Un modèle de structure de données Cache-aware pour un parallélisme et un équilibrage dynamique de la charge

Marwa Sridi

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Mathématiques et Informatique**

Arrêté ministérial :

Présentée par

Marwa Sridi

Thèse dirigée par **Bruno Raffin et codirigée par Vincent Faucher**

préparée au sein **Laboratoire de Dynamique CEA Saclay**
et de **l'Ecole Doctorale Mathématiques, Sciences et Technologie de l'Information, Informatique de Grenoble**

Un modèle de structure de données Cache-aware pour un parallélisme et un équilibrage dynamique de la charge

Thèse soutenue publiquement le **28 Avril 2016**,
devant le jury composé de :

M. Gilles Grimaud
Professeur, Université Lille 1, Rapporteur
M. Sébastien Limet
Professeur, Université d'Orléans, Rapporteur
M. Pierre-Alain Boucard
Professeur, ENS Cachan, Président
M. Bruno Raffin
Directeur de recherche, Université de Grenoble, Directeur de thèse
M. Vincent Faucher
Ingénieur de recherche, CEA Cadarache, Co-directeur
M. Thierry Gautier
Chargé de recherche, INRIA ENS Lyon, Examinateur

Table des matières

Table des figures

Liste des tableaux

Liste des Algorithmes

Première partie

Avant propos

Introduction

Apparue initialement au 20$^{\text{ième}}$ siècle, la simulation numérique est devenue un outil incontournable pour la recherche scientifique dans de nombreux secteurs (biologie, métrologie, aéronautique, nucléaire, ...). Cet outil consiste à étudier plusieurs scénarios d'évolution d'un système de manière virtuelle. Il nous évite ainsi de courir le risque des expérimentations coûteuses et parfois irréalisables ou dangereuses en laboratoires (dans le domaine du nucléaire par exemple).
D'après le cabinet international de conseil stratégique et d'analyse CIMdata [1], la simulation numérique occupe un marché en forte progression avec un rythme de l'ordre de 7,5% atteint en 2014. Ce progrès va de pair avec les avancées connues ces dernières années dans le domaine du calcul haute performance (HPC).
Afin de résoudre des problèmes scientifiques de plus grande taille et pour réaliser les simulations le plus rapidement possible, les scientifiques ont eu recours à la programmation parallèle. Celle-ci consiste à utiliser plusieurs unités de calcul pour traiter de manière simultanée les informations.
Pour tirer partie du parallélisme sur les architectures actuelles qui deviennent de plus en plus complexes, il est inéluctable de développer des modèles de programmation adaptés à l'organisation hiérarchique de ces architectures. Le développement de ces modèles se base sur deux leviers incontournables :

- l'utilisation efficace des niveaux hiérarchiques de la mémoire.
- la répartition équilibrée de la charge sur les unités de calcul disponibles de manière efficace en mémoire.

Le premier levier consiste à adapter les codes de simulation aux caractéristiques architecturales de la plateforme de calcul. Cette adaptation repose sur la construction de modèles qui approximent l'architecture mémoire très complexe. Cette architecture intègre des niveaux intermédiaires entre l'unité de calcul et la mémoire centrale. Ces niveaux, communément appelés *mémoire cache*, sont plus rapides d'accès que la mémoire centrale. Ils comportent à leur tour différents niveaux hiérarchiques privés à une unité de calcul (*le cœur* d'un processeur) ou partagés entre plusieurs unités de calcul (entre les cœurs d'un processeur dans une architecture multi-cœurs par exemple).
Les modèles à concevoir doivent profiter de la mémoire cache afin de contourner les accès fréquents à la mémoire distante et éviter de laisser l'unité de calcul passer des cycles à attendre le chargement de la donnée. Ainsi, l'enjeu consiste à maximiser les accès à des données locales au cache du cœur qui les demande.
Dans une démarche d'optimisation de l'utilisation de la mémoire cache, il est primordial d'optimiser *la localité spatiale* en veillant à ce que les données soient contiguës en mémoire et placées selon leur ordre d'utilisation par le processeur. Il faut, également, faire en sorte que les données présentes dans le cache soient réutilisées dans un futur proche avant qu'elles ne soient évincées du cache. Il s'agit d'une optimisation de *la localité temporelle* des données dans le cache.

1. CIMdata (http ://www.cimdata.com) est un cabinet de conseil qui étudie le marché du PLM (Product Lifecycle Management) et fournit une analyse de l'état d'évolution des investissements pour identifier les leaders du marché de l'année précédente.

Concrètement, les données manipulées dans les simulations correspondent à des grandeurs physiques liées entre elles à travers des lois de la physique qui prédisent le comportement du système simulé. Les modifications apportées par le modèle d'approximation doivent respecter les spécificités des applications ainsi que leurs structures de données.
Dans ce contexte, plusieurs travaux d'optimisation ont porté sur le ré-ordonnancement des boucles des codes de simulation dans le but d'améliorer la localité [134]. Ces approches sont utilisées dans certaines configurations des compilateurs. En revanche, leur intégration "manuelle" dans les applications ayant des dépendances complexes entre les instructions s'avère très difficile. De par cette complexité, des stratégies basées sur la modification de l'ordre d'accès aux données plutôt que sur le réordonnancement des instructions du programme ont été développées. Dans certaines de ces méthodes, l'objectif consiste à définir des lois d'algèbre linéaire qui décrivent l'ordre d'accès aux données lors de l'exécution du programme [37, 80]. L'inconvénient majeur de ces approches est qu'elles nécessitent une connaissance a priori de l'ordre d'accès aux données ce qui ne correspond pas au caractère générique des applications.
D'autres approches reposent sur des optimisations en amont du simulateur, au niveau de la phase du maillage. Elles consistent à projeter l'espace physique du maillage généralement représenté en 2D ou en 3D dans un espace unidimensionnel représentant l'organisation des données du maillage en mémoire. La numérotation proposée par ces méthodes est basée sur des courbes spécifiques appelées *space-filling curves* [115]. Ces techniques permettent de découper l'espace de manière récursive optimisant la localité des données en mémoire à la lumière de leur localité en espace. Ces approches sont universelles et peuvent être portées sur n'importe quelle architecture sans se soucier des caractéristiques de la mémoire cache. Ce sont des méthodes *cache-oblivious*. À la différences de ces approches *cache-oblivious* [59], les méthodes dites *cache-aware* [21, 67] se basent sur des modèles de réorganisation de la structure de données guidés par une connaissance des spécificités de l'architecture cible et qui respectent les besoins de l'application en termes de données suivant le déroulement du calcul. Parmi ces méthodes, nous citons l'approche AOS (*Array Of Structure*) [66], l'approche SOA (*Structure Of Arrays*) [67] et l'approche hybride SOAOS [131]. Ces trois approches agissent sur la manière dont les données seront stockées en mémoire selon l'architecture de cette dernière et en fonction des besoins de l'application.Par exemple, dans le cadre de l'approche AOS, si on considère un ensemble de points matériels P_i, nous stockons dans un seul tableau les coordonnées de chaque point de la liste X_i, Y_i, Z_i de manière contiguë. Ce modèle d'organisation est particulièrement adapté aux codes de simulation où les calculs de l'évolution du système simulé s'effectuent élément par élément.

Le deuxième levier essentiel pour atteindre l'objectif annoncé consiste à mettre en place une stratégie de parallélisme qui prend en considération la localité des données en mémoire. La localité doit être conservée dans le cadre d'une collaboration entre les différentes unités de calcul afin d'exécuter l'application le plus rapidement possible. Cette collaboration soulève de nombreux défis à relever pour assurer une utilisation efficace de la mémoire en exécution parallèle et pour que la charge de travail soit équitablement répartie entre les différentes unités de calcul.

Classiquement, la charge de travail est distribuée entre les unités de calcul de manière statique. Autrement dit, au début de l'exécution parallèle, l'ordonnanceur découpe le programme sur les ressources de calcul disponibles et cette répartition de charge sera maintenue jusqu'à la fin de l'exécution.
Dans des problèmes de simulations dynamiques, le système physique initial subit des évolutions importantes au cours du temps. À cause de ces évolutions, la quantité de travail n'est pas maintenue constante. Par conséquent, l'occupation des unités de calcul (cœurs) peut varier tout au long du calcul. Si la charge n'est pas rééquilibrée de manière dynamique au cours du temps, des cœurs risquent de terminer leur travail avant d'autres. Ainsi, au lieu d'exploiter toutes les ressources de calcul disponibles, nous allons nous retrouver avec seulement quelques cœurs actifs (avec éventuellement une grande quantité de travail), tandis que les autres cœurs sont passés à l'état inactif.
Pour répondre à cette problématique, des modèles d'ordonnancement ont été développés. Ces modèles implémentent des stratégies d'équilibrage dynamique de charge qui permettent, effectivement, de prendre en considération le caractère dynamique de la simulation. Ces modèles mettent en œuvre un ordonnancement qui permet d'éviter l'inactivité des unités de calcul tant que d'autres unités de calcul n'ont toujours pas achevé leur travail. Par défaut, dès qu'une unité de calcul termine son travail, elle sélectionne de manière aléatoire une unité de calcul "victime" pour lui voler une partie du travail lui restant à faire. C'est le principe de l'ordonnancement par *vol de travail aléatoire* (*Random workstealing*) [27] utilisé dans de nombreuses bibliothèques parallèles telles que Cilk [111], Intel TBB [112] et XKAAPI [53].
Cependant, la technique du vol aléatoire ne prend pas en compte la hiérarchie de l'architecture, ce qui peut conduire à une perte de performance. En effet, lors d'un vol, des échanges de données entre la mémoire du voleur et celle de la victime prennent place. Ces échanges sont d'autant plus couteux que le voleur et sa victime sont éloignés et ne se partagent pas le même niveau de la mémoire.
Pour remédier à ce problème, des techniques d'ordonnancement par vol de travail hiérarchique [136] ont été proposées. Ces techniques conditionnent les vols selon l'organisation hiérarchique de la plateforme de calcul. Elles favorisent les vols locaux dans les mêmes niveaux hiérarchiques de l'architecture. La priorité des vols décroit en s'éloignant du domaine local du voleur dans le but de minimiser les vols distants qui engendrent des accès à des mémoires distantes. Des approches d'ordonnancement statique pourraient également être considérées. Nous citons à ce propos les approches d'ordonnancement de liste ETF (*Earliest Task First*) qui consistent à ordonnancer d'abord la tâche prête exécutable le plus tôt. Dans cette même fin, des approches hybrides basées sur la pré-distribution des tâches sur les threads et le vol de travail comme dans XKAAPI peuvent être exploitées.

Contributions

Les travaux effectués au cours de cette thèse sont aiguillés par les deux problématiques que nous avons dégagées plus haut : l'optimisation de l'utilisation de la mémoire cache dans le cas des codes de simulation de grande taille et le parallélisme

basé sur l'équilibrage dynamique de charge dans un contexte d'utilisation efficace de la mémoire. Sur ces deux axes nous nous intéressons aux machines multi-processeurs et multi-cœurs à mémoires partagées. Les approches développées dans le cadre de ces travaux sont implémentées dans le code EUROPLEXUS qui est un code industriel de simulation en dynamique rapide des fluides et des structures. Ce code est caractérisé par des dépendances complexes entre ses routines de calcul et une structure de données très riche de par le large panel d'applications qu'il permet de simuler.
Le parallélisme dominant dans ce code est MPI. Il se base sur la gestion des échanges d'informations entre les nœuds. La stratégie de décomposition de domaine utilisée dans le parallélisme à mémoire distribuée dans EUROPLEXUS a été étudiée et optimisée dans [56]. Le présent travail vient compléter ce projet d'optimisation du parallélisme dans EUROPLEXUS en exploitant un parallélisme à mémoire partagée au niveau des multi-cœurs des nœuds de calcul inter-connectés. Le recours à cette méthode parallèle à l'intérieur des sous-domaines associée avec l'utilisation efficace de la mémoire à travers des modèles de structures de données spécifiques permettent ainsi de repousser les limites de l'extensibilité de l'implémentation parallèle à mémoire distribuée.

L'influence de l'organisation de la structure de données sur les performances des codes de simulation a fait l'objet de nombreux travaux de recherche. Cette voie d'optimisation nous a guidés vers le développement d'une approche de réorganisation de la structure de données pour les codes de simulation numérique par éléments finis.
Notre approche est concrétisée par un modèle d'approximation de la hiérarchie mémoire. Ce modèle est basé sur le passage d'une structure de données globale inappropriée au principe de la localité spatiale dans le cache vers une structure de données conçue pour cet objectif. Ce passage est effectué de manière dynamique au cours de l'exécution du programme à raison d'une fois par pas de temps. Dans cette approche, nous construisons des blocs contigus. Ces blocs, que nous appelons *groupes*, contiennent l'ensemble des données nécessaires et suffisantes pour mener à terme le calcul d'un certain nombre d'éléments du maillage. Dans ce modèle que nous appelons `Par_gpe`, le nombre d'éléments d'un groupe est paramétrable en fonction de la taille de la mémoire partagée par unité de calcul et en fonction de la complexité du jeu de données à simuler.
L'évaluation des performances de ce développement dans un environnement d'exécution séquentiel sur des jeux de données variés nous a permis de valider l'apport de l'approche `Par_gpe` par rapport à la version de référence du code (version sans groupes). Notre approche `Par_gpe` consiste à imbriquer la boucle élémentaire responsable du déroulement de toutes les opérations élémentaires sur les éléments du maillage à l'intérieur d'une boucle externe sur les groupes d'éléments. Cela nous permet d'acquérir un espace de travail local avec des données contiguës favorisant la mise en place d'une stratégie de parallélisation efficace en mémoire.

Dans une première étape, nous avons conservé une exécution séquentielle de la boucle élémentaire imbriquée dans la boucle sur les groupes. En revanche, nous avons parallélisé cette boucle externe itérant sur les groupes. Dans la logique de notre approche, nous attribuons à chaque cœur un ou plusieurs groupes à traiter de manière totalement indépendante des autres cœurs. Ainsi, lorsqu'un cœur prend en charge l'exécution d'un groupe, il n'a pas besoin d'échanger avec d'autres cœurs

pour récupérer des données. Ce raisonnement reste valable aussi bien pour le cas d'une exécution parallèle statique que lorsqu'un équilibrage dynamique par vol de travail prend place. En effet, dans la version parallèle de notre modèle `Par_gpe`, les vols s'effectuent par groupe indépendant ce qui favorise le travail avec des données locales au cœur actif.
Pour implémenter notre version parallèle de l'approche `Par_gpe`, nous avons opté pour l'utilisation de la bibliothèque de programmation parallèle XKAAPI [2]. Intégrant un ordonnancement dynamique basé sur le vol de travail, XKAAPI nous permet d'équilibrer dynamiquement la charge sans dégrader la localité des données.
L'implémentation de ce premier niveau de parallélisme s'est avérée plus performante que l'implémentation parallèle standard de la bibliothèque XKAAPI dans le code EUROPLEXUS ; elle nous a permis de réduire le temps total écoulé dans l'exécution de la boucle élémentaire de près de 40%.

Ces résultats nous ont incités à tenter de réduire le temps d'exécution de la région parallèle. Ainsi, pour une meilleure accélération de l'exécution des groupes en environnement parallèle, nous avons opté pour la parallélisation du second niveau de boucle à l'intérieur de chaque itération sur les groupes. L'implémentation de cette stratégie revient à profiter de la localité des éléments d'un même groupe pour générer une seconde équipe de travailleurs au niveau de la boucle interne.
La notion de parallélisme imbriqué a été développée dans de nombreux environnements de programmation parallèle à mémoire partagée. Par exemple, elle a été intégrée dans OpenMP pour offrir la possibilité d'inclure des régions parallèles dans d'autres régions déjà parallèles. Les travaux de recherches sur ce sujet ont souligné la complexité de la parallélisation des nids de boucles pour plusieurs codes de calcul. Cette approche est difficile à mettre en œuvre dans le cas d'un équilibrage dynamique de la charge entre les différents niveaux de boucles. L'ordonnanceur dans ce cas doit prendre en compte la hiérarchie de la plateforme de calcul. Il faut veiller à ce que les échanges entre les différents niveaux de cette hiérarchie de boucles restent locaux. L'analyse des performances de cette méthode pour des jeux de données de tailles et de caractéristiques variées, nous a permis de cerner les limitations de cette technique. Nous avons pu, également, soulever des problèmes de contention mémoire qui étaient moins dérangeants dans la version à un seul niveau de parallélisme.

Cadre de la thèse

Ce projet de thèse s'est déroulé au sein du laboratoire de dynamique (DYN) au Commissariat à l'Énergie Atomique et aux énergies alternatives (CEA). C'est le fruit d'une collaboration entre la communauté des mécaniciens représentée par le laboratoire DYN et la communauté des informaticiens de l'Institut National de Recherche en Informatique et Automatique (INRIA) de Grenoble et du laboratoire d'informatique de Grenoble.
L'encadrement sur les thématiques reliées à la dynamique rapide a été assuré par Vincent Faucher au CEA ainsi que toute l'équipe EUROPLEXUS de DYN. Bruno Raffin, en tant que directeur de thèse, m'a encadrée, principalement, sur les aspects

2. http ://kaapi.gforge.inria.fr

de gestion de la mémoire, de programmation parallèle et d'évaluation des performances des codes en environnement parallèle. Dans le cadre de la parallélisation d'EUROPLEXUS et de l'intégration de la bibliothèque XKAAPI dans ce code, j'ai collaboré avec Thierry Gautier de l'équipe de Multi-Programmation et Ordonnancement pour les Applications Interactives de Simulation (MOAIS) qui fait partie de l'INRIA.
J'ai aussi eu l'occasion de collaborer avec des chercheurs et des doctorants de la Maison de la Simulation, à Saclay, sur des problématiques liées aux allocations mémoire et à la gestion du cache.
Dans le cadre de l'étude des problématiques liées à la sensibilité des performances de calcul aux politiques d'économie d'énergie employées sur les clusters de calcul, j'ai collaboré avec Gilles Grimaud, Pierrick Burret et d'autres chercheurs de l'Institut de Recherche en Composants logiciels et matériels pour l'Information et la Communication Avancée (IRCICA) à Lille.

Guide de lecture

Ce manuscrit est composé, principalement, de 3 parties. La première partie, "État de l'art", s'articule sur 3 chapitres qui invoquent les notions générales et les principaux résultats connus dans les thématiques en lien avec notre projet. Le premier chapitre est consacré à la présentation des machines à mémoire partagée. Puis, dans le deuxième chapitre, nous introduisons les environnements de programmation parallèle les plus courants. Enfin, le dernier chapitre de cette partie est introduit pour éclairer les fondements mécaniques sur lesquels s'appuie le code EUROPLEXUS.
La deuxième partie intitulée "Contributions", contient la valeur ajoutée de notre thèse en termes de développement et d'optimisation de code. Nous avons jugé opportun de débuter cette partie par un chapitre "État des lieux" où nous présentons le code EUROPLEXUS et nous décrivons l'organisation de sa structure de données. Cette description nous mène à introduire, dans le deuxième chapitre, notre approche d'optimisation de l'utilisation de la mémoire cache : l'approche `Par_gpe`. Les résultats que nous avons obtenus sont analysés et discutés dans la dernière partie de ce chapitre. Le troisième chapitre est voué à la présentation de notre stratégie de parallélisme imbriqué que nous avons implémentée dans EUROPLEXUS.
Enfin, la partie "Conclusion et perspectives" conclut le manuscrit par un bilan des principaux résultats obtenus suite à ces 3 ans de thèse. Nous proposons également quelques pistes intéressantes à explorer dans le cadre de travaux futurs.

