–30. Springer-Verlag, Berlin, Heidelberg, 2008. pages 63

[136] Gregory Diamos, Andrew Kerr, and Mukil Kesavan. Translating gpu binaries to tiered simd architectures with ocelot. Technical Report 0901, January 2009. pages 63

[137] Chris Lattner and Vikram Adve. Llvm : A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004. pages 63

[138] OpenCL - the open standard for parallel programming of heterogeneous systems. http://khronos.org/opencl. pages 63

[139] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. From cuda to opencl : Towards a performance-portable solution for multi-platform gpu programming. *Parallel Computing*, 38(8) :391–407, 2012. pages 63

[140] Justin Hensley. Close to the metal - gpgpu,siggraph, 2007. pages 63

[141] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus : stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3) :777–786, August 2004. pages 63

[142] AMD. Amd brook+ presentation, 2007. pages 63

[143] Tianyi David Han and Tarek S Abdelrahman. hicuda : High-level gpgpu programming. *Parallel and Distributed Systems, IEEE Transactions on*, 22(1) :78–90, 2011. pages 63

[144] Michael Wolfe. Implementing the pgi accelerator model. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 43–50. ACM, 2010. pages 64

[145] Openhmpp directives specification. http://www.caps-entreprise.com/wp-content/uploads/2012/11/OpenHMPP-Directives.pdf. pages 64

[146] R. Dolbeau, S. Bihan, and F. Bodin. HMPP : A hybrid multi-core parallel programming environment. In *Proc. of the Workshop on GPGPU'07*, 2007. pages 64

[147] Openacc directives for accelerators. http://www.openacc.org. pages 64

[148] Ruymán Reyes, Iván López-Rodríguez, Juan J Fumero, and Francisco de Sande. accull : an openacc implementation with cuda and opencl support. In *Euro-Par 2012 Parallel Processing*, pages 871–882. Springer, 2012. pages 64

[149] Robit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001. pages 65

[150] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5) :212–223, May 1998. pages 65

[151] Intel Corp. A Quick, Easy and Reliable Way to Improve Threaded Performance : Intel Cilk Plus, 2010. pages 65

[152] INTEL. Thread Building Blocks. http://www.intel.com/software/products/tbb. pages 66

[153] The openmp api specification for parallel programming - version 3.0. http://www.openmp.org, 2008. pages 66

[154] Marc Pérache, Hervé Jourdren, and Raymond Namyst. Mpc : A unified parallel runtime for clusters of numa machines. In *Proceedings of the 14th international Euro-Par conference on Parallel Processing*, Euro-Par '08, pages 78–88, Berlin, Heidelberg, 2008. Springer-Verlag. pages 66

[155] Marc Pérache, Patrick Carribault, and Hervé Jourdren. Mpc-mpi : An mpi implementation reducing the overall memory consumption. In Matti Ropo, Jan Westerholm, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Lecture Notes in Computer Science, pages 94–103. Springer Berlin Heidelberg, 2009. pages 66

[156] Patrick Carribault, Marc Pérache, and Hervé Jourdren. Enabling low-overhead hybrid mpi/openmp parallelism with mpc. In *Beyond Loop Level Parallelism in OpenMP : Accelerators, Tasking and More*, pages 1–14. Springer, 2010. pages 66

[157] François Galilée, Gerson GH Cavalheiro, J-L Roch, and Mathias Doreille. Athapascan-1 : On-line building data flow graph in a parallel language. In *Parallel Architectures and Compilation Techniques, 1998. Proceedings. 1998 International Conference on*, pages 88–95. IEEE, 1998. pages 66

[158] Thierry Gautier, Xavier Besseron, and Laurent Pigeon. Kaapi : A thread scheduling runtime system for data flow computations on cluster of multi-processors. In *Proceedings of the 2007 international workshop on Parallel symbolic computation*, pages 15–23. ACM, 2007. pages 66