Deuxième partie

Etat de l'art

Chapitre 1

Machines à mémoire partagée

Nous débutons ce chapitre par une description générale des architectures parallèles. Puis, nous focalisons notre étude sur les architectures à mémoire partagée. Nous commençons par détailler l'organisation hiérarchique de la mémoire. Ensuite, nous invoquons les notions de base sur les mémoires cache, leur principe de fonctionnement ainsi que les principaux algorithmes de gestion du cache.

Sommaire

1.1 Introduction

Une machine parallèle est un ordinateur constitué de plusieurs processeurs identiques ou non qui coopèrent et communiquent pour traiter des informations de manière simultanée.

1.2 Classification des architectures : Taxonomie de Flynn

Plusieurs travaux cherchent à classifier les architectures matérielles selon différents critères. La classification la plus courante est celle proposée par Flynn en 1972 [58]. Le critère utilisé dans cette taxonomie est basé sur le nombre de flots d'instructions et le nombre de flots de données pouvant être gérées par ces instructions. Selon ce critère, il a défini quatre types d'architectures [104, 128, 93] :

SISD (Single Instruction, Single Data) Ce type d'architecture est le moins complexe puisque un seul processeur exécute une seule instruction sur des données se trouvant dans une seule mémoire. Dans ces machines, il n'y a aucun parallélisme. Il s'agit de machines mono-processeur. L'architecture SISD est inspirée des architectures éponymes de Von Neumann décrites pour la première fois par ce dernier dans [130].

SIMD (Single Instruction, Multiple Data) Ce modèle utilise des architectures où plusieurs processeurs identiques sont contrôlés par une même unité de contrôle centralisée. Dans ces architectures, on applique sur un ensemble de données différentes une seule instruction de façon synchrone.
Parmi les architectures SIMD, nous retrouvons les architectures vectorielles et les processeurs graphiques GPU (*Graphics Processing Unit*).

MISD (Multiple Instruction, Single Data) Dans ce modèle, plusieurs instructions sont exécutées en même temps sur la même donnée. Sur le plan pratique, ces architectures ne sont pas courantes étant donné l'étroitesse de leur champ d'application.
Les architectures de pipeline sont basées sur le modèle MISD.

MIMD (Multiple Instruction, Multiple Data) Dans le modèle MIMD, plusieurs unités de calcul peuvent être regroupées dans le même système. Chaque unité dispose de son propre flot de données et de son propre flot d'instructions. Les interconnexions entre les différentes unités de calcul sont possibles dans la logique de ce modèle.
De par sa grande flexibilité, ce modèle est le modèle le plus répandu dans les systèmes parallèles actuels et c'est celui d'ailleurs que nous allons utiliser dans le cadre de notre travail. Pour cette raison, nous allons consacrer la prochaine partie à la présentation des différentes architectures basées sur ce modèle.

1.3 Architectures à mémoire partagée

Le modèle MIMD de la taxonomie de Flynn permet de différencier deux familles d'architectures en se basant sur le type de mémoires dont elles disposent. On distingue, ainsi, les architectures parallèles à *mémoire partagée* et les architectures parallèles à *mémoire distribuée*. Dans une architecture à mémoire distribuée, chaque processeur accède à son espace de stockage local. Aucun accès direct au contenu de la mémoire d'un processeur distant n'est possible. Les échanges entre les différents processeurs s'effectuent moyennant des envois et des réceptions de messages à travers des interconnexions. Nous revenons sur ces communications dans le chapitre suivant.
À la différence de la classe des architectures à mémoire distribuée, la classe des architectures à mémoire partagée réfère aux machines où les différents processeurs du système parallèle accèdent à un espace mémoire commun. Les opérations d'écriture et/ou de lecture effectuées par chaque processeur sont indépendantes et asynchrones entre les différentes unités de calcul. Ceci peut donc conduire à des conflits suite à des accès concurrents au même emplacement mémoire.
Dans le cadre de cette thèse, nous nous intéressons à ce type d'architectures.
Afin d'éviter l'accès d'une donnée partagée par différents processeurs à la fois, des outils de synchronisations sont utilisés. À ce titre, les développeurs utilisent des verrous, des sémaphores, des instructions atomiques et des barrières de synchronisation qui sont certes coûteux en matière de temps total d'exécution mais, inévitables pour gérer les accès concurrents.

1.3.1 Architectures à accès mémoire non uniforme

Un des points importants à traiter lors de l'étude des architectures à mémoire partagée est le coût d'accès de chaque processeur à la mémoire.
Ce coût peut être identique pour tous les processeurs qui se partagent la même mémoire. Dans ce cas, les accès mémoire sont uniformes. Il s'agit de la sous-classe des architecture à accès mémoire uniforme : *UMA* (Uniform Memory Access).
Si ce coût est dépendant de l'emplacement du processeur et de l'adresse mémoire à laquelle il accède, il est question d'une architecture à accès mémoire non uniforme : *NUMA* (Non Uniform Memory Access) [24]. Pour quantifier le temps d'accès mémoire dans ces architecture, on se base sur le facteur NUMA. Ce facteur informe sur le rapport entre le temps mis dans une architecture NUMA pour accéder à une mémoire distante et une mémoire locale. Il est défini formellement comme suit :

Remarque 1. *Lorsqu'un cœur c accède à une donnée placée sur un nœud NUMA n, sa latence (temps d'accès) est L_n^c. Si la donnée accédée est stockée dans sa mémoire locale, on définit par L_{loc}^c sa latence locale. Sous ces conditions, le facteur NUMA F_c est donné par l'équation 1.1 :*

$$F_c = \frac{L_n^c}{L_{loc}^c} \tag{1.1}$$

Réduire l'impact du facteur NUMA est un défi qui préoccupe de nombreux chercheurs. Dans la littérature, des travaux de recherche proposent des techniques qui

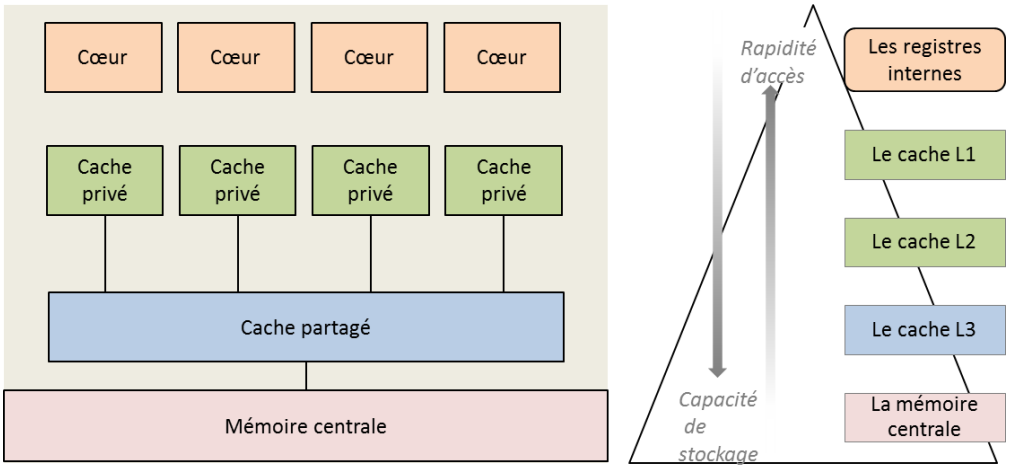


FIGURE 1.1 – Architecture d'une machine à mémoire partagée

permettent de limiter l'impact de ce facteur. Dans ce contexte, des stratégies de placement optimal des données par rapport à la localisation de l'unité de calcul qui les utilise ont été développées [97, 29, 113, 108].

1.3.2 Mémoires à organisation hiérarchique

Bien que son rôle paraisse simple, la mémoire présente une grande complexité architecturale et intègre des technologies diverses pour répondre au rythme d'évolution de la puissance de calcul des processeurs. Ces mémoire peuvent contenir dans leur hiérarchie des bancs de mémoire dynamique et/ou des mémoires caches. L'accès aux différents nivaux obéit à une topologie d'accès. Ces topologies peuvent être indépendantes ; par exemple dans les mémoire des séries UV-XX de SGI, la topologie d'accès aux mémoires dynamiques est indépendante des mémoires caches. Dans cette topologie chaque saut (hop) rajoute une latence. De ce fait, pour résoudre ce problème de latence, la mémoire doit pouvoir suivre le rythme du processeur. Il faut éviter de laisser ce dernier passer des cycles à attendre le chargement des données ou des instructions à partir de la mémoire principale.

Pour répondre à ce besoin, différents types de mémoires sont intégrés entre les unités de calcul et la mémoire centrale. Ces mémoires se distinguent par leurs capacités de stockage et leurs vitesses d'accès. Généralement, ces deux critères sont inversement proportionnels : les mémoires les plus rapides ont une capacité plus réduite et elles sont plus chères.
Le dilemme vitesse d'accès - capacité de stockage nous conduit à parler d'une organisation hiérarchique des mémoires dans les machines de calcul.
Voici ce que l'on peut rencontrer au fur et à mesure que l'on descend dans cette hiérarchie (cf. figure.1.1) : au sommet de cette pyramide (à droite dans la figure 1.1) se situent les registres internes. Ils sont considérés comme étant la mémoire la plus rapide et ils sont intégrés directement dans le circuit du processeur.
Dans le deuxième niveau de cette hiérarchie, on trouve les mémoires caches (appelées aussi antémémoires [133]) situées entre les registres internes et la mémoire centrale

(la mémoire RAM).

1.4 Mémoire cache

Introduite pour la première fois par M. V. Wilkes en 1965 [133], la mémoire cache est conçue pour combler la latence entre le processeur et la mémoire principale. Elle offre à ce dernier un accès rapide aux données et aux instructions les plus utilisées.

1.4.1 Les niveaux du cache

Dans les architectures actuelles, on distingue jusqu'à trois (voire même quatre) [1] niveaux de cache intégrés entre la mémoire principale et le processeur[17] :

- **Le cache L1** est intégré sur le circuit du processeur. Il est le plus rapide et le plus petit en matière de capacité de stockage (de l'ordre de quelques Ko). Ce niveau est généralement divisé en deux parties, l'une pour stocker les instructions et l'autre est réservée aux données ;
- **Le cache L2** a une capacité de stockage plus importante que celle du cache L1, mais avec une vitesse d'accès moins rapide. À la différence du cache L1, le cache L2 est généralement dédié au stockage des données et des instructions sans être physiquement divisé.
- **Le cache L3** quant à lui, est disponible seulement sur certains processeurs des machines haut de gamme. Il est partagé entre les cœurs du processeur. Dans les architectures les plus récentes, il est logé dans le même circuit que celui du processeur. Sa capacité atteint quelques Méga octets. Il est moins rapide que les deux premiers niveaux.

1.4.2 Principe de fonctionnement du cache

Lorsque le processeur tente de lire des données, il commence par vérifier si celles-ci sont disponibles dans son cache. Si c'est le cas, il récupère les données directement et il les utilise pour poursuivre son calcul. Il s'agit d'un succès de cache ou *cache hit*.
Dans le cas contraire, l'ensemble des mots qui représentent les données demandées est chargé dans le cache à partir de la mémoire principale selon la loi d'associativité adoptée par la politique de gestion du cache (cf. § 1.4.3). Cet ensemble de mots consécutifs ainsi chargé est appelé *ligne* de cache. Quand le processeur échoue à trouver la donnée dans son cache, le temps de récupération des mots manquants est plus long. On parle ainsi d'un défaut de cache ou *cache miss*[124].
Le diagramme de la figure 1.2, récapitule le principe de fonctionnement du cache.
Pour toute donnée présente dans le cache il existe une copie dans la mémoire principale. Quand le processeur fait référence à une donnée dans le cache et qu'il y accède en écriture, une opération de mise à jour du contenu de la mémoire principale aura lieu.

1. Cette organisation hiérarchique de la mémoire n'est pas générale, il y a des architectures qui n'ont pas de cache comme le processeur Tera [13]

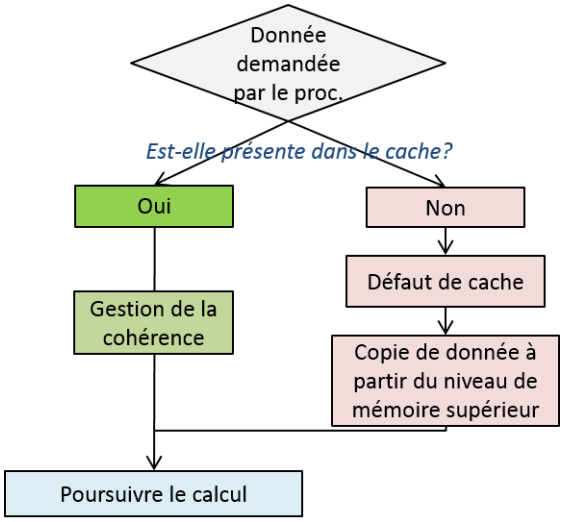


FIGURE 1.2 – Principe de fonctionnement du cache

Cette opération consiste à remplacer l'ancienne valeur de la donnée qui existe dans la mémoire du niveau supérieur (mémoire centrale) par la bonne valeur qui vient d'être calculée. Il existe différentes méthodes de mise à jour :

- **Écriture immédiate** (write through) : cette procédure consiste à recopier les données tout de suite après leur écriture dans la mémoire cache. Elle était largement utilisée dans les premiers caches mais elle tend à disparaître dans les nouvelles machines. Toutes les écritures se font dans la mémoire principale et engendrent une mise à jour directe des données présentes dans les différents niveaux du cache.
- **Écriture différée** (write back) : l'écriture en mémoire est effectuée lorsqu'on doit libérer la place occupée par la donnée pour y charger une autre selon l'algorithme de remplacement utilisé (cf. § 1.4.5). Cette technique est généralement utilisée au niveau des caches L2 et elle est considérée comme étant la plus performante car les écritures ne sont effectuées que lorsque la donnée a été modifiée. Chaque ligne possède un bit qui indique si une modification de la donnée a eu lieu ou pas.

Cependant, le bloc de données ainsi chargé suite à une requête du processeur ne peut pas être placé dans n'importe quel endroit du cache, il est dépendant de l'*organisation interne* du cache.

1.4.3 Lignes de cache et associativité

De par sa taille limitée, la mémoire cache ne peut pas contenir toutes les données de la mémoire centrale. La solution utilisée dans les architectures actuelles consiste à associer à chaque ensemble d'adresses contiguës de la mémoire centrale une adresse dans le cache.
L'associativité ou l'organisation interne du cache décrit la façon avec laquelle les données provenant de la mémoire centrale sont stockées dans le cache.
Ces données sont stockées dans la mémoire principale dans des blocs appelés éga-

lement *pages*. Chaque page est constituée d'un ensemble de lignes. Tout transfert entre la mémoire centrale et la mémoire cache s'effectue par ligne ou par ensemble de lignes. En pratique, il s'agit de la largeur de transaction entre le processeur et le système de gestion de la mémoire (contrôleur). Ceci est très dépendant d'un type de processeur (Intel, GPU, Nvidia ou ARM) voir d'une génération à une autre. Le transfert des données entre la mémoire cache et la mémoire centrale se produit lors des *défauts de caches* de manière automatique. Ce transfert d'informations entre les deux mémoires est géré par une politique de remplacement qui permet de décider quelle ligne doit être évincée pour charger une autre.

À une échelle plus petite, une fois dans le cache, des transferts explicites des données s'effectuent entre les niveaux du cache et les registres du processeur. Les transferts entre le cache et les registres sont déclenchés directement par les instructions mémoire. L'unité de transfert entre les lignes du cache et les registres internes est le *mot* (cf. figure.1.3). Cette unité est très variable sur les processeurs Intel où la largeur du mot dépend de l'instruction.

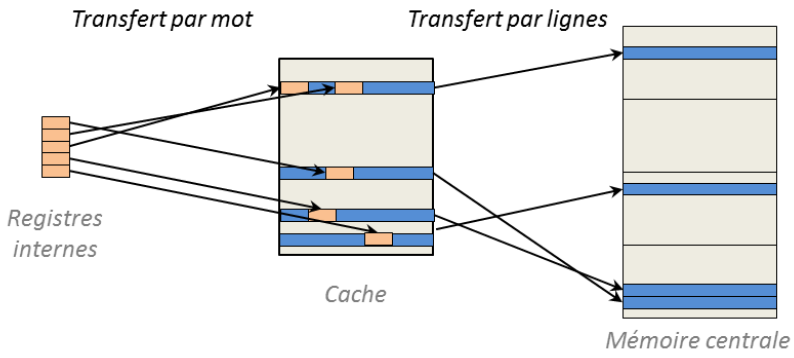


FIGURE 1.3 – Transferts de données dans une hiérarchie mémoire

Plusieurs techniques ont été développées dans le but d'assurer une gestion optimale entre la mémoire cache et la mémoire centrale. Dans les architectures actuelles, il existe principalement 3 méthodes utilisées :

La méthode de correspondance directe *(direct mapped)* : Elle consiste à affecter à chaque ligne du cache un bloc de la mémoire centrale de taille fixe (cf. figure1.4). Chaque bloc de la mémoire du niveau supérieur (mémoire centrale) ne peut correspondre qu'à une seule adresse de ligne de cache.
Lorsque le processeur demande une donnée, il sait directement dans quelle ligne la chercher. La disponibilité de cette information lui évite de perdre du temps à balayer toutes les adresses des lignes du cache pour vérifier la présence de la donnée en question.
Dans cette méthode, une ligne de cache peut être associée à plusieurs adresses de la mémoire du niveau supérieur. Cette associativité rend ambiguë la détermination de quel bloc mémoire est chargé dans la ligne du cache concernée à un instant donné. Pour lever cette ambiguïté, on stocke dans chaque ligne du cache une information supplémentaire appelée le *tag* permettant de préciser l'adresse de la mémoire principale qui correspond au contenu actuel de la ligne

du cache.
Cependant, le fait qu'une seule ligne de cache soit affectée à un bloc fixe de la mémoire RAM, pourrait être désavantageux. En effet, lorsque la taille du bloc mémoire utilisé par le processeur dépasse la taille de la ligne du cache, le processeur sera contraint de travailler sur une petite portion des données. Par conséquent, il sera obligé de faire plusieurs appels à la mémoire centrale pour charger une autre portion du bloc associé à cette adresse du cache.

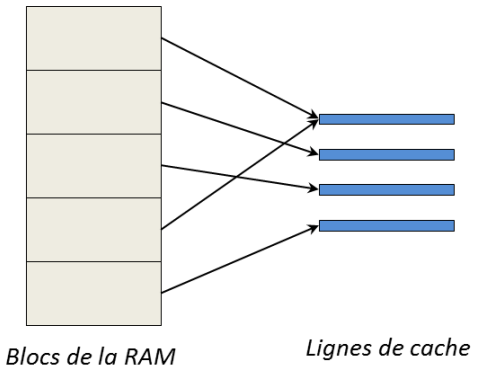


FIGURE 1.4 – Méthode de correspondance directe

La méthode totalement associative *(fully associative)* : Cette méthode est souple par rapport au choix du bloc à remplacer lorsqu'on a besoin de lire de nouvelles données à partir de la mémoire du niveau supérieur.
Cette méthode, contrairement à la méthode directe, consiste à charger chaque bloc de la mémoire du niveau supérieur dans n'importe quelle adresse du cache [118]. En effet, lorsque le processeur tente de lire une information, il effectue une recherche de la ligne concernée dans une table d'adresse dans le cache. Ceci donne accès à une multitude de combinaisons possibles pour le choix d'une adresse du cache où charger le bloc mémoire demandé.
Cette pluralité de combinaisons amplifie la complexité des circuits à concevoir dans le cadre des caches totalement associatifs par rapport à ceux des autres méthodes d'associativité.

La méthode associative par ensemble *(N-way associative)* : Un ensemble de lignes de cache peut être affecté à une même adresse de la mémoire du niveau supérieur. Le paramètre N fait référence à ce nombre de lignes de cache (cf. Figure1.6).
Le processeur vérifie la présence de la donnée dans les adresses des N lignes associées au bloc mémoire contenant l'information recherchée. Cette méthode d'associativité a été conçue pour compromettre la simplicité de la méthode directe et l'efficacité de la méthode totalement associative. Nous notons que les processeurs actuels ont en général des caches *N-way* associatifs.

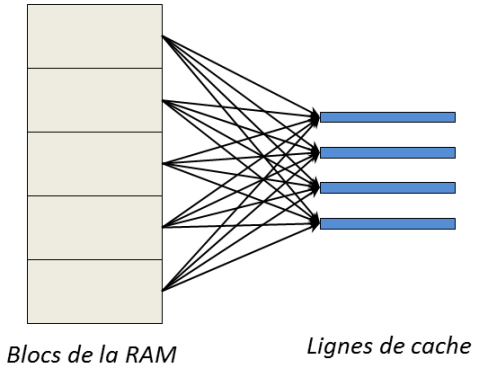


FIGURE 1.5 – Méthode totalement associative

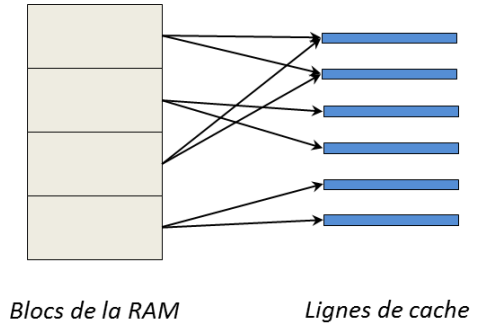


FIGURE 1.6 – Méthode 2-way associative

1.4.4 Les défauts de cache

On parle d'un défaut de cache (cache miss), lorsque le processeur demande une donnée du cache et que cette demande aboutit à un échec qui traduit son absence du cache (cf. figure 1.2).
Dans [75], M. D. Hill et A. L. Smith ont distingué trois types de défauts de cache :

Défauts de premier accès (*compulsory misses*) : ce sont des défauts obligatoires qu'on ne peut pas éviter. Ils se produisent lorsque le processeur demande une donnée pour la première fois.

Défauts de capacité (*capactity misses*) : ces défauts sont liés à l'associativité limitée du cache. Ils se manifestent quand la donnée accédée n'est plus disponible dans le cache suite à une expulsion par un algorithme de remplacement. La donnée utile est evincée du cache car la taille du bloc de données utiles dépasse la taille de la zone disponible dans le cache.

Défauts conflictuels (*set-conflict misses*) : ils sont dus à un adressage identique dans le cache pour des blocs ayant des adresses distinctes dans la mémoire de niveau supérieur. Ce type de défauts se manifeste lorsqu'il s'agit d'un facteur d'associativité strictement supérieur à 1 (cas d'associativité par ensemble et associativité totale). Dans ce cas, les lignes d'un ensemble associatif s'évincent mutuellement pour libérer de l'espace à leurs blocs respectivement associés.

Défauts de cohérence (*coherence misses*) : ils sont dus à l'invalidation des lignes du cache par les protocoles de cohérence (cf. section 1.4.6) afin de conserver la cohérence entre les différents caches des processeurs dans une architecture multi-processeurs.

Ces défauts de cache, quelle que soit leur catégorie, coûtent cher en matière de temps d'exécution et pénalisent les performances des codes de calculs. L'enjeu consiste donc à améliorer la localité des données dans le cache afin de réduire le nombre de défauts.

1.4.5 Les algorithmes de remplacement

À cause de sa capacité limitée, la mémoire cache ne peut pas contenir toutes les données et les instructions utiles à la fois. Il est indispensable de sacrifier des données qu'il faut évincer du cache afin de libérer de la place pour stocker les nouvelles données demandées par le processeur.
C'est le rôle des *algorithmes de remplacement des lignes de cache*. Ce problème ne se pose pas dans le cas d'une associativité directe (cf. section 1.4.3) puisque l'adresse de la ligne à remplacer est celle associée de manière unique au bloc mémoire contenant l'information demandée. Le recours à ces algorithmes est plutôt primordial lorsqu'il est question d'une organisation associative où les blocs mémoire peuvent être associés à plusieurs lignes de cache en même temps.
Dans la littérature, plusieurs types d'algorithmes de remplacement sont évalués et comparés[19, 10]. Ils reposent tous sur *le principe de localité* qu'on détaillera par la suite.
L.A Belady a étudié dans [19] différents algorithmes de remplacement de cache et il a proposé un *algorithme optimal OPT*. Dans ce formalisme, le choix du meilleur bloc à remplacer nécessite une connaissance a priori du prochain bloc à utiliser.
Parmi les algorithmes de remplacement de lignes de cache les plus répandus nous citons l'algorithme LRU (Least Recently Used) décrit dans [7]. Il consiste à remplacer le bloc le moins récemment consulté par le nouveau bloc demandé. Cependant, la complexité de cet algorithme est d'autant plus importante que la valeur du paramètre d'associativité N est élevée. Ce coût est dû au conflit qu'il pourrait y avoir entre les différentes lignes associées au même bloc mémoire. Pour contourner ce problème, plusieurs heuristiques *PLRU* (Pseudo LRU) ont été proposées [10][3].
L'algorithme *FIFO* (First In, First Out) est une alternative à l'algorithme LRU. Le choix de la ligne à évincer est basé sur l'ancienneté de cette dernière dans le cache : la ligne la plus ancienne sera la première à être éliminée du cache. La simplicité de cette décision a fait de l'algorithme FIFO une technique usuelle dans le marché des mémoires. Cependant, dans [117], J. E. Smith et J. R. Goodman ont prouvé que l'algorithme LRU l'emporte sur l'algorithme FIFO dans certaines configurations particulières. Le taux de défaut de cache obtenu pour deux configurations identiques s'est avéré plus élevé avec l'algorithme FIFO par rapport à celui du LRU.
Pour d'autres cas, la méthode aléatoire *Random* [105], se comporte mieux que les autres algorithmes. Cette technique consiste à choisir d'une manière aléatoire une ligne victime pour la remplacer par une nouvelle ligne.
La liste des algorithmes de remplacement que nous venons de présenter dans cette partie n'est pas exhaustive. D'autres algorithmes de remplacement de lignes de cache

existent dans la littérature.
Ce large panel d'algorithmes de remplacement met en relief la difficulté de ce problème qui s'accentue avec la complexité des processeurs qui ne cesse d'accroitre dans les architectures actuelles.

1.4.6 Cohérence du cache

Comme le cache gère des copies plus proche du processeur, avec les multi-cœurs, le problème est de gérer les modifications apportées sur l'ensemble de copies des différents cœurs. Ce problème nécessite l'application de protocoles spécifiques appelés protocole de cohérence du cache.

Lors d'une exécution parallèle en mémoire partagée, la première question qui se pose est comment rendre les modifications apportées sur une donnée visible par les autres cœurs. En fait, quand une tâche migre d'un cœur à un autre après avoir effectué des modifications sur les données disponibles dans la mémoire privée du premier. Dans ce cas, il est probable qu'elle réutilise une copie de la donnée qu'elle vient de modifier sur l'autre cœur avant sa migration. Étant donné que la valeur de la copie présente dans la mémoire du nouveau cœur n'est pas mise à jour, les résultats seront faussés.

Pour pallier ce problème, une stratégie d'invalidation de lignes a été mise en place. En effet, quand un cœur modifie une donnée, toute la ligne du cache qui contient cette donnée est invalidée. Ainsi, les lignes des caches des autres cœurs qui possèdent une copie de la donnée modifiée deviennent également invalides. Cette technique garantit que seules les données à jour seront utilisées dans la production des résultats. Ces stratégies d'invalidation nécessitent une vision cohérente de l'état des copies de la donnée partagée qui existent dans les différents caches privés des unités de calcul [121, 14]. Cette mission de vérification et d'invalidation est gérée par des protocoles spécifiques appelés *protocoles de cohérence de cache* [15, 50, 52, 51]. Dans ces protocoles, chaque ligne de cache possède différents états : elle peut être modifiée (M pour *Modified*), partagée (S pour *Shared*) ou invalide (I pour *Invalid*). Dans le premier cas (M), la ligne de cache possède la donnée la plus récente. Elle est la seule d'ailleurs à l'avoir. Toutes les autres copies de cette données ne sont plus valables. Ainsi, les lignes qui contiennent ces copies passent à l'état I. Si la ligne est logée dans le cache sans jamais avoir été modifiée et que d'autres cache hébergent cette ligne sans la modifier, alors son état est S. Dans ces conditions, la cohérence est assurée.

Dans cette voie de gestion de la cohérence, des communications entre les caches des cœurs prennent place pour informer les caches du changement d'état des lignes. Les principales méthodes de communication qui ont été développées dans ce contexte sont :

Protocole par inondation : [48] Pour assurer la cohérence des données dans son cache privé par rapport aux copies que possèdent les autres cœurs, chaque cœur récupère, dans un premier temps, la liste des caches qui contiennent la donnée en question. Ensuite, il envoie une demande à tous ces caches pour récupérer cette donnée. Suite à cette demande, parmi les réponse reçues, seule la première sera retenue, les autres réponses seront ignorées. Dans ce protocole, l'envoi des

requêtes aux autres caches est effectué en respectant l'ordre de proximité de ces derniers.

Protocole par espionnage (snooping-based protocols) : Pour assurer la cohérence des données dans son cache privé par rapport aux copies que possèdent les autres cœurs, chaque cœur suit l'état de partage du bloc des données en question en surveillant les communications à travers un bus commun à tous les cœurs. Il invalide les lignes contenant les données qui ont été accédées en écriture par un autre cœur. Cette famille de protocoles est la plus répandue dans les architectures multi-processeur qui possèdent un seul cœur par processeur et une mémoire partagée entre tous les processeurs à laquelle ils accèdent tous à travers un bus commun[14].

Protocole par répertoire (*directory-based protocols*) : Cette classe de protocoles consiste à répertorier les informations relatives à chaque ligne de cache dans un endroit unique appelé *répertoire*. Son utilisation dans les architectures hiérarchiques permet au nœud (ou processeur dans le cas d'une architecture multi-cœur) d'avoir toutes les informations sur les états des données stockées dans sa mémoire partagée. Ce modèle est adopté dans la plupart des machines à mémoire partagée actuelles.

Bien que la cohérence de cache dans les architectures multi-processeurs soit considéré parmi les paramètres les plus influents sur les performances des programmes, son étude sur des systèmes réels s'est avérée une opération très compliquée [49]. Pour cette raison, de nombreux travaux de recherche proposent des modèles analytiques pour évaluer l'effet de la cohérence des caches sur les performances des algorithmes parallèles exécutés sur des systèmes multi-processeurs. Pour plus de détails sur ce sujet, on pourra consulter [87, 76, 8, 31].

L'effet de cohérence de cache se traduit par la pénalité causée par les défauts de cache de cohérence. Ainsi, dans notre voie d'optimisation des performances de notre application, nous implémentons une approche qui vise à réduire le nombre de défaut de cache de cohérence. Cette technique permet de limiter le travail des cœurs des processeurs sur des données partagées dans un contexte d'exécution parallèle. Pour ce faire, nous favorisons le travail des cœurs sur des blocs **indépendants** de données **contiguës**. La réorganisation de la structure de données ainsi dans des blocs de petites tailles qui doivent tenir dans le cache du cœur qui les utilise permet de minimiser les opérations d'invalidation de lignes par le protocole de gestion de la cohérence.

Chapitre 2

Programmation parallèle

Si la pseudo loi de Moore[96] s'est avérée fiable dans la description de l'évolution de la puissance de calcul, ses limitations se sont manifestées par l'atteinte d'un seuil de finesse de gravure de transistors par circuit et par l'échauffement thermique dû à l'augmentation de la fréquence d'horloge. Ces limitations ont conduit à une nouvelle stratégie basée sur l'augmentation du nombre de cœurs plutôt que de transistors.
C'est l'ère du parallélisme généralisé...
Avant de décrire en profondeur les techniques de programmation parallèle efficaces dans le cache que nous allons détailler dans la deuxième partie de ce chapitre, il est nécessaire de commencer par présenter les environnements de programmation parallèle ainsi que les principales stratégies d'ordonnancement de tâches.

Sommaire

2.1 Introduction

Pour réduire le temps d'exécution de nos programmes, nous aimerions bien avoir des "ciseaux magiques" qui les découpent, répartissent parfaitement la charge et réalisent un parallélisme sur mesure pour ces derniers. Malheureusement, cette invention ne s'est pas encore concrétisée. Mais en attendant...
Chercheurs et développeurs se sont réunis pour trouver d'autres moyens plus réels afin de rendre possible l'exécution parallèle des applications. Nous étudions dans ce chapitre les techniques les plus courantes en programmation parallèle.

2.2 Programmation parallèle par passage de messages

La mise au point des applications parallèles pour des architectures extensibles sur des clusters de grande taille impose l'utilisation de bibliothèques d'échange de messages comme MPI (*Message Passing Interface*) [132] ou PVM (*Parallel Virtual Machine*) [122]. Cette famille de bibliothèque, en particulier MPI qui représente le standard de programmation parallèle pour les architectures à mémoire distribuée, permet d'exprimer explicitement la distribution de la mémoire ainsi que la gestion des communications à travers les bus du réseau. Dans ce standard, le programme est initialement découpé en différentes portions tel qu'illustre la figure 2.1. Chaque por-

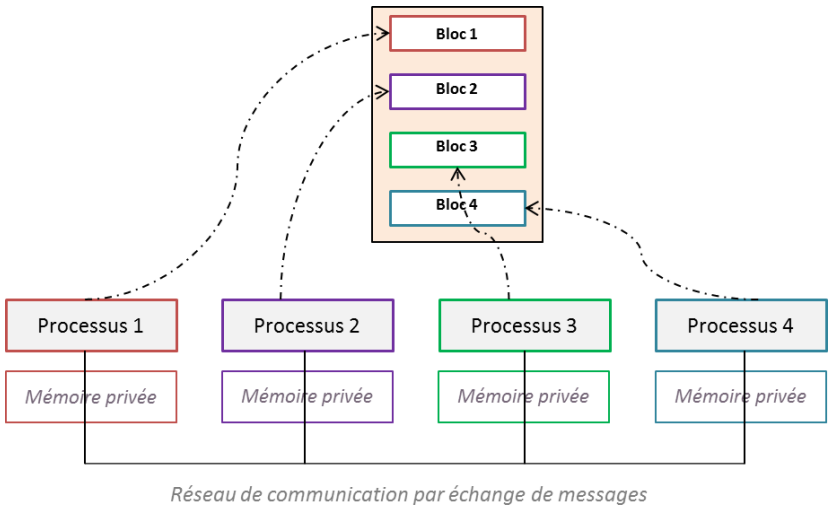


FIGURE 2.1 – Modèle de programmation parallèle avec MPI

tion est prise en charge par un processus autonome. Ce mode d'exécution dérive de la famille des MIMD (cf. section1.2). Les données sont privées par processus et elles résident dans son espace de stockage local. En cas de besoin d'une donnée qui appartient à un autre processus, le transfert de cette dernière de l'espace d'adressage d'un processus vers celui d'un autre est assuré par des opérations d'envoi (MPI_SEND) et de réception (MPI_RECV). Ces échanges sont basés sur des communicateurs. Ces

derniers peuvent être de type *point à point* lorsque la communication s'établit entre un émetteur qui envoie une donnée et un seul processus récepteur. Quand il s'agit de récepteurs multiples pour le même émetteur, le communicateur est dit *collectif*. Ce type de communication peut être remplacé par un ensemble de communications point à point.

Certes l'utilisation de MPI dans les applications industrielles comme EUROPLEXUS a beaucoup contribué au passage à l'échelle de ces dernières ; cependant, en contrepartie du grand nombre de nœuds sur lequel une application MPI peut s'exécuter, le temps de transfert des données via le réseau contraint le temps d'exécution global.
Une des plus grande difficulté que nous pouvons rencontrer en utilisant le modèle MPI est l'effort considérable d'adaptation du code pour qu'il s'apprête à supporter l'équilibrage dynamique de charge. La structure du code devient ainsi de plus en plus complexe et la programmation y devient difficile à gérer.

2.3 Programmation parallèle par tâches

2.3.1 Modèle de programmation par graphe de tâches

Les architectures parallèles à mémoire partagée offrent la possibilité à plusieurs flots d'instructions d'accéder à une zone mémoire commune pour communiquer entre eux. Ainsi, l'utilisation de processus léger (*thread*) pour encapsuler ces instructions permet de profiter de ces mémoires partagées.

2.3.2 Graphes de tâches

Pour modéliser une application et analyser les dépendances entre l'ensemble de ses instructions, on a souvent recours à une schématisation par des *graphes orientés*. Ces graphes sont formés de *nœuds* et d'*arêtes* (cf. figure 2.2).

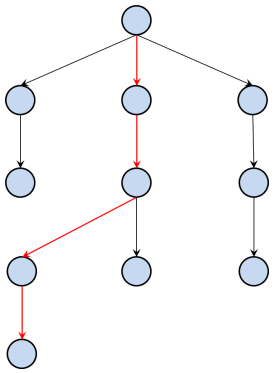


FIGURE 2.2 – Exemple de DAG

Chaque nœud représente un ensemble d'instructions qu'on appelle communément *tâche*. Ces tâches sont exécutées par des threads. Les arêtes du graphe décrivent les dépendances entre ses tâches. Le sens d'orientation d'une arête qui relie deux nœuds

du graphe traduit l'ordre d'exécution des tâches.
Nous nous intéressons dans le cadre de notre étude aux *graphes orientés sans cycles* (*DAG* pour *directed acyclic graph*).
Si on considère que toutes les tâches du DAG prennent le même temps pour s'exécuter, on réalise que le temps nécessaire pour l'achèvement de la dernière tâche du chemin le plus long du DAG (représenté en rouge sur la figure 2.2), correspond au temps nécessaire pour terminer l'exécution de l'ensemble des tâches du DAG.
Cette notion de graphe est systématiquement liée au sujet d'*ordonnancement des tâches*. L'allocation des ressources matérielles sur lesquelles les tâches du DAG peuvent s'exécuter fait partie des problématiques à gérer par un ordonnanceur de tâches. Ce dernier définit la date de début d'exécution de chaque tâche du DAG [86] tout en prenant en compte les contraintes de dépendance entre l'ensemble des nœuds du graphe. Dans ce contexte, l'objectif consiste à minimiser le temps d'exécution du système de tâches en exploitant d'une façon rationnelle les processeurs disponibles.