[159] Xavier Besseron, Christophe Laferrière, Daouda Traore, and Thierry Gautier. X-kaapi : Une nouvelle implémentation extrême du vol de travail pour des algorithmes í¿ grain fin. *19èmes Rencontres Francophones Du Parallélisme (RenPar 2009)*. pages 66

[160] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, pages 355–364. ACM, 2010. pages 66

[161] Everton Hermann, Bruno Raffin, François Faure, Thierry Gautier, and Jérémie Allard. Multi-gpu and multi-cpu parallelization for interactive physics simulations. In *Euro-Par 2010-Parallel Processing*, pages 235–246. Springer, 2010. pages 67

[162] Umut A Acar, Guy E Blelloch, and Robert D Blumofe. The data locality of work stealing. *Theory of Computing Systems*, 35(3) :321–347, 2002. pages 67

[163] Thierry Gautier, Joao Vicente Ferreira Lima, Nicolas Maillard, Bruno Raffin, et al. Xkaapi : A runtime system for data-flow task programming on heterogeneous architectures. In *27th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2013. pages 67

[164] Alejandro Duran, Eduard Ayguade, Rosa M Badia, Jesus Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. Ompss : a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02) :173–193, 2011. pages 67

[165] Vladimir Subotić, Steffen Brinkmann, Vladimir Marjanović, Rosa M. Badia, Jose Gracia, Christoph Niethammer, Eduard Ayguade, Jesus Labarta, and Mateo Valero. Programmability and portability for exascale : Top down programming methodology and tools with starss. *Journal of Computational Science*, (0) :–, 2013. pages 67

[166] J. Balart, A. Duran, M. Gonzàlez, X. Martorell, E. Ayguadé, and J. Labarta. Nanos mercurium : a research compiler for openmp. In *European Workshop on OpenMP (EWOMP'04). Pp*, pages 103–109, 2004. pages 67

[167] Daouda Traore. Nanos++ User Manual. https://pm.bsc.es/projects/nanox. pages 67

[168] Josep M Perez, Rosa M Badia, and Jesus Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Cluster Computing, 2008 IEEE International Conference on*, pages 142–151. IEEE, 2008. pages 67

[169] Josep M Perez, Pieter Bellens, Rosa M Badia, and Jesus Labarta. Cellss : Making it easier to program the cell broadband engine processor. *IBM Journal of Research and Development*, 51(5) :593–604, 2007. pages 67

[170] Judit Planas, Rosa M Badia, Eduard Ayguadé, and Jesus Labarta. Hierarchical task-based programming with starss. *International Journal of High Performance Computing Applications*, 23(3) :284–299, 2009. pages 67

[171] Enric Tejedor, Montse Farreras, David Grove, Rosa M Badia, Gheorghe Almasi, and Jesus Labarta. Clusterss : a task-based programming model for clusters. In *Proceedings of the 20th international symposium on High performance distributed computing*, pages 267–268. ACM, 2011. pages 67

[172] Roger Ferrer, Judit Planas, Pieter Bellens, Alejandro Duran, Marc Gonzalez, Xavier Martorell, Rosa M Badia, Eduard Ayguade, and Jesus Labarta. Optimizing the exploitation of multicore processors and gpus with openmp and opencl. In *Languages and Compilers for Parallel Computing*, pages 215–229. Springer, 2011. pages 67

[173] Eduard Ayguadé, Rosa M Badia, Francisco D Igual, Jesús Labarta, Rafael Mayo, and Enrique S Quintana-Ortí. An extension of the starss programming model for platforms with multiple gpus. In *Euro-Par 2009 Parallel Processing*, pages 851–862. Springer, 2009. pages 67

[174] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU : A unified platform for task scheduling on heterogeneous multicore architectures. In *Euro-Par'09*. pages 68

[175] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu : a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation : Practice and Experience*, 23(2) :187–198, 2011. pages 68