2.3.3 Heuristiques d'ordonnancement de tâches

De par la complexité des problèmes d'ordonnancement de graphes, la majorité des travaux de recherche sur cet axe se contentent de proposer des heuristiques. Ces heuristiques ne présentent certes pas des solutions optimales mais elles se placent à un facteur connu de l'optimal. Ce facteur est appelé *facteur de garantie de performance*.
Dans la littérature, la classe des algorithmes les plus utilisés en ordonnancement de tâches est celle des *algorithmes de liste* (*list schedulers*)[**?**] dont nous donnons la structure générale dans **Algorithme 1**. Dans le cas où le nombre de processeurs

Algorithme 1 : Algorithme d'ordonnancement de liste

```
1  Entrées : graphe des tâches DAG, l'ensemble d'unités de calcul libres C ;
2  trier les tâches prêtes dans une liste L ;
3  Tant que L ≠ ∅ faire
4        Pour i ∈ L faire
5              Si i est terminée Alors
6                    inclure le successeur de i dans L selon l'ordre de priorité ;
7              Fin Si
8        Fin Pour
9        Pour j ∈ C faire
10             affecter la tâche de plus haute priorité à j ;
11       Fin Pour
12 Fin Tant que
```

disponibles est inférieur au nombre de tâches à exécuter, ces dernières sont triées selon leurs priorités et attribuées aux ressources de calcul disponibles dans cet ordre. L'ordre de priorité est compatible avec les dépendances des données.
Parmi les algorithmes de liste les plus courants nous citons l'algorithme proposé par Graham [68] avec un facteur de garantie de performance de 2. Son principe est très simple : à partir d'une liste de tâches prêtes, tant qu'il existe des processeurs libres, on sélectionne une tâche de la liste des tâches prêtes et on l'exécute.

Lors de l'exécution d'une application parallèle via un algorithme de liste, dès qu'un cœur devient inactif, il retire du travail à partir d'une liste de tâches. Dans l'algorithme de Graham, cette liste de tâches prêtes est partagée entre l'ensemble des unités de calcul impliquées dans l'exécution de l'application. Cela place l'algorithme de Graham dans la famille d'*algorithmes de liste centralisée*. Les *algorithmes de liste décentralisée* sont ceux où chaque cœur possède une liste locale à partir de laquelle il sélectionne les tâches à exécuter. L'algorithme le plus connu dans cette famille est l'algorithme de vol de travail dans lequel les cœurs inactifs accèdent à la pile des autres cœurs pour leur voler une partie de leur travail.

2.3.4 Ordonnancement par liste décentralisée

Le concept du *workstealing* remonte aux travaux de F. W. Burton et M. R. Sleep lorsqu'ils ont introduit pour la première fois dans [27], la notion d'entraide entre les processeurs dans l'exécution des programmes.

2.3.4.1 Gestion de la DEQueue

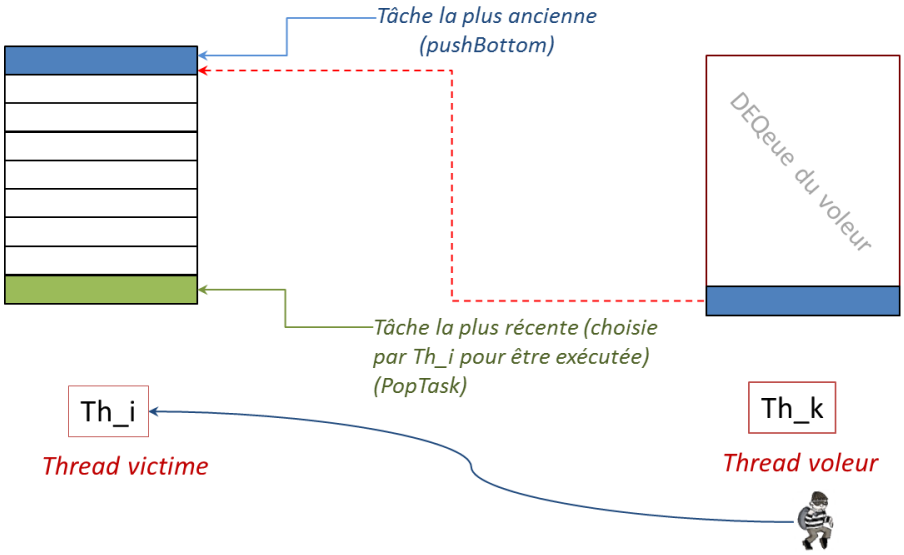


FIGURE 2.3 – Gestion de la DEQueue

Lors de son exécution, un thread TH_i crée des tâches et les place dans sa propre pile. Cette pile est de type *DEQueue* (Double-Ended Queue)[72]. Elle dispose de deux extrémités : TOP et $BOTTOM$(cf. figure2.3).
La gestion de cette DEQueue est assurée principalement par trois fonctions : pushBottom(), popBottom() et popTop().
Lorsqu'une nouvelle tâche vient d'être créée par TH_i, ce dernier appelle la fonction pushBottom() pour pousser la tâche à la fin de sa pile (extrémité BOTTOM de la pile).
Lorsque TH_i a besoin d'exécuter une tâche, il fait appel à la fonction popBottom() pour en tirer une à partir de sa propre pile (extrémité BOTTOM de la pile).

Cette tâche correspond à la dernière tâche ajoutée à la pile puisque l'algorithme d'ordonnancement de la pile locale est de type *LIFO* (Last In First Out). Cet ordre d'exécution local des tâches est en adéquation avec l'ordre séquentiel du programme. Lors de la consultation de la pile, TH_i pourrait se trouver face à l'un des scénarios suivants :

premier cas : sa pile locale contient encore des tâches prêtes. Dans ce cas, il choisit la tâche dans l'extrémité *BOTTOM* de sa pile. Ce choix correspond à la tâche la plus récente dans la pile.

deuxième cas : sa pile locale est vide. Dans ce cas, il se transforme en *voleur* et il tente de récupérer du travail sur un cœur victime choisi aléatoirement parmi les cœurs participant à l'exécution du programme.
Si la pile de la victime choisie est vide, le thread voleur continue à chercher une victime et il reste dans cet état jusqu'à ce qu'il réussisse à récupérer du travail sur l'un des cœurs impliqués dans l'exécution de l'application.
Si le voleur arrive à sélectionner une victime, il récupère une partie des tâches lui restant à faire à partir du haut de la pile de cette dernière (extrémité *TOP*). Ainsi, l'algorithme utilisé par le voleur dans la pile de la victime est de type *FIFO* (First In First Out). La quantité du travail à voler sera étudiée ultérieurement dans cette même section.

2.3.4.2 Vol de travail et choix de la victime

Dans un algorithme de vol de travail, le choix de la victime est compté parmi les points clés influents de la performance. Ce choix a été abordé dans de nombreux travaux de recherche, ce qui a donné lieu à l'implémentation de différentes stratégies qui s'y rapportent.

Vol de travail aléatoire : L'algorithme le plus trivial est celui basé sur le vol de travail aléatoire RS (*Random Workstealing*) [5, 16] (cf. figure 2.4).

Algorithme 2 : Algorithme de vol de travail aléatoire

```
1  Entrées : l'ensemble d'unités de calcul qui participent à l'exécution { 0, 1, ..., N } ;
2  Trouver ← faux ;
3  Tant que trouver = faux Faire
4        victime ← random { 0, 1, ..., N } ;
5        Si victime.Vide() = faux Alors
6              travail ← victim.PopTop()
7              trouver ← vrai
8        Fin Si
9  Fin Tant que
10 Retourner travail
```

C'est l'algorithme le plus courant dans les bibliothèques de programmation parallèle basées sur l'équilibrage dynamique de charge. En s'appuyant sur cet algorithme, R. D. Blumofe et C. E. Leiserson [23] ont démontré que l'implication de P threads dans l'exécution d'un programme nécessite un temps T_p

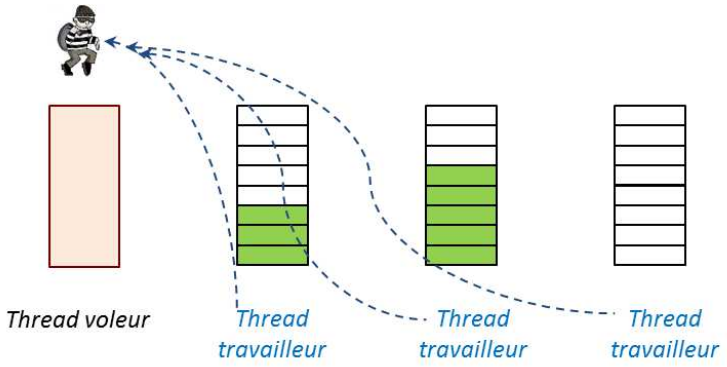


FIGURE 2.4 – Vol de travail aléatoire

évalué théoriquement par l'équation 2.1 :

$$T_p = \frac{T_1}{P} + O(T_\infty) \tag{2.1}$$

tels que :

T_1 **:** est le temps d'exécution de l'application avec un seul processeur ;

T_∞ **:** est le temps obtenu en impliquant un nombre infini de processeurs dans l'exécution du programme.

Vol de travail hiérarchique : De nombreux travaux de recherche se sont intéressés à la réduction du coût supplémentaire qui pénalise les performances lorsque des techniques d'équilibrage dynamique de charge sont mises en jeu. Pour réduire ce coût, M. A. Bender et M. O. Rabin [20] ont adapté l'algorithme du vol de travail implémenté dans la bibliothèque Cilk aux architectures hétérogènes intégrant des processeurs multi-cœurs et des processeurs graphiques (GPU) associés à des mémoires ayant une organisation hiérarchique.

La différence majeure entre la stratégie de vol pour les architectures homogènes et celle pour architectures hétérogènes est la condition d'autorisation du vol. En effet, dans le premier cas, si le voleur trouve que la pile de la victime est vide, il reste dans l'état voleur et il ne peut pas interrompre la victime durant son exécution. Le thread travailleur continue son exécution et le voleur reste inactif jusqu'à ce que le travailleur achève son exécution.

Dans le deuxième cas, si la pile de la victime est vide, on effectue une comparaison entre la puissance du processeur sur lequel s'exécute la victime et celui du voleur. Si le processeur du voleur est le plus puissant, la victime sera interrompue. Le travail de cette dernière sera transféré sur le processeur du voleur pour qu'il y soit exécuté plus rapidement.

Pour que cette méthode soit applicable, il faut que les puissances des différents processeurs participant à l'exécution du programme soient connues au préalable.

D'autres travaux de recherches se sont intéressés à l'étude des traces d'exécution des différents threads pour réduire les coûts du parallélisme et augmenter la localité mémoire.

J. Quentin[110, 109] a implémenté dans XKAAPI une stratégie basée sur l'historique des vols. Cette méthode analyse la quantité de travail effectué par chaque thread pendant le pas de temps T_{n-1} ainsi que le nombre de vols qui ont eu lieu. Ensuite, en se basant sur les résultats d'analyse de l'historique des threads, le moteur exécutif décide la redistribution du travail sur les threads au pas de temps T_n.

D'autres approches de type *CRS* (pour Cluster aware workstealing) [5] prenant en considération la hiérarchie de la plateforme de calcul ont été implémentés dans les algorithmes de vol de travail.
À ce titre, S. J. Min et C. Iancu [79] ont proposé leur algorithme haut niveau basé sur la sélection hiérarchique de la victime. Leur algorithme appelé *HotSLAW* permet d'effectuer un équilibrage dynamique de charge qui prend en compte la hiérarchie de la plateforme de calcul.
L'algorithme *HotSLAW* est une extension des travaux élaborés par Y. Guo et J.Zhao [69]. Dans ces travaux, les chercheurs ont proposé l'ordonnanceur *SLAW (Scalable Locality Adaptive Workstealing)*; une combinaison des deux algorithmes *Work-first* et *help-first*. SLAW autorise les vols uniquement à l'intérieur d'un domaine prédéfini par le développeur appelé "locality domain". Cependant, l'ordonnanceur HotSLAW permet une gestion plus précise des vols à travers les niveaux hiérarchique de la plateforme de calcul. En effet, cet algorithme attribue une priorité plus élevée aux vols locaux; plus on remonte dans la hiérarchie, plus la priorité devient faible. Ainsi, le thread voleur tente de voler en premier au niveau de son socket. S'il échoue à trouver du travail en local il relance sa requête vers les sockets voisins. Si cette deuxième tentative échoue il passe au niveau supérieur (nœud).

À ce stade, si on considère que le voleur a réussi à trouver une victime, l'enjeu consiste à définir la quantité de travail qu'il est autorisé de voler à partir de la pile de la victime.

2.3.4.3 Grain de vol

Combien de tâches peut-on voler à partir de la pile de tâches prêtes de la victime ? Pour répondre à cette question, N. S. Arora et H. D. Blumofe [16] ont fixé ce nombre de tâches à voler (*grain de vol*)à 1. Plusieurs chercheurs se sont basés sur cet algorithme pour proposer d'autres versions.

Dans ce même contexte, D. Hendler a implémenté l'algorithme *StealHalf* [71] qui consiste à permettre au voleur de récupérer la moitié du travail restant à faire dans la pile de tâches de la victime au lieu de voler par une seule tâche à la fois. L'étude des performances de cette méthode [46] (half-steal) a affirmé son efficacité par rapport à d'autres méthodes impliquant des grains de vol fixes (1-steal et 2-steal).

2.3.5 Environnements de programmation parallèle basés sur un équilibre de tâches dynamique par ordonnancement de listes

2.3.5.1 La norme OpenMP

En Janvier 1993, universitaires, constructeurs et utilisateurs se sont réunis pour définir un langage de programmation parallèle standard pour les machines à mémoire partagée [82]. Dans ce cadre, le langage *HPF*[1] (*High Performance Fortran*) [91] a été proposé pour les machines à mémoire partagée (version 1.0 pour le langage FORTRAN a été publiée en Octobre 1997). Cependant, des lacunes dues à l'immaturité des techniques de compilation utilisées, à la pauvreté de la boîte à outils offerte par ce langage, et à d'autres facteurs étudiés dans [82] ont empêché HPF d'atteindre le succès espéré.

Évitant les failles de son ancêtre le langage HPF et profitant des principaux apports de ce dernier, le langage *OpenMP*[2](*Open Multi Processing*) [34] a été proposé comme standard de programmation parallèle pour les machines à mémoire partagée [43].
Dans sa norme, OpenMP a gardé la notion de directive de compilation introduite dans le langage HPF. Ces directives sont identifiées par des sentinelles (préfixes) et conçues pour être visibles et prises en compte seulement lorsque la parallélisation OpenMP est activée. Dans le cas contraire, le programme séquentiel s'exécute normalement.

Nous montrons dans le pseudo-code suivant un programme écrit en Fortran qui permet de calculer le factoriel d'un entier n. La boucle *DO* est parallélisée avec OpenMP.

Exemple de programme Fortran parallélisé avec OpenMP

```
1   Program Exemple_OpenMP
2   !$use OMP_LIB
3   implicit none
4   integer, parameter :: n = 1000
5   integer :: i, fact
6   !$OMP PARALLEL
7   fact = 1
8   !$OMP DO
9   do i = 1, n
10          fact = fact * i
11  enddo
12  !$OMP END DO
13  !$OMP END PARALLEL
14  end program
```

OpenMP intègre des directives spécifiques pour la définition des sections parallèles (portions de code à exécuter par plusieurs threads simultanément).
Le comportement des régions parallèles obéit à un modèle de type *fork-join* : quand

1. http ://hpff.rice.edu/
2. http ://openmp.org/

un thread arrive à une région parallèle délimitée par les directives « OMP PARALLEL » et « OMP END PARALLEL », il crée une équipe de threads dont il devient le maître (master). Ces threads fils vont collaborer pour exécuter la section parallèle. Une fois la section parallèle achevée, les threads fils se terminent et le thread maître poursuit l'exécution de la région séquentielle (cf. figure 2.5). Le contrôle du

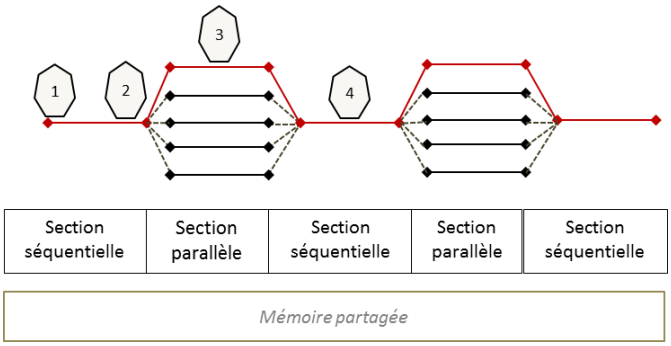


FIGURE 2.5 – Modèle de programmation parallèle avec OpenMP

placement des données par rapport au cœur qui les utilise est assurée par le paramétrage de la variable d'environnement *OMP_PLACES*. Cette dernière permet de lier le thread à un cœur ou à un processeur donné. Cela permet d'assurer l'accès des threads à des données locales et d'éviter, par conséquent les accès distants.

La politique d'ordonnancement peut également être gérée par l'utilisateur en configurant la variable d'environnement *OMP_SCHEDULE*. L'ordonnanceur répartit, ainsi, les itérations sur les threads disponibles selon la stratégie définie. Nous distinguons principalement 4 stratégies d'ordonnancement :

static : les itérations sont divisées en blocs comportant chacun *chunk*[3] itérations successives. Elles sont ensuite réparties sur les threads suivant l'algorithme d'ordonnancement circulaire (*Round-Robin*). Cette méthode est efficace dans le cas où les itérations présentent la même quantité de travail et que les threads mettent à peu prêt le même temps pour achever leur travail ;

dynamic : le travail est fourni aux threads à la demande. En d'autres termes, lorsqu'un thread n'a plus de travail, il demande *chunk* itérations consécutives. Dans ce cas, lorsque un thread devient inactif, il a la possibilité de récupérer une autre portion de travail à partir de la pile centralisée ou à un autre thread suivant l'implémentation. Un équilibrage de charge s'établit donc entre les threads malgré l'hétérogénéité des coûts des différentes itérations. Les coûts supplémentaires de parallélisme (*overhead*) sont plus élevés que ceux engendrés par la politique static ;

guided : il est fondé sur le même principe que l'ordonnancement *dynamic* sauf que la taille des blocs, dans le cas présent, n'est pas fixe. Au fur et à mesure que nous avançons dans le calcul, cette dernière décroît exponentiellement. La quantité du travail restant à faire est de moins en moins importante. De ce

3. le *chunk* désigne la plus petite taille de bloc d'itérations à l'exception du dernier qui possède une taille inférieure ou égale à la taille des autres blocs

fait, diviser le travail en de plus petites portions garantit (espérons-le) que les cœurs restent occupés pour la même période de temps. Cependant, comme la stratégie *dynamic*, cette méthode est caractérisée par un overhead élevé ;

auto : Le choix de la politique d'ordonnancement est délégué au compilateur et/ou au runtime.

Tel un standard de programmation parallèle, OpenMP offre différentes fonctionnalités au service du HPC. En revanche, l'ordonnancement par liste centralisée qu'il adopte dans la plupart de ses implémentations introduit des problèmes de contention au niveau de l'accès à la liste centralisée.
Pour remédier à ce problème de contention, la stratégie d'ordonnancement par listes partielles a été introduite. L'algorithme de vol de travail se base sur l'utilisation des listes décentralisées pour l'ordonnancement des tâches entre les différents threads.

2.3.5.2 Cilk

Cilk [4] [111] [60] [22] est un langage de programmation parallèle développé en C pour les plateformes multi-processeurs. Il est basé sur un algorithme d'ordonnancement par vol aléatoire de travail (RS).
Cilk est principalement conçu pour les architectures à mémoire partagée.

La description du parallélisme dans le langage Cilk s'effectue à l'aide du mot clé *spawn* qui précède directement le nom de la fonction. Ce mot permet d'identifier une fonction en tant qu'une procédure Cilk susceptible d'être exécutée en parallèle par des threads fils appelés par le thread père. Le thread père peut poursuivre son exécution en parallèle de l'exécution des threads fils. Le caractère asynchrone de cette exécution impose l'utilisation d'une synchronisation explicite afin de permettre à ce thread père d'utiliser le résultat des threads ainsi créés. Cette synchronisation utilise l'instruction *sync* qui oblige d'attendre que tous les fils terminent leurs exécutions.

L'exécution d'un programme sous le langage Cilk peut être présentée comme étant un DAG composé d'unités de parallélisme potentiel délimitées par un *spawn*. Cilk ne supporte pas les DAG avec des dépendances entre les threads parallèle. D'après Blumofe [22] la seule dépendance possible est celle entre un thread fils et son père.

2.3.5.3 IntelTBB

IntelTBB [5] (*Threading Building Blocks*) [93, 106, 127] est une bibliothèque de programmation parallèle développée en C++. Historiquement, le développement de cette bibliothèque a été foretement inspiré de Cilk de laquelle elle a hérité de nombreuses fonctionnalités comme *spawn* et *sync*. Elle est principalement dédiée aux architectures multi-cœurs. Son implémentation basique s'appuie sur un équilibrage dynamique de charge par vol aléatoire de travail.
De nombreux chercheurs se sont penchés sur l'étude des performances de la version originelle de cette bibliothèque basée sur le vol de travail. À ce sujet, G. Contreras et M. Martonosi ont essayé d'identifier les sources de dégradation des performances

4. https ://www.cilkplus.org
5. https ://www.threadingbuildingblocks.org/

du calcul de cet environnement. Ils ont démontré dans [39] que la déclinaison des performances de IntelTBB sur des plateformes multi-cœurs est directement liée à l'augmentation du nombre de cœurs participant à l'exécution du programme. Cette augmentation implique une amplification du coût des synchronisations imposées par les vols aléatoires entre les cœurs.
A. C. Iordan et M. Jahre [78] ont implémenté des stratégies alternatives pour la sélection de la victime afin de palier les problèmes du vol aléatoire rencontrés dans Intel TBB. Ainsi, dans [78], les auteurs proposent une technique de vol pseudo-aléatoire qui prend en considération les tentatives de vols qui n'ont pas abouti à une récupération de travail pour les exclure du champ de sélection des victimes lors de la prochaine tentative. Ils ont également implémenté une stratégie de vol qui prend en compte l'état d'occupation de la pile de tâches pour chaque cœur et d'utiliser cette information pour sélectionner la victime.

2.3.5.4 XKAAPI

XKAAPI[6] [53, 125, 63, 62] (*eXtreme Kernel for Adaptative, Asynchrounous Parallel and Interactive programming*) est à la fois un moteur exécutif et une bibliothèque de programmation parallèle développé au sein de l'équipe MOAIS[7]. Il est codé en langage C. Il offre des interfaces aux applications écrites en C et en C++ et il dispose également d'une interface pour les applications codées en FORTRAN. XKAAPI est adapté pour différentes plateformes, notamment les plateformes multi-cœurs et les plateformes multi-GPU.

Dans la suite de notre présentation de XKAAPI, nous allons nous intéresser particulièrement à son coté applicatif sur les architectures à mémoire partagée qui font l'objet de mes travaux de thèse. Comme nous l'avons souligné pour le langage Cilk, XKAAPI permet également la création "non bloquante" des tâches, ce qui autorise le thread créateur de la tâche de poursuivre son exécution en parallèle de la tâche créée.

XKAAPI permet la parallélisation des boucles imbriquées dans des boucles déjà parallélisées (parallélisme imbriqué).
Afin de contrôler le placement des threads qui travaillent sur le même niveau de boucle, on définit des domaines de localité (*locality domain*) regroupant un ensemble d'unités de calcul. Les tâches sont forcées à s'exécuter à l'intérieur de ces domaines. Chaque unité de calcul correspond à une *place*. Ainsi, le placement des threads sur les unités de calcul peut être effectué par la donnée de la liste de ces places. Une place peut référer à un **cœur**, un **socket** ou un **processeur**.
La variable d'environnement qui permet de définir ces places est `OMP_PLACES`. Elle est disponible dans l'environnement OpenMP et supportée par la bibliothèque XKAAPI. La déclaration du parallélisme dans XKAAPI repose sur l'appel à la fonction fortran `KAAPIF_FOREACH` (la syntaxe équivalente en C est `KAAPI_FOREACH`). Cette dernière prend comme arguments les bornes de la boucle à paralléliser, la routine contenant cette boucle (`kaapi_inner`) ainsi que les arguments de cette dernière. Cette fonction est précédée par des fonctions KAAPI qui servent à configurer l'environnement de

6. http ://kaapi.gforge.inria.fr
7. INRIA Grenoble

son exécution :

- `KAAPIF_SET_GRAIN` pour définir la taille du *grain séquentiel* ainsi que la taille du *grain parallèle*.
  Le grain séquentiel représente la taille du bloc de calculs à évaluer de manière séquentielle par thread à partir de sa pile locale (c'est l'unité d'extraction des tâches parallèles). Dans un contexte de parallélisme de boucle, un grain séquentiel représente le nombre d'itérations de la boucle parallèle à traiter par chaque thread de manière séquentielle.
  Nous notons que l'utilisation d'un grain séquentiel de taille trop petite engendre un surcoût de communication important et des création multiples de tâches dans la pile de chaque thread. Cependant, une taille de grain trop élevée limite le degré de parallélisme en empêchant la distribution du travail sur un nombre plus important de threads.
  Le grain parallèle est étroitement lié à l'extraction du travail au moment du vol. Il est généralement plus grand que le grain séquentiel pour éviter les vols de petites tailles dont les coûts de parallélisme (coûts de création de tâches et coûts de migration de travail au moment du vol) deviennent plus importants que ceux estimés en bloquant les vols entre les threads).
- `KAAPIF_SET_POLICY` pour définir la stratégie d'ordonnancement des boucles que ce soit au moment de la première distribution du travail sur les threads au début de la section parallèle ou au cours de l'exécution parallèle pour l'équilibrage de la charge par vol de travail. Dans la version 3.0 de XKAAPI (la version que nous utilisons ici), il existe deux type d'ordonnancement pour les boucles parallèles : la stratégie d'ordonnancement statique `KAAPIF_POLICY_STATIC` et la stratégie d'ordonnancement adaptative `KAAPIF_POLICY_ADAPTIVE`. La première n'intègre pas d'équilibrage dynamique de la charge et les vols n'y sont pas autorisés. En revanche, la deuxième politique d'ordonnancement correspond à une distribution initiale du travail sur les threads de manière identique à celle utilisée dans OpenMP, mais elle permet un équilibrage dynamique de la charge par vol de travail au cours de l'exécution de la boucle parallèle.
- `KAAPIF_SET_DISTRIBUTION` permet de définir la distribution des itérations de la boucle parallèle au sein d'un même niveau hiérarchique de boucles imbriquées. Cette fonction permet également de limiter le domaine des vols à l'intérieur d'un même niveau de boucle ou d'autoriser des vols inter-niveaux (les vols inter-niveaux n'ont pas été intégrés dans la version de XKAAPI utilisée dans ce travail).
- `KAAPIF_SET_AFFINITY` permet de contrôler l'affinité des tâches implicitement créées du `FOREACH`.

Grâce à l'utilisation judicieuse des `KAAPIF_SET_AFFINITY` et de `KAAPIF_SET_DISTRIBUTION` qui vont guider la stratégie de vol, l'ordonnanceur dynamique de XKAAPI offre plusieurs possibilités de gestion de vols pour les différents niveaux de boucles parallèles. Ainsi, les threads travaillant sur un même niveau de boucle peuvent profiter d'un équilibrage par vol de travail à l'intérieur d'un domaine défini au préalable pour éviter de voler du travail à partir des cœurs distants. Cet ordonnancement se place dans le cadre de notre stratégie d'optimisation du parallélisme à mémoire partagée dans EUROPLEXUS par l'implémentation d'un niveau de boucle externe autour de

la boucle élémentaire (cf. chapitre 6).

2.4 Les langages PGAS, des langages émergents

Après avoir fait le tour d'horizon des points phares de la programmation parallèle pour les machines à mémoire distribuée et pour les architectures à mémoire partagée. Il est intéressant de noter qu'il existe dans la littérature de nouveaux paradigmes de programmation parallèle qui permettent de tirer profil des avantages de ces deux familles de langages. Il s'agit des langages *PGAS*.
PGAS (Partioned Global Adress Space)[44, 12] est un modèle de programmation parallèle par espaces d'adressage globaux partagés.
Ce modèle est basé sur un espace mémoire virtuellement partagé entre les différentes unités de calcul. Cet espace commun contient les adresses des données de tous les cœurs. Il permet, ainsi, l'écriture directe, dans la mémoire d'un nœud distant à partir d'un autre nœud dans des architectures à mémoire distribuée. Le modèle PGAS permet également la mise en place d'un système de communication à travers les nœuds d'une grappe de calcul. Cette flexibilité simplifie largement la programmation pour les plateformes à mémoire distribuée par rapport à MPI.
Pour assurer une bonne localité des données, une information sur la localité de chaque donnée par rapport au cœur qui la demande est introduite dans la table d'adressage globale.
De nombreux langages de programmation sont basés sur le modèle PGAS. Le langage HPF [91] et le langage ZPL [89] font partie des premiers langages mettant en œuvre ce modèle (dans les années 90).
Vers la fin des années 90 et les débuts des années 2000, Co-Array Fortran (CAF) [103] (une extension parallèle du langage de programmation Fortran 95), UPC [38] et Titanium [74] ont adopté la version dite *originale* du modèle PGAS. Ces trois langages ont proposé des extensions pour l'espace d'adressage global basé sur le modèle SPMD de la taxonomie de Flynn. Ces extensions ont, respectivement, concerné les langages Fortran, C et Java, dans lesquels des implémentations qui prennent en compte la structure de données ainsi que les communications ont été mises en place.
À partir de 2004, de nouveaux langages qui ne sont pas basés sur des extensions des langages de programmation, à la différence des langages UPC, CAF, et Titanium, ont été implémentés dans le cadre du projet HPCS (*High Productivity Computing Systems*). L'objectif principal de ce projet était de mettre en œuvre des systèmes portables dotés d'une grande robustesse et d'une facilité de programmation. Derrière ces caractéristiques souhaitées, l'enjeu était de réaliser un meilleur gain en performance par rapport aux langages existants. Dans ce cadre, trois langages PGAS ont été développés ; le langage Chapel [32] développé par Cray, le langage X10 [35] développé par IBM et langage Fortness [11], principalement, développé par Sun Microsystems. D'autres langages PGAS ont été, récemment, développés en dehors du projet HPCS comme le langage XCalableMP [84]. Ce langage est le plus récent des langages PGAS. Il est dédié aux applications Fortran et C. Ce langage ressemble, dans son implémentation, au langage OpenMP. Il est basé sur des directives à insérer dans la version séquentielle du code pour construire une version parallèle.
L'atout du modèle PGAS est qu'il permet de bénéficier d'une programmation paral-

lèle générique qui ne dépend pas de l'architecture de la plateforme de calcul (mémoire distribuée ou mémoire partagée). Bien qu'il ne soit pas encore très répandu dans le monde de la programmation parallèle, il semble être prometteur pour la programmation des futures architectures parallèles de plus en plus complexes. L'implémentation de ce modèle permet de masquer la topologie de la machine cible afin d'augmenter la portabilité de l'application sur différentes architectures matérielles. Cependant, cette généricité n'est pas toujours bénéfique et elle risque d'engendrer une dégradation des performances par rapport à une stratégie de placement des données et des threads optimisée selon les spécificités de l'application et de la plateforme de calcul sur laquelle elle s'exécute.

En guise de "petit debriefing" autour des environnements de programmation parallèle

La diversité des techniques d'ordonnancement et le large panel des langages parallèles qui ne cessent d'évoluer ont permis aux développeurs et aux utilisateurs de trouver ce qui correspond le mieux à leurs besoins. En ce qui concerne nos besoins en matière d'optimisation des simulations en dynamique rapide, notre choix a été dirigé vers la bibliothèque de programmation parallèle XKAAPI. Ce choix est fondé sur diverses considérations. D'un point de vue architectural, les machines à mémoire partagée sont la première cible de ce langage ce qui correspond à la nature de notre plateforme pour l'implémentation des travaux de la thèse.

D'un point de vue logiciel, pour assurer l'équilibrage dynamique de la charge dans les applications de simulation en dynamique rapide, le workstealing implémenté dans XKAAPI semble être la technique la plus adéquate. Elle nous permet de profiter d'une programmation plus facile grâce à la gestion implicite de ces synchronisations et nous évite de complexifier davantage la structure du code comparé à une implémentation d'un équilibrage dynamique avec MPI. D'autre part, la précision dans la gestion de l'affinité processeur offerte par XKAAPI, nous a donné l'occasion de mettre en avant les apports de notre approche d'optimisation de l'utilisation de la mémoire en environnement parallèle. Nous nous sommes basés dans le cadre de ce travail sur une implémentation spécifique de XKAAPI qui supporte l'équilibrage dynamique de charge dans un contexte de parallélisme imbriqué. Nous avons intégré cette version de XKAAPI dans le code EUROPLEXUS.

D'autres versions parallèles ont été implémentées dans EUROPLEXUS dans le cadre de la thèse. Ces implémentations nous ont servi comme référence pour situer les performances de notre bibliothèque expérimentale XKAAPI par rapport à un standard de programmation parallèle pour les mémoires partagée OpenMP.

2.5 Parallélisme et optimisation du cache

La performance d'une application parallèle est étroitement liée à la manière dont on utilise la hiérarchie de la mémoire cache. En effet, une bonne organisation des données en mémoire qui respecte l'ordre chronologique de leur utilisation par l'unité de calcul permet d'augmenter l'efficacité de l'application et contribue à son accélération.

2.5.1 La localité du cache

Le chargement des données à partir de la mémoire centrale est une opération coûteuse qui engendre un ralentissement de l'exécution de l'application. Pour rentabiliser ce coût, il faut profiter des données tant qu'elles sont présentes dans le cache. Pour arriver à cette fin, on essaye de profiter des principes de localité spatio-temporelle :

localité spatiale : si le processeur consulte une case mémoire à un instant t, la probabilité qu'il accède à des cases voisines dans les instants suivants est importante ;

localité temporelle : si le processeur demande une donnée d, cette donnée sera réutilisée par ce dernier dans un instant proche avec une forte probabilité.

Ces deux principes constituent les deux axes majeurs sur lesquels reposent l'efficacité des caches.
Nous exposons, dans la partie suivante, les principaux travaux de recherche qui ont été mis en œuvre pour assurer cette efficacité.

2.5.2 Optimisation de la localité du cache par réorganisation des données en mémoire

De nombreux chercheurs se sont penchés sur la problématique de l'optimisation de l'utilisation de la mémoire en mettant en œuvre des stratégies de réorganisation des données.

2.5.2.1 Modification du placement des données en mémoire

Nous désignons par *layout* la manière dont les données sont stockées en mémoire. Un choix judicieux de ce layout a un impact direct sur l'efficacité du cache. Dans cette voie d'optimisation, l'enjeu consiste à organiser correctement les données afin d'éviter les accès non contigus. Il existe différentes façons de réordonner les données d'une application. Nous présentons ici les trois méthodes les plus répandues.

Le modèle SOA (Structure Of Arrays) : désigne une organisation de la structure de données sous forme de *structure de tableaux*. Ce modèle est, selon les développeurs de Intel [21], le modèle le plus adapté aux architectures SIMD dans lesquelles une seule instruction sera appliquée à différentes données (par exemple : dans certains codes de simulation, on calcule les vitesses de tous les éléments d'un maillage, ensuite, on calcule les accélérations de tous les éléments ....)
L'organisation SOA est parfaitement exploitée dans le cas des applications qui répondent à ces 3 critères :

- Le nombre de champs visités lors du parcours des tableaux de données est considérablement faible par rapport au nombre total de champs.
- La taille du tableau est suffisamment grande pour dépasser la taille des caches L1 et L2. En effet, si cette taille n'est pas assez grande, le compilateur va insérer des données inutiles (*padding*) pour garantir que chaque donnée du tableau soit alignée sur le bon nombre d'octets par ligne. L'ajout du

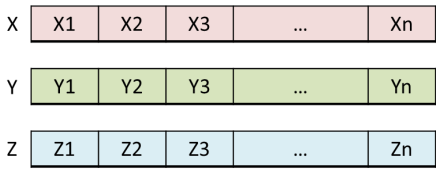


FIGURE 2.6 – Organisation de la structure de donnée en SOA

padding est une technique généralement utilisée dans les compilateurs afin d'éviter les défauts de cache conflictuels.

- Les accès au tableau sont séquentiels.

Si on considère une structure de données qui contient les coordonnées *X, Y, Z* de n points, dans ce cas, une organisation de cette structure en SOA consiste à placer toutes les valeurs X_i, Y_i et Z_i tel que $i \in [\![1; n]\!]$ dans des tableaux séparés comme le montre la figure 2.6.

Le modèle AOS (Array Of Structures) : désigne une organisation de la structure de données sous la forme d'un *tableau de structures* (cf. figure 2.7). Ce modèle est dédié aux applications disposant d'une structure de données de grande taille.

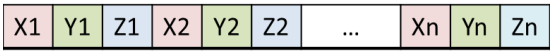


FIGURE 2.7 – Organisation de la structure de donnée en AOS

Pour tirer profit de cette organisation, il faut que tous les champs des structures stockées dans le tableau soient fréquemment utilisés. Cette condition n'est, néanmoins, pas toujours satisfaite. Pour remédier à ça et éviter de charger des données inutiles dans les tableaux de structures, T. M. Chilimbi et B. Davidson[36], ont proposé une technique de séparation des données des structures (*structure-splitting*). La technique structure-splitting est basée sur la catégorisation des données des structures en deux classes selon leurs fréquences d'accès par le processeur : la classe *HOT* qui contient les données des structures les plus fréquemment consultées et la classe *COLD* dans laquelle les données peu fréquemment consultées sont stockées. Ainsi, on construit des blocs de la taille du cache qui ne contiennent que les données fréquemment consultées ; les autres données sont stockées dans un bloc à part. Cette technique optimise le contenu du cache et évite le remplissage du cache avec des données peu utilisées lors du chargement des données fréquemment demandées dans les lignes du cache.