[176] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Mathieu Faverge, Julien Langou, Hatem Ltaief, and Stanimire Tomov. LU factorization for accelerator-based systems. In *9th ACS/IEEE International Conference on Computer Systems and Applications (AICCSA 11)*, Sharm El-Sheikh, Egypt, 2011. pages 68

[177] Haluk Topcuoglu, Salim Hariri, and Min-you Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *Parallel and Distributed Systems, IEEE Transactions on*, 13(3) :260–274, 2002. pages 68

[178] Cédric Augonnet, Samuel Thibault, and Raymond Namyst. Automatic Calibration of Performance Models on Heterogeneous Multicore Architectures. In *3rd Workshop on Highly Parallel Processing on a Chip (HPPC 2009)*, Delft, Pays-Bas, August 2009. pages 68

[179] Cristina Boeres, George Chochia, and Peter Thanisch. *On the scope of applicability of the ETF algorithm*. Springer, 1995. pages 68

[180] Laxmikant V. Kale and Sanjeev Krishnan. Charm++ : a portable concurrent object oriented system based on c++. *SIGPLAN Not.*, 28(10) :91–108, October 1993. pages 68

[181] L. V. Kalé, B. Ramkumar, A. B. Sinha, and V. A. Saletore. The CHARM Parallel Programming Language and System : Part II – The Runtime system. *Parallel Programming Laboratory Technical Report #95-03*, 1994. pages 68

[182] David M. Kunzman, Gengbin Zheng, Eric Bohm, James C. Phillips, and Laxmikant V. Kale. Charm++ simplifies coding for the cell processor. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM. pages 68

[183] David M Kunzman and Laxmikant V Kalé. Programming heterogeneous clusters with accelerators using object-based programming. *Scientific Programming*, 19(1) :47–62, 2011. pages 68

[184] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin : exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 45–55, New York, NY, USA, 2009. ACM. pages 68

[185] Tze Meng Low and Robert A. Van De Geijn. An api for manipulating matrices stored by blocks. Technical report, 2004. pages 69

[186] Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, Robert A. Van De Geijn, Field G. Van Zee, and Ernie Chan. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Trans. Math. Softw.*, 36(3) :14 :1–14 :26, July 2009. pages 69

[187] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Comput.*, 35(1) :38–53, January 2009. pages 69

[188] Fred G. Gustavson, Isak Jonsson, Bo Kågström, and Per Ling. Towards peak performance on hierarchical smp memory architectures - new recursive blocked data formats and blas. In *PPSC*, 1999. pages 69

[189] Vinod Valsalam and Anthony Skjellum. A framework for high-performance matrix multiplication based on hierarchical abstractions, algorithms and optimized low-level kernels. *Concurrency and Computation : Practice and Experience*, 14(10) :805–839, 2002. pages 69

[190] Marc Baboulin, Jack Dongarra, Julien Herrmann, and Stanimire Tomov. Accelerating linear system solutions using randomization techniques. Technical report, INRIA, 2011. pages 69

[191] Enrique S. Quintana, Gregorio Quintana, Xiaobai Sun, and Robert vande Geijn. A note on parallel matrix inversion. *SIAM J. Sci. Comput.*, 22(5) :1762–1771, May 2000. pages 70

[192] Enrique S. Quintana-Ortí. The science of programming matrix computations, 2008. pages 70

[193] Kyungjoo Kim, Victor Eijkhout, and Robert A. van de Geijn. Dense matrix computation on a heterogenous architecture : A block synchronous approach. Technical Report TR-12-04, Texas Advanced Computing Center, The University of Texas at Austin, 2012. submitted to Concurrency and Computation : Practice and Experience. pages 70, 71

[194] J. J. Dongarra, Jeremy Du Croz, Sven Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16(1) :1–17, March 1990. pages 70

[195] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of fortran basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 14(1) :1–17, March 1988. pages 70

[196] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3) :308–323, September 1979. pages 70

[197] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen. Lapack : a portable linear algebra library for high-performance computers. In *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, Supercomputing '90, pages 2–11, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press. pages 71

[198] Scalapack : A scalable linear algebra package. http://www.netlib.org/scalapack. pages 71

[199] Jaeyoung Choi, Jack J Dongarra, Roldan Pozo, and David W Walker. Scalapack : A scalable linear algebra library for distributed memory concurrent computers. In *Frontiers of Massively Parallel Computation, 1992., Fourth Symposium on the*, pages 120–127. IEEE, 1992. pages 71

[200] Field Van Zee. libflame : The complete reference, 2011. pages 71

[201] Field G. Van Zee, Ernie Chan, Robert A. van de Geijn, Enrique S. Quintana-Orti, and Gregorio Quintana-Orti. The libflame library for dense matrix computations. *Computing in Science and Engineering*, 11(6) :56–63, 2009. pages 71

[202] Ernie Chan, Robert van de Geijn, and Andrew Chapman. Managing the complexity of lookahead for lu factorization with pivoting. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, SPAA '10, pages 200–208, New York, NY, USA, 2010. ACM. pages 71

[203] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Comput.*, 35(1) :38–53, January 2009. pages 71

[204] Dongarra Jack YarKhan Asim, Kurzak Jakub. Quark users' guide - innovative computing laboratory, 2011. pages 71

[205] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Azzam Haidar, Thomas Herault, Jakub Kurzak, Julien Langou, Pierre Lemarinier, Hatem Ltaief, Piotr Luszczek, Asim Yarkhan, and Jack Dongarra. Distibuted dense numerical linear algebra algorithms on massively parallel architectures : Dplasma. pages 71

[206] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. Towards dense linear algebra for hybrid gpu accelerated manycore systems. *Parallel Comput.*, 36 :232–240, 2010. pages 71

[207] Fengguang Song, Stanimire Tomov, and Jack Dongarra. Enabling and scaling matrix computations on heterogeneous multi-core and multi-gpu systems. In *Proceedings of the 26th ACM international conference on Supercomputing*, ICS '12, pages 365–376, New York, NY, USA, 2012. ACM. pages 71

[208] Michael Deisher, Mikhail Smelyanskiy, Brian Nickerson, Victor W. Lee, Michael Chuvelev, and Pradeep Dubey. Designing and dynamically load balancing hybrid LU for multi/many-core. In *ISC*, 2011. pages 72

[209] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. Towards dense linear algebra for hybrid gpu accelerated manycore systems. *Parallel Comput.*, pages 232–240, June 2010. pages 72, 83

[210] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures : The plasma and magma projects. In *Journal of Physics : Conference Series*, volume Vol. 180, 2009. pages 72, 83

[211] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, and S. Tomov. A hybridization methodology for high-performance linear algebra software for GPUs. In *GPU Computing Gems*. Morgan Kaufmann, 2010. pages 72, 83

[212] Chongxiao Cao, Jack Dongarra, Peng Du, Mark Gates, Piotr Luszczek, and Stanimire Tomov. clmagma : High performance dense linear algebra with opencl. pages 72

[213] Patrick R. Amestoy, Iain S. Duff, Jean-Yves L'Excellent, and Jacko Koster. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM J. Matrix Anal. Appl.*, 23 :15–41, 2001. pages 72

[214] Pascal Henon, Pierre Ramet, and Jean Roman. PaStiX : A parallel sparse direct solver based on a static scheduling for mixed 1D/2D block distributions. In *Proc. of Irregular'00*, pages 519–525. Springer-Verlag, 2000. pages 72

[215] Xiaoye S. Li. An overview of SuperLU : Algorithms, implementation, and user interface. *ACM Trans. Math. Softw.*, 31 :302–325, 2005. pages 72

[216] John R. Humphrey, Daniel K. Price, Kyle E. Spagnoli, Aaron L. Paolini, and Eric J. Kelmelis. Cula : hybrid gpu accelerated linear algebra routines. pages 770502–770502–7, 2010. pages 72