Dans la littérature les avis divergent quand à l'efficacité de ces modèles : Par exemple N. Govender et D. N. Wilke ont prouvé, dans [67], que le modèle SOA est trois fois plus performant que le modèle AOS pour une simulation réalisée sur une architecture GPU. Cet avis n'a pas été partagé par M. B. Giles et G. R. Mudalige [66] qui ont affirmé la supériorité du modèle AOS par rapport au modèle SOA sur le même type

d'architectures. Les auteurs ont souligné l'efficacité de la méthode AOS dans le cas des applications basées sur des données de maillage. En effet, dans la plupart de ces applications, les calculs s'effectuent maille par maille. On a donc intérêt à ordonner les données relatives à une maille dans une suite contiguë de cases du tableau.

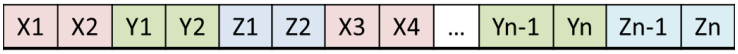


FIGURE 2.8 – Organisation hybride de la structure de donnée

Ces avis opposés prouvent que le choix du modèle de la structure de données permettant de réaliser les meilleures performances est étroitement lié aux spécificités de l'application. Afin de ne pas être tranchant sur ce choix et risquer de perdre en performance, il est recommandé de profiter des avantages de chacune de ces méthodes. Pour ce faire, l'adoption d'une approche hybride SOAOS (cf. figure 2.8) qui permet de combiner les deux méthodes pour un meilleur gain pourrait être bénéfique. Dans ce contexte, de nombreux travaux de recherche ont opté pour une implémentation générique qui permet de basculer entre les deux modèles en agissant sur quelques paramètres selon les besoins de l'application. Ainsi, une étape de paramétrage (ou d'auto-paramétrage) peut être mise en place pour effectuer ce choix en se basant sur des données de performance préalablement prélevées.

2.5.2.2 Re-numérotation du maillage

Les simulations numériques travaillent souvent avec une discrétisation de l'espace simulé suivant un maillage. A l'issue de cette étape, nous définissons des éléments géométriques qui décrivent de la manière la plus fidèle possible le milieu continu d'origine. Cette discrétisation est, communément, la tâche confiée à un outil spécifique appelé *le mailleur*. Il utilise des algorithmes particuliers pour optimiser la numérotation des mailles et des nœuds selon des considérations bien définies (frontière du domaine, finesse exigée, ...). Cependant, en dépit des optimisations effectuées par le mailleur selon les contraintes géométriques pour essayer de conserver au mieux la localité spatiale, la projection de l'espace multidimensionnel dans un espace unidimensionnel (en mémoire) est une tâche de grande ampleur.
Conscients de cette complexité, les développeurs des codes de simulation ont opté pour la dissociation de cette dernière de l'ensemble des tâches confiées au mailleur. Ils se sont penchés sur le post-traitement des fichiers de maillage en exploitant des techniques de re-numérotation avant de les passer en entrée des codes de calcul.

Un des mailleurs utilisés avec le code de simulation EUROPLEXUS, dans lequel les travaux de cette thèse ont été implémentés, est CAST3M [8]. Ce mailleur implémente différents algorithmes de re-numérotation pour les nœuds des éléments du maillage. Ces algorithmes permettent de réduire l'encombrement mémoire des matrices creuses associées aux systèmes linéaires à résoudre. L'algorithme Cuthill McKee avec ses deux versions directe et inverse [42, 41] est le plus connu d'entre eux. Le principe de l'algorithme de *Cuthill McKee direct*(CM) est le suivant : on

8. http ://www-cast3m.cea.fr

choisit un premier sommet auquel on attribue le numéro 1. Ensuite, on attribue à ses voisins qui n'ont pas encore été numérotés les numéros successifs, dans un ordre croissant (numéro 2, numéro 3, ...). Dans le cas où plusieurs sommets sont situés dans le premier niveau de voisinage du nœud numéro 1, on numérote, en premier, les nœuds qui possèdent le moins de voisins non encore numérotés. L'algorithme *Cuthill McKee inverse* (RCM pour *Reverse Cuthill McKee*) est basé sur l'inversion de l'ordre de numérotation des nœuds obtenu par la méthode directe de McKee.
Dans [90], Wai-Hung Liu et Andrew H. Sherman ont comparé les profils obtenus par ces deux algorithmes pour des *matrices creuses*. Ils ont prouvé que la méthode RCM est plus performante que la méthode CM. La méthode RCM permet d'obtenir *un profil* plus réduit. Elle est, par conséquent, moins encombrante en mémoire que la méthode CM.
Notons que, dans le cas du code EUROPLEXUS, le maillage évolue dynamiquement pendant chaque pas de temps. Ceci nécessite une adaptation du mailleur pour une mise à jour des numéros des éléments ainsi que ceux de leurs voisins pour maintenir la localité spatiale des données en mémoire. Cette solution induit une multiplication des coûts supplémentaires de re-numérotation des éléments contribuant à la dégradation des performances de calcul. Ainsi, pour limiter nos interventions au niveau du code de calcul et afin d'éviter ces coûts supplémentaires de re-numérotation, nous avons développé dans l'approche que nous présentons dans ce document, une technique permettant une mise à jour des numéros des voisins des éléments à chaque pas de temps pour assurer une bonne localité en mémoire.

2.5.2.3 Les *Space-filling curves*

L'utilisation des courbes remplissant l'espace (*Space-filling curves*) [115] est une technique de réorganisation des données de manière récursive introduite par Peano en 1890. Cette technique consiste découper l'espace multidimensionnel en plusieurs blocs de manière récursive et à tracer un chemin continu qui passe une seule fois par ces blocs. La projection des blocs de ce chemin dans l'espace unidimensionnel de la mémoire permet de conserver au mieux la localité spatiale des mémoires caches.
De nombreux chercheurs se sont inspirés de la méthode de Peano pour construire d'autres courbes remplissant l'espace. Les courbes de Hilbert sont comptées parmi les courbes qui respectent le mieux la localité. Nous représentons, dans les figures 2.9 et 2.10, un exemple de deux itérations d'indexation avec les courbes de Hilbert.

En se basant sur cette numérotation, nous assurons que, dans un intervalle d'indices donné, la majorité des éléments qui font partie de la liste sont des voisins directs entre eux. Ceci implique une correspondance entre l'ordre des éléments numérotés dans l'espace multidimensionnel représenté par la grille du maillage et l'espace unidimensionnel de la mémoire.

De par son efficacité en mémoire, la méthode de Hilbert a, particulièrement, fait l'objet d'une multitude de travaux de recherche [94, 28, 101, 25, 115]. Originellement, ces courbes ont été conçues pour le cas 2D. Ensuite, plusieurs travaux ont proposé des versions généralisées multidimensionnelles de ces courbes. Nous citons, à ce propos, les travaux de thèse de G. Nguyen [101]. Dans ces travaux, l'auteur propose, effectivement, une généralisation de la courbe de Hilbert élargissant son champs d'applications pour des cas multidimensionnels, dans le domaine du traite-

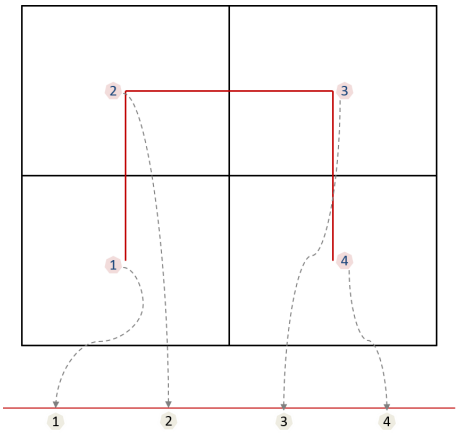


FIGURE 2.9 – Première itération de la courbe de Hilbert

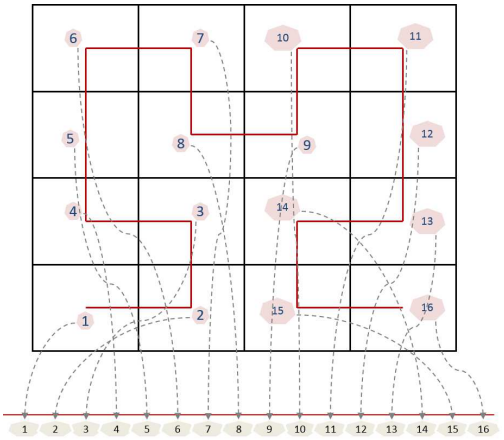


FIGURE 2.10 – Deuxième itération de la courbe de Hilbert

ment d'images. L'implémentation proposée est restée, toutefois, fidèle au principe de localité spatiale qui caractérise la méthode de Hilbert.

Dans la littérature, d'autres méthodes de numérotation sont proposées telles que la courbe de Lebesgue [116], la courbe de Moore [95] et la courbe de Sierpinski [115]. Elles dérivent toutes de la courbe originelle de Peano.

2.5.2.4 Les algorithmes Cache-oblivious

Bien que le caractère universel des algorithmes *cache oblivious* favorise leur portabilité sur différentes architectures, les algorithmes *cache aware* sont dotés d'une meilleure connaissance des caractéristiques de l'architecture cible. Ces dernières sont jugées, en général, plus performants que les algorithmes cache oblivious [83].

Pour cette raison, dans le cadre de cette thèse, nous avons opté pour une méthode cache aware lors de l'implémentation de notre approche d'optimisation de l'utilisation du cache. En revanche, notre approche implique des paramètres (cf. chapitre 5) permettant sa portabilité sur d'autres architectures par le simple réglage de ces derniers. Nous abordons, dans la suite de cette section, les principales techniques d'optimisation de type cache aware.

2.5.2.5 Caractérisation de l'ordre d'utilisation des données

Convaincus que presque 90% du temps de calcul d'un programme s'écoule dans les nids de boucles qui ne représentent que 10% du code, de nombreux chercheurs ont emprunté le chemin de l'optimisation de la localité dans ces boucles. Dans cette voie, P. Clauss et V. Leochner [37] ont proposé d'analyser les accès aux données à travers les différents niveaux de boucles. Ils se sont basés sur le calcul du polynôme d'Ehrhart pour déterminer la liste des données effectivement utilisées. Cette information leur a permis de définir des vecteurs qui décrivent l'ordre d'utilisation des données pour chaque niveau de boucle.

Les travaux de P. Clauss, dans ce cadre, ont été inspirés de ceux proposés par M. Kandemir et A. Choudhary [80]. Ces derniers ont contribué au développement

des approches basées sur l'optimisation de la localité spatiale des données pour les nids de boucles. Ils se sont concentrés sur l'optimisation de la boucle la plus interne tandis que P. Clauss a visé tous les niveaux de boucles dans l'implémentation de sa méthode.

Ces méthodes de caractérisation de l'ordre d'accès aux données sont efficaces pour l'optimisation de la localité spatiale. En revanche, elles sont difficiles à mettre en œuvre dans un contexte de parallélisme de boucles imbriquées de par la complexité de la caractérisation des accès mémoires. Cette complexité est justifiée par différentes considérations : les niveaux de boucles, la hiérarchie mémoire, la hiérarchie des unités de calcul (architectures multi-processeurs multi-coeurs) et le caractère dynamique de l'application. En effet, cette prédiction de l'ordre d'accès aux données à travers les lois de l'algèbre linéaire nécessite, impérativement, une connaissance a priori de l'ordre d'accès aux données ce qui n'est pas applicable pour les applications à caractère dynamique qui subissent une grande évolution au cours de leur exécution. De plus, la définition d'une loi statique qui décrit l'ordre d'accès aux données n'est pas compatible avec le caractère générique de l'application. Pour ces raisons, nous n'avons pas opté pour cette méthode dans notre approche d'optimisation de l'efficacité du cache pour le code EUROPLEXUS caractérisé par sa capacité de simuler des cas tests de nature très variés.

2.5.3 Optimisation de la localité du cache par transformation de boucles

Le point commun entre les méthodes d'optimisation de la localité que nous venons de présenter dans la section 2.5.2 est qu'elles se basent toutes sur la réorganisation de la structure de données sans effectuer des transformations majeures sur le corps du code. Il existe, néanmoins, un autre type d'approches basées sur la transformation des boucles et sur la réécriture des dépendances entre ces dernières afin d'optimiser la localité. Ces interventions sur le corps du programme sont d'autant plus difficiles à mettre en œuvre que la complexité des dépendances est importante.

Dans le cadre de cette thèse, nous ne nous intéressons pas à cette famille d'approche de par la complexité des applications que nous ciblons qui sont principalement des applications industrielles disposant de dépendances très complexes. Pour plus de détails sur ces approches, nous faisons référence aux travaux de Wolfe [135] et aux travaux de W. Li et K. Pingali [88] qui font partie des principales contributions dans le développement de ce type d'approches.

2.5.4 Parallélisme imbriqué efficace dans le cache

Pour faciliter le passage à l'échelle des applications parallèles, une des manières les plus naturelles proposées dans la littérature consiste à créer du parallélisme du *parallélisme imbriqué*. Cela se définit comme l'ouverture d'une région parallèle à l'intérieur d'une autre région parallèle. Cet emboîtement permet de "créer des threads par des threads" et de les faire participer à l'exécution de la région parallèle. Le niveau de parallélisme externe ainsi créé constitue un espace de travail local pour les threads fils.

De nombreux travaux ont approuvé l'utilisation de cette technique pour augmenter l'efficacité du parallélisme. Dans [123], Y. Tanaka et K. Taura ont souligné le gain réalisé par l'utilisation du parallélisme imbriqué sous OpenMP via une bibliothèque spécifique StackThreads/MP permettant de réduire les surcoûts du parallélisme.
Les nids de boucles sont la première application du parallélisme imbriqué. Dans ce contexte, dès ses premières versions, le standard OpenMP a implémenté le parallélisme imbriqué de différentes manières. Son activation est assurée soit par la variable d'environnement OMP_NESTED ou à l'intérieur du programme via l'appel à des fonctions spécifiques (omp_set_nested()).
Pour gérer le parallélisme imbriqué à différents niveaux de boucles, OpenMP implémente, également, une directive spécifique *COLLAPSE* [9]. Cette directive permet de déplier un certain nombre de boucle imbriquées pour constituer un large espace local des itérations. Cependant cette directive est uniquement applicable pour les boucles parfaitement imbriquées.
De plus, le parallélisme imbriqué, n'est pas encore intégré dans le compilateur pour la plupart des bibliothèques parallèles qui l'utilisent. Il risque, par conséquent, d'être ignoré au moment de l'exécution [**?**]. Certains compilateurs se contentent de sérialiser les boucles imbriquées.
Pour garantir que le parallélisme imbriqué soit pris en compte par le compilateur, de nombreux développeurs ont proposé des optimisations sur les compilateurs. Dans ce contexte, X. Tian et J. P. Hoeflinger [126] ont travaillé sur le compilateur Intel ICC pour implémenter un parallélisme imbriqué "sûr" du point de vue de l'ordonnancement des boucles imbriquées et de la gestion des synchronisations.
Les développeurs ne se sont pas arrêtés à ce stade dans l'exploitation de la puissance des architectures en environnement parallèle. Ils ont implémenté des stratégies pour gérer l'emplacement des threads de manière cohérente. Dans ce cadre, F. Broquedis et O. Aumage [26] ont proposé leur ordonnanceur FOREST_GOMP. Cet ordonnanceur se base sur l'arbre de dépendance généré par le compilateur. Ainsi, il contribue à une distribution optimale des données et des threads à travers les différents niveaux de boucles et entre les niveaux hiérarchiques de l'architecture.
Dans [114], A. Robison et M. Voss ont proposé une stratégie qui intègre le parallélisme imbriqué dans les blocs de tâches volées dans le cadre d'un ordonnancement par vol de travail. Ils ont implémenté et validé, expérimentalement, l'apport de cette stratégie dans la bibliothèque IntelTBB. Ils sont parvenus à optimiser la localité spatiale des données en favorisant la collaboration des threads à l'intérieur du domaine imbriqué.
Cependant, malgré la contribution du parallélisme imbriqué dans l'optimisation des performances dans certains cas, son efficacité n'a pas été reconnue par tous les développeurs. P. E. Hadjidoukas et V. V. Dimakopoulos [70] ont affirmé, en se basant sur un ensemble de microbenchmarks et sur des applications réelles, que "le parallélisme imbriqué ne garantit pas un gain en performance". Ce manque d'efficacité est justifié par la complexité de la gestion de ce type de parallélisme par les compilateurs [70].
La bibliothèque *libgomp*, présentée dans [102], est caractérisée par une supériorité par rapport aux résultats des travaux précédents par son intégration du parallélisme imbriqué dans le compilateur. Elle permet d'assurer la création des threads fils à l'intérieur des régions imbriquées et de gérer leur synchronisation. Libgomp prend

également en charge le partage du travail entre les threads. Ainsi, l'implémentation de cette bibliothèque a permis de réaliser un progrès en termes de performance.
Cependant, malgré ces efforts, l'implémentation de cette bibliothèque présente encore de nombreuses failles. Elle n'a pas pu surmonter la difficulté d'affecter efficacement les threads aux boucles internes sans risquer une dégradation de la localité mémoire. De plus, le rééquilibrage de la charge à travers les différents niveaux de boucles n'est pas prévu. Les vols sont limités aux threads du même niveau, ce qui oblige ceux du niveau interne à rester inactifs dans le cas où ils achèvent leur travail avant les threads d'un autre niveau. Ces évaluations sont basées sur les résultats expérimentaux présentés par V. V. Dimakopoulos et P. E. Hadjidoukas dans [45] uniquement pour deux niveaux de boucles. Les auteurs n'ont pas été optimistes pour une extension de leur étude sur un degré supérieur d'imbrication de boucles.
Le retour d'expérience des développeurs concernant l'efficacité du parallélisme imbriqué est partagé entre l'efficacité théorique de cette méthode et la difficulté de sa mise en place qui nécessite un effort particulier sur différents plans (compilateur, application, bibliothèque parallèle, ordonnanceur, localité de la mémoire, ...).

Dans le cadre de cette thèse, nous comptons enrichir cette expérience en se concentrant sur un axe que nous jugeons indispensable pour réduire les coûts engendrés par le nouveau niveau de boucle créé pour favoriser le travail de la boucle internes sur des données plus compactes. Notre idée motrice inspirée du parallélisme des boucles imbriquées consiste à mettre en place un mode de parallélisme qui autorise l'équilibrage dynamique de la charge à travers les différents niveaux de boucles tout en respectant les contraintes de localité des données dans la mémoire. Cette stratégie a été implémentée dans la bibliothèque parallèle XKAAPI développée à l'INRIA. Elle a été ensuite évaluée sur le code industriel EUROPLEXUS, objet de nos travaux d'optimisation, que nous présentons en détails au chapitre 4.

Chapitre 3

Un peu de mécanique

Deux siècles avant le début de l'ère chrétienne, en prenant son bain, Archimède comprit que tout corps plongé dans un fluide subit une poussée verticale, dirigée de bas en haut égale au poids du fluide déplacé. Cette découverte date les début d'une discipline entière et complexe qui étudie le comportement des fluides, c'est-à-dire les liquides et les gaz : "La mécanique des fluides".