[217] Junjie Lai and André Seznec. Performance Upper Bound Analysis and Optimization of SGEMM on Fermi and Kepler GPUs. In *CGO '13 - 2013 International Symposium on Code Generation and Optimization*, Shenzhen, Chine, February 2013. pages 82

[218] NVIDIA CUDA Basic Linear Algebra Subroutines (NVIDIA cuBLAS). https://developer.nvidia.com/cublas. pages 82

[219] Matrix Algebra on GPU and Multicore Architectures (MAGMA). http://icl.cs.utk.edu/magma. pages 83

[220] Matrices Over Runtime Systems @ Exascale (MORSE). http://icl.cs.utk.edu/projectsdev/morse. pages 83

[221] A. YarKhan, J. Kurzak, and J. Dongarra. Quark users' guide : Queueing and runtime for kernels. Technical report, Innovative Computing Laboratory, University of Tennessee, 2011. pages 84

[222] Automatically Tuned Linear Algebra Software (ATLAS). http://math-atlas.sourceforge.net. pages 85

[223] Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the atlas project. *PARALLEL COMPUTING*, 27 :2001, 2000. pages 85

[224] The PHiPAC (Portable High Performance ANSI C). http://www1.icsi.berkeley.edu/~bilmes/phipac. pages 85

[225] J. Bilmes, K. Asanović, J. Demmel, D. Lam, and C.W. Chin. PHiPAC : A portable, high-performance, ANSI C coding methodology and its application to matrix multiply. LAPACK working note 111, University of Tennessee, 1996. pages 85

[226] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2) :216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation". pages 85

[227] Akira Nukada and Satoshi Matsuoka. Auto-tuning 3-d fft library for cuda gpus. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 30 :1–30 :10, New York, NY, USA, 2009. ACM. pages 85

[228] SPARSITY. http://www.cs.berkeley.edu/~yelick/sparsity. pages 85

[229] Eun jin Im and Katherine Yelick. Optimizing sparse matrix vector multiplication on smps. In *In Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 1999. pages 85

[230] Portable Hardware Locality (hwloc). http://www.open-mpi.org/projects/hwloc. pages 89

[231] Francois Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc : A generic framework for managing hardware affinities in hpc applications. In *Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, PDP '10, pages 180–186, Washington, DC, USA, 2010. IEEE Computer Society. pages 89

[232] Joseph Antony, Pete P. Janes, and Alistair P. Rendell. Exploring thread and memory placement on numa architectures : solaris and linux, ultrasparc/fireplane and opteron/hypertransport. In *Proceedings of the 13th international conference on High Performance Computing*, HiPC'06, pages 338–352, Berlin, Heidelberg, 2006. Springer-Verlag. pages 90

[233] K LEEN (A.). A numa api for linux. Technical report, Novell, Technical Linux Whitepaper, 2005. pages 91

[234] Sebastien Valat. *Contribution à l'amélioration des méthodes d'optimisation de la gestion de la mémoire dans le cadre du calcul haute performance*. PhD thesis, UVSQ, 2013. pages 91

[235] Minas Project - Memory affInity maNAgement System. https://sites.google.com/site/pousachristiane/minas. pages 91

[236] Laércio L Pilla, Christiane Pousa Ribeiro, Daniel Cordeiro, and Jean-François Méhaut. Charm++ on numa platforms : the impact of smp optimizations and a numa-aware load balancer. In *4th workshop of the INRIA-Illinois Joint Laboratory on Petascale Computing. Urbana, IL, USA*, 2010. pages 91

[237] Timothy G. Mattson, Michael Riepen, Thomas Lehnig, Paul Brett, Werner Haas, Patrick Kennedy, Jason Howard, Sriram Vangal, Nitin Borkar, Greg Ruhl, and Saurabh Dighe. The 48-core scc processor : the programmer's view. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society. pages 97

[238] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8) :103–111, August 1990. pages 113

[239] R.H. Bisseling. *Parallel Scientific Computation : A Structured Approach Using BSP and MPI*. Oxford scholarship online. OUP Oxford, 2004. pages 113