Ce chapitre a pour but d'éclairer les fondements mécaniques qui servent de base au code EUROPLEXUS. Bien que l'accent soit mis sur la mécanique des fluides, les approches développées dans ce travail sont aussi applicables dans le cas de la mécanique des structures. Les systèmes composés par des fluides et/ou des structures sont les systèmes les plus complexes à traiter et nous pouvons considérer que les structures solides sont des cas particuliers des fluides.

Sommaire

3.1 Cinématique

La cinématique des fluides est la branche qui étudie les mouvements des particules d'un système mécanique [1]. En mécanique [77],[1],[119] il existe des formalismes distincts pour décrire ces mouvements : la description Lagrangienne [2] et la description Eulérienne [3].
Le choix entre ces deux descriptions est fortement lié à la nature du problème à traiter. Dans [64],[98], P. Germain décrit les formulations mathématiques de ces deux concepts. Dans cette section nous n'allons pas nous attarder sur l'aspect mathématique mais nous allons nous contenter d'une description littéraire suffisante pour comprendre les notions générales.

3.1.1 Description Lagrangienne

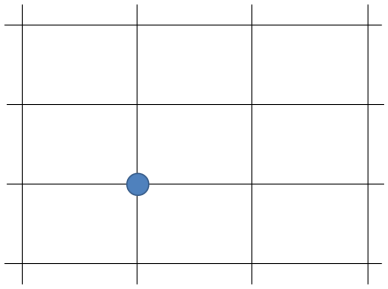


FIGURE 3.1 – Configuration initiale

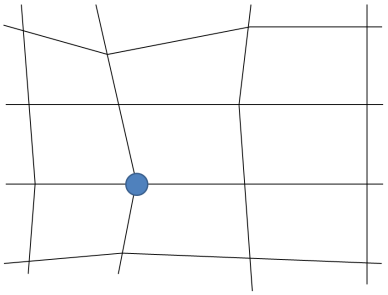


FIGURE 3.2 – Mise à jour du maillage en formulation Lagrangienne

Dans cette description, on suppose connaître la trajectoire de la particule et on ne s'intéresse qu'au temps t et aux coordonnées initiales.
Lorsqu'il est question de traiter des simulations présentant de grandes déformations [4][54],[2],[33], le problème à résoudre est fortement non linéaire. Il est donc primordial d'adopter une approche incrémentale pour parvenir à sa résolution. On calcule des itérations successives en partant à chaque fois d'un ensemble d'informations disponibles sur une configuration initiale connue. Le choix de la configuration de référence nous mène à distinguer deux classes de formulation Lagrangienne ; la formulation Lagrangienne totale et la formulation Lagrangienne réactualisée. Toutes les deux sont capables de traiter des problèmes de grandes déformations.
Hibbit a introduit dans [73] la formulation Lagrangienne totale. Cette formulation se base sur la configuration initiale comme configuration de référence. Dans cette méthode, la configuration initiale reste inchangée durant toute la simulation. Cependant, dans une formulation Lagrangienne réactualisée [18], la configuration de

1. Nous désignons par *particule d'un fluide* un petit volume qui contient, en échelle microscopique, un certain nombre de molécules du milieu considéré.
2. Joseph-Louis Lagrange, 1736 - 1813.
3. Leonhard Euler, 1707 - 1783.
4. En grandes déformations, les variations subies par les particules du système au cours du temps ne peuvent pas être négligées.

référence est recalculée à la fin de chaque pas de temps. La configuration de référence qui sera retenue est donc celle qui est la plus récente.
Plusieurs auteurs se sont intéressés à l'étude de ces deux formalismes et sont parvenus à démontrer qu'ils sont mathématiquement équivalents [18],[85].

3.1.2 Description Eulérienne

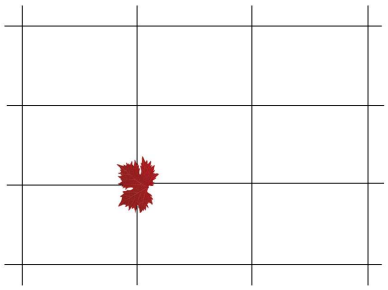


FIGURE 3.3 – Configuration initiale

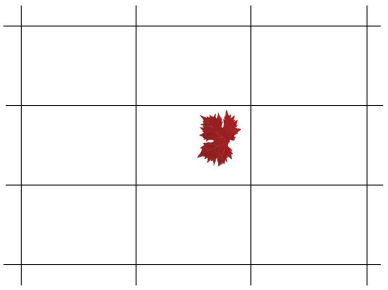


FIGURE 3.4 – Mise à jour du maillage en formulation Eulérienne

A la différence du formalisme Lagrangien, en description Eulérienne, au lieu de décrire la vitesse d'une particule, on s'intéresse plutôt à l'étude du mouvement des particules à des points fixes de l'espace. Dans ce cas, l'observateur n'est pas lié aux particules du fluide. En guise d'illustration, un promeneur se place sur un pont et regarde une feuille emportée par le courant dans la rivière.
Ainsi, en suivant l'évolution temporelle de la vitesse en un point fixe, on aura à chaque instant t, une particule différente qui se situe en ce même point.
Le maillage est supposé fixe et les particules en mouvement le traverse. Si on se place dans le cadre de la méthode des éléments finis ou des volumes finis, le maillage est considéré comme un volume de contrôle fixe dans l'espace. Chaque particule du fluide peut se déplacer d'un élément fini vers un autre. Cependant, cette liberté de migrer à travers les éléments pourrait engendrer un dépassement du domaine initialement défini. Dans ce cas, on risque de perdre les informations relatives aux particules qui ont quitté le domaine. D'où la nécessité d'être bien attentionné en définissant *les conditions aux limites* pour ne pas dépasser les frontières matérielles du domaine à traiter. Malgré la difficulté de définir ces conditions aux limites, la méthode Eulérienne est parfaitement adaptée à l'étude des problèmes présentant d'importantes distorsions. Cette caractéristique justifie la large utilisation de cette formulation dans la mécanique des fluides.

3.1.3 Formalisme général : Description ALE

La description Arbitrairement Lagrangienne Eulérienne (ALE) (Arbitrary Lagrangian–Eulerian, en anglais) représente le cas le plus général des deux formulations précédentes [107]. Elle a été initialement conçue en différences finies, ensuite pour les éléments finis vers la fin des années 70 [47].

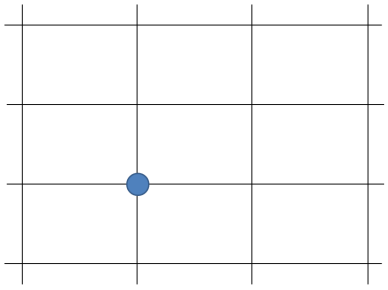


FIGURE 3.5 – Configuration initiale

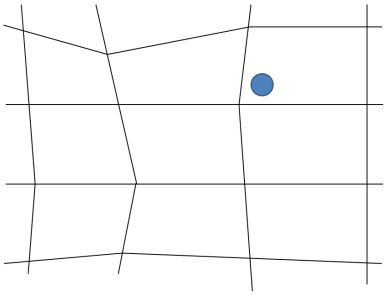


FIGURE 3.6 – Mise à jour du maillage en formulation ALE

Remarque 2. Wikipédia : *La méthode des différences finies est une technique courante de recherche de solutions approchées d'équations aux dérivées partielles qui consiste à résoudre un système de relations (schéma numérique) liant les valeurs des fonctions inconnues en certains points suffisamment proches les uns des autres.*

Cette méthode suppose que le système étudié n'est pas préalablement lié à la grille et que la grille est située dans une zone mobile de l'espace.
De même que pour la description Eulérienne, on s'intéresse à la vitesse des particules traversant une grille, mais dans le cas ALE, la grille traversée est à son tour mobile. La méthode ALE repose sur l'étude du mouvements des particules qui traversent une grille placé dans un repère mobile. Elle est valable aussi bien pour l'étude des mouvements des fluides que ceux des structures. La description ALE est particulièrement utilisée dans l'étude des problèmes couplés. Elle nous permet de tirer profit des points forts de la méthode Lagrangienne caractérisée par sa facilité d'imposer les conditions aux limites et des points forts de la méthode Eulérienne connue pour sa parfaite adaptabilité aux problèmes à grandes déformations.

3.2 Discrétisation en espace

S'aventurer à résoudre analytiquement des problèmes de la mécaniques des milieux continus est une mission (presque) impossible. Conscients de la complexité de la tâche, des ingénieurs ont opté pour des méthodes numériques pour résoudre ces problèmes. On trouve dans la littérature plusieurs techniques qui permettent de résoudre les équations aux dérivées partielles décrivant un problème mécanique donné. A titre d'exemple nous citons les méthodes des différences finies, les méthodes des volumes finis, les méthodes spectrales, les méthodes des éléments finis, etc. Dans [61], A.Fortin et A.Garon affirment que la méthode des éléments finis est "sans doute" la plus répandue. Cette méthode se distingue par sa robustesse pour des problèmes de mécanique des milieux continus et sa rigueur mathématique fortement consolidée par les travaux de Strang et Fix[120] et soulignée par A.Ern dans [6].
Le domaine d'application de cette méthode n'a pas cessé de s'élargir et plusieurs travaux de recherche sur ce sujet se sont largement développés [6][30][40].

3.2.1 Méthode des éléments finis

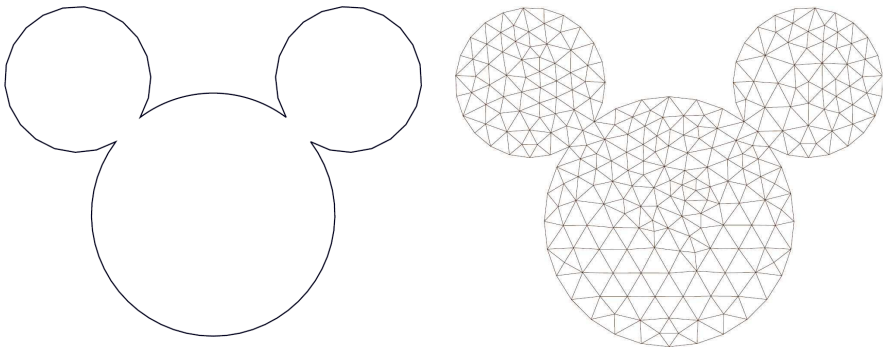


FIGURE 3.7 – Discrétisation d'un système réél (a) en éléments géométriques (b) sous la plateforme Salome

L'idée fondamentale de cette méthode est de découper géométriquement un système physique continu en un nombre fini d'éléments géométriques simples [61][65][54] qu'on appelle "*mailles*". La méthode des éléments finis consiste à définir une solution approchée pour un ensemble d'équations correspondant à un domaine spatial. Cette solution doit également respecter les conditions imposées.
L'étape initiale dans l'approche des éléments finis est la discrétisation du système continu en mailles de petites tailles. L'opération de maillage sera évoquée plus tard dans ce chapitre.
Les mailles ainsi obtenues sont connectées les unes par rapport aux autres en un nombre fini de points situés sur les arêtes des éléments. Un point d'intersection entre deux mailles est un nœud. C'est au niveau de ces nœuds que les inconnues du système linéaire sont évaluées.
On s'intéresse, ensuite, à la définition d'une formulation simplifiée pour chaque maille. En d'autres termes, on assure le passage d'un système d'équations complexes sur le système global, difficile à résoudre, vers un système d'équations linéaires en général plus facile à résoudre. Chaque système peut se représenter sous forme d'une matrice.
A partir de ces système matriciels, on forme une grande matrice qui rassemble tous les éléments finis pour reconstruire le système entier. Dans le système global, on veille à ce que l'équilibre local en chaque nœud soit respecté. Ainsi, la résolution du nouveau système recomposé forme la solution approchée du problème.

3.3 Discrétisation en temps

Étudier le mouvement continu d'un système physique est une opération très compliquée voire même impossible. La discrétisation spatiale d'un système est nécessaire pour simplifier l'étude et la résolution du problème comme nous venons de voir dans la section précédente. Dans cette partie nous allons nous intéresser à la discrétisation

dans le temps du mouvement d'un système dynamique.

La résolution d'un problème de dynamique non linéaire, en mécanique des fluides, nécessite le calcul de certaines valeurs nodales à partir d'un état connu du système en question. Il faut notamment calculer au niveau des nœuds le déplacement, la vitesse et l'accélération à un instant t_{n+1}. Ces grandeurs doivent vérifier les équations du mouvement :

$$u_n \tag{3.1}$$

$$\dot{u}_n \tag{3.2}$$

$$\ddot{u}_n \tag{3.3}$$

tels que :

$$u_n = u(t_n) \tag{3.4}$$

$$\dot{u}_n = \dot{u}(t_n) \tag{3.5}$$

$$\ddot{u}_n = \ddot{u}(t_n) \tag{3.6}$$

À partir de ces équations à l'instant t_n, l'enjeu consiste à trouver :

$$u_{n+1} \tag{3.7}$$

$$\dot{u}_{n+1} \tag{3.8}$$

$$\ddot{u}_{n+1} \tag{3.9}$$

ces équations doivent valider l'équation de l'équilibre dynamique :

$$M\ddot{u}_{n+1} + F^{int}_{n+1} = F^{ext}_{n+1} \tag{3.10}$$

Cette opération possède le mérite de rendre possible et réaliste toute étude dynamique. Elle consiste à décomposer l'intervalle continu du temps en un ensemble de sous-intervalles.
L'évaluation des inconnues du système à résoudre est effectuée en un nombre finis d'instants appartenant à l'intervalle total d'étude.
On appelle la durée qui sépare deux instants successifs d'évaluation *'pas de temps'*.
La résolution numérique de problèmes en dynamique non linéaire repose sur l'utilisation des *schémas d'intégration temporelle* dans ces instants. Le schéma le plus répandu dans le domaine de discrétisation temporelle est celui proposé par Newmark [100] en 1959. Il repose sur une famille de schémas basée sur l'approximation des déplacements u_{n+1} et des vitesses $\dot{u}_{n+1}$ à un instant t_{n+1} dans le but de calculer la valeur de l'accélération $\ddot{u}_{n+1}$ à cet instant.
L'utilisation de ces relations permet de réduire les inconnues du système aux accélération à l'instant t_{n+1} et par conséquent, avoir un système plus facile à résoudre [92].
Plusieurs références dans la littérature détaillent les différents schémas d'intégration de Newmark. Chaque schéma de Newmark correspond à des valeurs particulières des deux paramètres γ et β issus du développement en série de Taylor de la formulation générale de l'équation de la dynamique :

$$M\ddot{u}_t + C\dot{u}_t + Ku_t = f_t \tag{3.11}$$

Avec :

- **M** est la matrice de masse ;
- **C** est l'opérateur d'amortissement ;
- **K** est la matrice de rigidité.

Les opérateurs figurant dans l'équation de la dynamique découlent des équations de la discrétisation par la méthode des éléments finis. Ce qui conduit après développement en série de Taylor à :

$$u_{t+\Delta t} = u_t + \Delta t \dot{u}_t + \frac{\Delta_t^2}{2}\ddot{u}_t + \beta \Delta_t^3 \dddot{u} \tag{3.12}$$

et

$$\dot{u}_{t+\Delta t} = \dot{u}_t + \Delta t \ddot{u}_t + \gamma \Delta_t^2 \dddot{u} \tag{3.13}$$

Si on s'appuie sur l'hypothèse de linéarité de l'accélération $\ddot{u}$ dans un pas de temps Δt :

$$\dddot{u} = \frac{\ddot{u}_{t+\Delta t} - \ddot{u}_t}{\Delta t} \tag{3.14}$$

Ce qui implique que :

$$u_{t+\Delta t} = u_t + \Delta t \dot{u}_t + \frac{\Delta_t^2}{2}\ddot{u}_t + \beta \Delta_t^3 \dddot{u} \tag{3.15}$$

$$= u_t + \Delta t \dot{u}_t + \frac{\Delta_t^2}{2}\ddot{u}_t + \beta \Delta_t^3 \frac{\ddot{u}_{t+\Delta t} - \ddot{u}_t}{\Delta t} \tag{3.16}$$

$$= u_t + \Delta t \dot{u}_t + \frac{\Delta_t^2}{2}\ddot{u}_t + \beta \Delta_t^2 \ddot{u}_{t+\Delta t} - \beta \Delta_t^2 \ddot{u}_t \tag{3.17}$$

$$= u_t + \Delta t \dot{u}_t + \frac{\Delta_t^2}{2}(\ddot{u}_t(1-2\beta) + 2\beta \ddot{u}_{t+\Delta t}) \tag{3.18}$$

et que :

$$\dot{u}_{t+\Delta t} = \dot{u}_t + \Delta t \ddot{u}_t + \gamma \Delta_t^2 \dddot{u} \tag{3.19}$$

$$= \dot{u}_t + \Delta t \ddot{u}_t + \gamma \Delta_t^2 \frac{\ddot{u}_{t+\Delta t} - \ddot{u}_t}{\Delta t} \tag{3.20}$$

$$= \dot{u}_t + \Delta t \ddot{u}_t + \gamma \Delta_t \ddot{u}_{t+\Delta t} - \gamma \Delta_t \ddot{u}_t \tag{3.21}$$

$$= \dot{u}_t + (1-\gamma)\Delta t \ddot{u}_t + \gamma \Delta_t \ddot{u}_{t+\Delta t} \tag{3.22}$$

Ainsi, on obtient le système suivant :

$$\begin{cases} \dot{u}_{t+\Delta t} = \dot{u}_t + (1-\gamma)\Delta t \ddot{u}_t + \gamma \Delta_t \ddot{u}_{t+\Delta t} \\ \\ u_{t+\Delta t} = u_t + \Delta t \dot{u}_t + \frac{\Delta_t^2}{2}(\ddot{u}_t(1-2\beta) + 2\beta \ddot{u}_{t+\Delta t}) \end{cases}$$

Remarque 3. *Dans le cas où*

$$\gamma = \frac{1}{2} \tag{3.23}$$

et

$$\beta = 0 \tag{3.24}$$

le schéma d'intégration de Newmark est dit schéma explicite des différences finies centrées. *Pour*

$$\gamma = \frac{1}{2} \tag{3.25}$$

et

$$\beta = \frac{1}{4} \tag{3.26}$$

le schéma d'intégration obtenu fait partie des schémas de Newmark implicites. Il est dit schéma de l'accélération moyenne.

Le schéma d'intégration **explicite** [92][54] repose sur une dépendance explicite entre les déplacements au pas de temps t_{n+1}, u_{n+1}, et les variables au pas de temps précédent t_n, tandis que pour les méthodes d'intégration dites **implicites**, les déplacements au pas de temps t_{n+1} dépendent des vitesses et des accélérations à ce même pas de temps.

3.3.1 Schéma explicite

Le schéma explicite est particulièrement utilisé dans la simulation des événements rapides comme les explosions, les chocs et les crashs. Cette méthode est économe en mémoire et elle est, par conséquent, bien adaptée aux problèmes de grandes tailles. Lorsqu'il est associé à une discrétisation spatiale par la méthode des éléments finis, le schéma explicite s'accompagne courrament d'une matrice de masse condensée pour produire une matrice diagonale ce qui donne à cet algorithme une grande facilité de programmation. Cette famille de schémas est *conditionnellement stable*, ce qui impose l'utilisation de petits pas de temps et lui confère précision et pertinence pour la représentation de phénomène propagatifs rapides. Le pas de temps est périodiquement mis à jour dans le but de vérifier toujours au plus juste la condition de stabilité.