[240] OpenMP 4.0 complete specifications. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf, July 2013. pages 119

[241] Amos Gilat. *MATLAB : An introduction with Applications*. Wiley Hoboken, NJ, 2005. pages 119

[242] Rice University. Dept. of Computer Science, D. Callahan, and K. Kennedy. *Compiling Programs for Distributed-memory Multiprocessors*. Technical report. Rice University, 1988. pages 119

[243] A. Rogers and K. Pingali. Process decomposition through locality of reference. PLDI '89. pages 119

[244] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S Müller, and Wolfgang E Nagel. The vampir performance analysis tool-set. In *Tools for High Performance Computing*, pages 139–155. Springer, 2008. pages 129

[245] Markus Geimer, Felix Wolf, Brian JN Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The scalasca performance toolset architecture. *Concurrency and Computation : Practice and Experience*, 22(6) :702–719, 2010. pages 129

[246] Sameer S Shende and Allen D Malony. The tau parallel performance system. *International Journal of High Performance Computing Applications*, 20(2) :287–311, 2006. pages 129

[247] Vincent Pillet, Jesús Labarta, Toni Cortes, and Sergi Girona. Paraver : A tool to visualize and analyze parallel

code. In *Proceedings of WoTUG-18 : Transputer and occam Developments*, volume 44, pages 17–31, 1995. pages 129

[248] Jean-Baptiste Besnard, Marc Pérache, and William Jalby. Event streaming for online performance measurements reduction. *Fourth International Workshop on Parallel Software Tools and Tool Infrastructures (PSTI 2013)*, 2013. pages 129

[249] NVIDIA Visual Profiler. https://developer.nvidia.com/nvidia-visual-profiler. pages 131

[250] Intel Vtune Amplifier. http://software.intel.com/en-us/intel-vtune-amplifier-xe. pages 131

[251] Lamia Djoudi, Denis Barthou, Patrick Carribault, William Jalby, Christophe Lemuet, Jean-Thomas Acquaviva, et al. Exploring application performance : a new tool for a static/dynamic approach. In *LACSI Symposium, Santa Fe*, pages 41–49, 2005. pages 131

[252] Jakub Kurzak and Jack Dongarra. Implementation of the mixed-precision high performance linpack benchmark on the cell processor. *Computer Science Tech. Report UT-CS-06-580, LAPACK Working Note*, 177, 2006. pages 134

[253] Massimiliano Fatica. Accelerating linpack with CUDA on heterogenous clusters. GPGPU-2, pages 46–51. ACM, 2009. pages 134

[254] HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers - Version 2.1. http://netlib.org/benchmark/hpl, 2012. pages 134

[255] E. Phillips M. Fatica. CUDA accelerated linpack on clusters. NVIDIA Corporation, 2010. pages 137

[256] Thomas A. Brunner and James Paul Holloway. Two-dimensional time dependent riemann solvers for neutron transport. *J. Comput. Phys.*, 210(1) :386–399, November 2005. pages 138

[257] Sc09 : Ibm lässt cell-prozessor auslaufen. http://www.heise.de/newsticker/meldung/SC09-IBM-laesst-Cell-Prozessor-auslaufen-864497.html. pages 151

[258] Amd firepro technology. http://www.amd.com/firepro. pages 151

[259] Intel Many Integrated Core Symmetric Communications Interface (SCIF), User Guide, octobre 2012. pages 151

[260] Isaac Gelado, John E. Stone, Javier Cabezas, Sanjay Patel, Nacho Navarro, and Wen-mei W. Hwu. An asymmetric distributed shared memory model for heterogeneous parallel systems. In *ASPLOS'10*. pages 152

[261] Plugins - GNU compiler collection (GCC) internals. http://gcc.gnu.org/onlinedocs/gccint/Plugins.html. pages 153

[262] Bianca Schroeder and Garth A Gibson. A large-scale study of failures in high-performance computing systems. *Dependable and Secure Computing, IEEE Transactions on*, 7(4) :337–350, 2010. pages 154

[263] HBP report - Human Brain Project. http://www.humanbrainproject.eu/, 2012. pages 154

[264] Nice ! the brain as a model for future supercomputers. https://share.sandia.gov/news/resources/news_releases/brain_supercomputers/, May 2013. pages 154

[265] Nages Sieslack. The races to understand the human brain. http://www.isgtw.org/feature/race-understand-human-brain, 2013. pages 154

[266] "no exascale for you !" an interview with berkeley lab's horst simon, hpcwire. http://www.hpcwire.com/hpcwire/2013-05-15/no_exascale_for_you_an_interview_with_nersc_s_horst_simon.html, 2013. pages 154

Jean-Yves VET

Parallélisme de tâches et localité de données dans un contexte multi-modèle de programmation pour supercalculateurs hiérarchiques et hétérogènes

Résumé

Les contributions de cette thèse s'appuient sur un modèle de programmation par tâches, dont l'originalité réside dans l'ajustement de la quantité de calcul en fonction de l'unité d'exécution ciblée. Ce modèle de programmation est particulièrement adapté à un équilibrage de charge dynamique entre des ressources de calcul hétérogènes. Il favorise une meilleure exploitation des unités de traitement en offrant une meilleure réactivité en présence de variations des temps d'exécution, lesquelles peuvent être générées par des codes de calcul irréguliers ou des mécanismes matériels difficilement prévisibles. De plus, la sémantique des tâches de calcul facilite le recours à des mécanismes de gestion automatisée des opérations de cohérence des mémoires déportées et décharge les développeurs de cette tâche fastidieuse et source d'erreurs. Nous avons développé la plateforme d'exécution H3LMS afin d'agréger les propositions de cette thèse. Cette plateforme est intégrée à l'environnement de programmation MPC afin de faciliter la cohabitation de plusieurs modèles de programmation pour une meilleure exploitation des grappes de calcul. H3LMS permet, entre autres, de mieux aiguiller les tâches vers les unités de traitement appropriées en réduisant la quantité de coûteux accès distants au sein d'un nœud de calcul. Ces travaux s'intéressent également à l'adaptation de codes de simulation existants, conçus à l'origine pour exploiter exclusivement des processeurs traditionnels et pouvant comporter plusieurs centaines de milliers de lignes de code. Les performances de la solution développée sont évaluées sur la bibliothèque Linpack et par une application numérique réaliste du CEA.

Mots clés : supercalculateur hétérogène, parallélisme, ordonnancement, tâches de calcul, localité de données, GPU, MIC, NUMA

Abstract

This thesis makes several distinct contributions which rely on a dedicated task-based programming model. The novelty of this model resides in a dynamic adjustment of the quantity of embedded operations depending on the targeted processing unit. It is particularly well adapted to dynamically balance workloads between heterogeneous processing units. It better harnesses those units by strengthening responsiveness in the presence of execution times fluctuations induced by irregular codes or unpredictable hardware mechanisms. Moreover, the semantics and programming interface of the task-parallel model facilitates the use of automated behaviors such as data coherency of deported memories. It alleviates the burden of developers by taking care of this tedious work and which can be a source of errors. We developed H3LMS an execution platform designed to combine the propositions detailed in the thesis. The platform is integrated to the MPC programming environment in order to enhance cohabitation with other programming models and thus better harness clusters. H3LMS is elaborated to improve task scheduling between homogeneous and heterogeneous processing units by reducing the need to resort to distant accesses in a cluster node. This thesis also focuses on the adaptation of legacy codes which are originally designed to exploit traditional processors and may also consist of hundreds of thousand lines of code. The performance of this solution is evaluated on the Linpack library and on a legacy numerical application from the CEA.

Keywords : Heterogeneous supercomputer, parallelism, task scheduling, data locality, GPU, MIC, NUMA