3.3.2 Schéma implicite

Dans la méthode d'intégration implicite, la matrice du système à inverser est une combinaision de la matrice de masse et de la matrice de rigidité du système. Elle n'est pas diagonale, ce qui alourdit considérablement le calcul numérique, en particulier pour des phénomènes non-linéaires. On se retrouve donc avec des programmes coûteux en mémoire et en temps de calcul, bénéficiant en retour de contraintes moins lourdes sur le pas de temps, grâce même à une stabilité inconditionnelle de l'intégration temporelle dans certaines conditions.

Pour ces raisons, les schémas implicites sont généralement utilisés en statique ou en dynamique lente.

Bilan bibliographique

Dans la présente partie, nous avons rappelé les notions de base autour des architectures parallèles (chapitre 1). Nous avons particulièrement mis l'accent sur les machines à mémoire partagée auxquelles les développements de cette thèse sont destinés.
La hiérarchie mémoire de ces architectures, notamment, les mémoires cache devient de plus en plus complexe. Afin de répondre aux besoins des applications industrielles, les développeurs et les constructeurs conçoivent, régulièrement, des techniques permettant de profiter au mieux de ces architectures.
Ensuite (chapitre 2), nous avons vu que l'accélération obtenue grâce à ces mémoires a vite atteint ses limites dans le cadre mono-processeur tandis que les applications industrielles n'ont pas cessé de se complexifier avec le progrès dans tous les domaines de la science. Pour pallier à ça, les architectes ont généralisé une nouvelle catégorie d'architectures mettant plusieurs unités de calcul en parallèle au service de ces applications.
De nombreux modèles de programmation parallèle ont été développés chacun étant dédié à un type d'architecture particulier. Pour assurer le passage à l'échelle, les développeurs ont mis en œuvre des stratégies d'ordonnancement spécifiques qui permettent à l'environnement parallèle de s'adapter aux spécificités architecturales de la plateforme de calcul ainsi qu'aux besoins de l'application.
Pour répondre à la nature dynamique des applications de simulations, des stratégies d'équilibrage dynamique de charge par vol de travail ont été implémentées dans les ordonnanceurs. Nous avons étudié ces méthodes que nous allons implémenté par la suite dans notre application pour augmenter l'efficacité du parallélisme.
Suite à notre étude bibliographique, nous avons eu conscience de la nécessité de la mise en place d'un parallélisme qui soit efficace dans le cache pour réaliser un meilleur gain en performance. Ainsi, nous avons consacré le chapitre 2 de cette partie à parcourir les principaux travaux d'optimisation de l'utilisation des caches. Nous avons pu distinguer deux catégories d'approches : les approches d'optimisation par ré-ordonnancement de boucles et les approches basées sur la réorganisation de la structure de données. Ainsi, guidés par les caractéristiques des applications de simulation que nous visons, nous avons écarté l'implémentation de la première approche de par la complexité des dépendances que comporte notre application. Nous avons focalisé notre étude sur la deuxième catégorie d'approches et nous nous sommes inspirés de la plus adaptée entre elles pour mettre en place notre approche d'optimisation de l'utilisation du cache pour le code EUROPLEXUS.
Conscients que l'optimisation d'une application nécessite un minimum de connaissances de la branche scientifique qu'elle traite, nous avons consacrer le chapitre 3 de

cette partie à cette fin. Nous y avons rappelé les bases de la mécanique en rapport avec les notions que nous manipulerons lors de notre étude du code EUROPLEXUS.

Troisième partie

Contributions

Chapitre 4

Etat des lieux

Dans ce chapitre, nous effectuons l'état des lieux de l'environnement technique de la réalisation des travaux de ce projet. Nous décrivons dans la première section l'organisation hiérarchique de la plateforme de calcul utilisée. Ensuite, nous présentons le code EUROPLEXUS dans lequel nous avons implémenté ces travaux.

Nous commençons par une présentation générale de ce code. Puis, nous exposons quelques applications réelles parmi le large panel des modèles qu'il permet de simuler. Nous choisissons le jeu de données sur lequel nous nous appuyons pour évaluer les performances de notre approche.

Dans la deuxième partie de ce chapitre, après la présentation de la structure générale d'EUROPLEXUS, nous nous attardons sur l'étude de sa structure de données pour définir les points d'appui de notre approche d'optimisation.

Sommaire

4.1 Contexte architectural

Toutes les expérimentations et les calculs évoqués dans ce travail sont effectués sur un des nœuds Sandy Bridge hébergés au laboratoire DYN du CEA. Ce nœud nommé Pollux-2 possède des caractéristiques identiques à certains nœuds du super-calculateur Curie [1].
Il dispose de 4 processeurs (sockets) Intel Xeon E5-4620 2,2 Ghz.
La mémoire RAM est partagée entre les 4 processeurs. Elle est d'une capacité de stockage totale de 128 GB répartis équitablement sur les 4 bancs NUMA des 4 sockets (1 banc Numa de 32 GB par socket).
Les sockets du nœud communiquent entre eux à travers un bus haut débit InfiniBand. Nous reportons dans la figure 4.1 l'architecture d'un processeur du nœud Pollux-2.

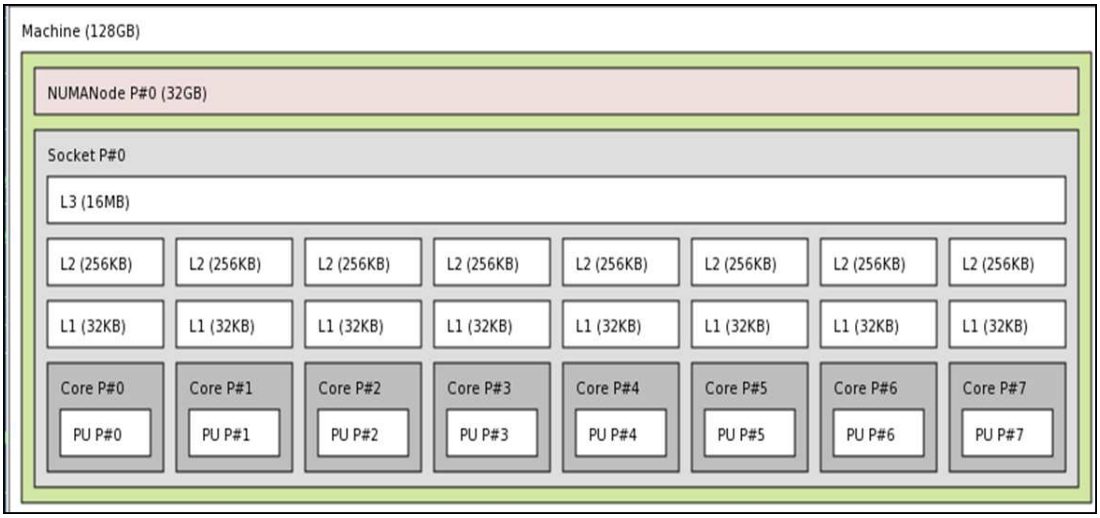


FIGURE 4.1 – Architecture d'un processeur du nœud Pollux-2

Chaque processeur dispose d'une mémoire cache à trois niveaux. Les deux premiers niveaux sont privés par cœur. Le dernier niveau est partagé entre les 8 cœurs du socket.
Nous décrivons dans la table 4.1 les caractéristiques des différents niveaux du cache d'un processeur du noeud Pollux-2.

4.2 EUROPLEXUS

Afin de démontrer la pertinence des approches proposées dans ce travail doctoral, nous les avons implémentées dans le logiciel de simulation en dynamique rapide des fluides et des structures en interaction EUROPLEXUS (abrégé EPX dans la suite du document). Il est la copropriété du CEA et de la Commission Européenne via le Joint Research Center (centre d'Ispra, Italie). Son développement est conduit dans

1. http ://www-hpc.cea.fr/en/complexe/tgcc-curie.htm

TABLE 4.1 – Caractéristiques des niveaux du cache d'un processeur de Pollux-2

Niveau	L1	L2	L3
Taille	32 KB	256 KB	16MB
Type	données	données + instructions	données + instructions
Associativité	8	8	16
Taille d'une ligne	64 o	64 o	64 o
Organisation externe	inclusif	inclusif	non-inclusif
Degré du partage	privé	privé	8 cœurs

le cadre d'un consortium impliquant en plus des copropriétaires des partenaires dits *majeurs*, EDF et ONERA, disposant d'un accès complet au code source.
EPX est construit sur un algorithme explicite adapté aux transitoires brutaux fortement non-linéaires. Son processus de résolution des équations de la dynamique rapide est nettement marqué par l'hétérogénéité des modèles mis en œuvre et l'évolution des caractéristiques des systèmes considérés au cours du temps. Ceci en fait un outil pertinent de test à l'échelle industrielle des concepts introduits dans les chapitres précédents. Il forme ainsi un support d'implémentation pour des structures de données efficaces en présence de nombreux accès dans les différents niveaux de la mémoire et pour un parallélisme adaptatif à même de préserver ses performances face à l'évolution du coût relatif des tâches à effectuer.

4.3 Organisation du code

4.3.1 Algorithme général

Dans son organisation interne, EPX obéit à la structure générale d'un code de simulation en dynamique rapide. Ainsi, il comporte un ensemble de routines FORTRAN inter-connectées suivant une architecture algorithmique spécifique. Cette organisation est basée sur des "switchers" permettant de se positionner dans une zone bien définie du code selon la nature des éléments traités et selon d'autres options de configuration du code. Une partie de ces décisions est prise durant la phase d'initialisation. Les autres décisions sont prises en compte lors du lancement du calcul. Cette souplesse dans le choix des options permet l'adaptabilité du code à une grande variété de modèles. Nous résumons dans la figure 4.2 la structure générale du code EPX.

Les données nécessaires à la simulation sont fournies par l'intermédiaire d'un fichier de maillage pour le support de la discrétisation spatiale et d'un fichier de données pour les paramètres de calcul (nature et caractéristiques des matériaux, formulation eulérienne, lagrangienne ou ALE, options pour l'intégration temporelle...). Le maillage est construit par une application dédiée (voir par exemple la section 2.5.2.2). Conformément à sa fonction de simulation de phénomènes transitoires,

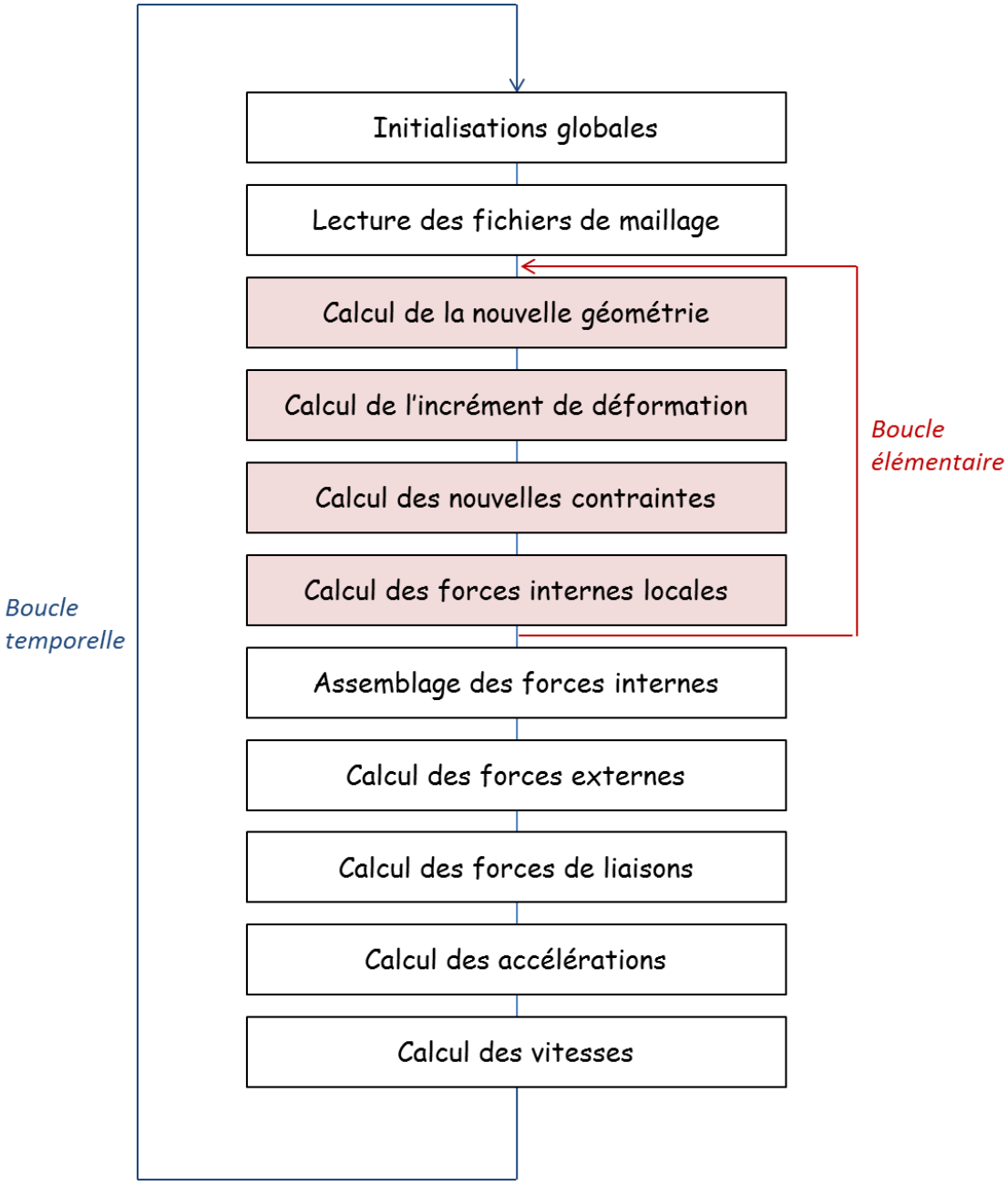


FIGURE 4.2 – Schéma simplifié de la structure générale du code EUROPLEXUS

EPX comporte une boucle principale sur les pas de temps. À l'intérieur de cette boucle, se déroulent les calculs des différentes grandeurs physiques ainsi que les calculs des interactions entre les éléments du maillage. Dès le début d'un pas de temps n, les positions et les déplacements de tous les nœuds des éléments sont connus.Le calcul des forces internes à chaque élément, son incrément de déformation ainsi que ses nouvelles contraintes à partir de la loi du comportement du matériau utilisé se déroulent au sein d'une boucle particlière. Cette boucle qui parcourt tous les éléments du maillage dans chaque pas temps sera appelée, dans la suite du document, la *boucle élémentaire*. Lorsque les calculs élémentaires arrivent à terme, une étape d'assemblage des forces aux nœuds est nécessaire pour déduire les nouvelles accélérations et les nouvelles vitesses. Ces grandeurs seront utilisées dans le calcul de la nouvelle géométrie au pas de temps $n + 1$. Par la donnée de la nouvelle position et du nouveau déplacement, le nouveau pas de temps se déclenche dans le cadre d'une nouvelle itération de la boucle temporelle.

4.3.2 Périmètre applicatif et caractéristiques algorithmiques

EPX est dédié à la simulation de la réponse dynamique de systèmes fluide-structure complexes à des chargements extrêmes, tels que des chocs et des explosions (voir par exemple les figures 4.3 et 4.4).

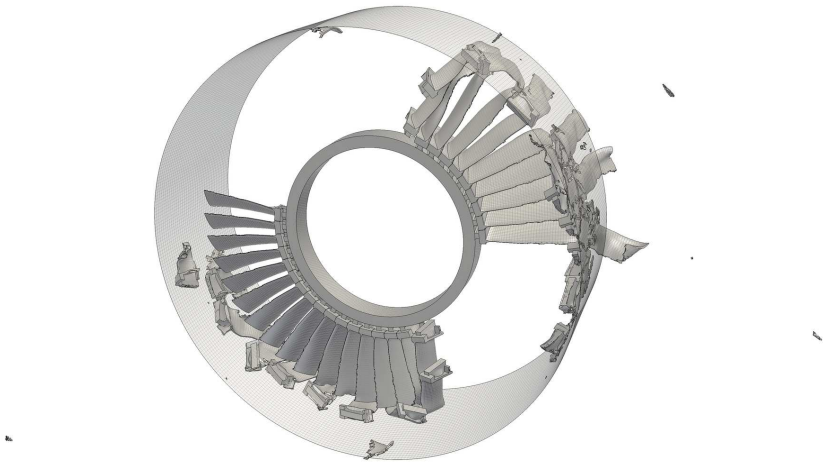


FIGURE 4.3 – Crash d'un rotor avec contact rotor/stator : endommagement majeur des structures, contact généralisé

Les tâches algorithmiques majeures consistent donc, conformément à la figure 4.2, d'une part au calcul des forces internes dans les structures et les fluides, avec également les flux de quantités eulériennes à travers les faces des cellules pour ces derniers, dans la boucle élémentaire, et d'autre part à l'identification des interactions cinématiques entre structures et entre fluide et structure, puis à l'évalution des forces de liaisons associées. On obtient deux familles d'algorithmes aux caractéristiques distinctes : la boucle élémentaire est un enchaînement d'actions indépendantes présentant une forte hétérogénéité dans le coût associé à chaque cellule de la discrétisation spatiale, alors que le traitement des liaisons cinématiques comprend des

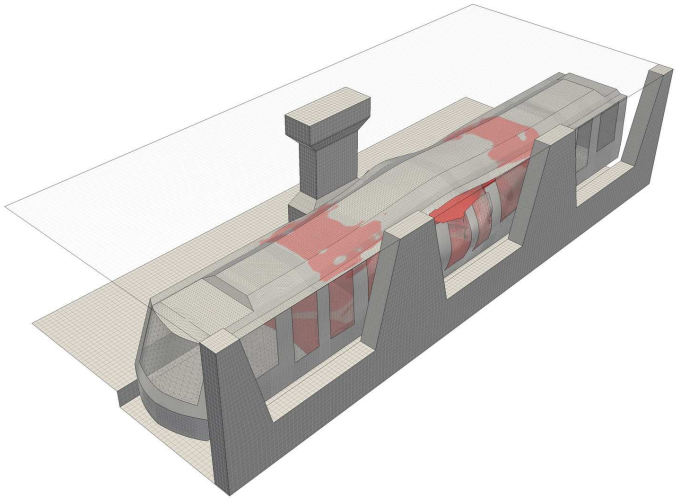


FIGURE 4.4 – Explosion dans une infrastructure ferroviaire : interaction fluide-structure avec rupture et grands déplacements des structures

opérations géométriques pour la phase d'identification et la résolution d'un système linéaire spécifique pour la phase de calcul des efforts d'interaction.
Une stratégie de résolution efficace doit prendre en compte les caractéristiques des deux familles ci-dessus, mais pour séparer les difficultés et permettre une étude approfondie des questions d'équilibrage dynamique de la charge et d'efficacité du placement des données, impactant particulièrement la première famille, on se place volontairement dans une situation de calcul sans liaison cinématique additionnelle (voir en particulier la section 4.5 décrivant le cas servant de support aux analyses dans la suite de ce document).

4.3.3 Profiling

Cette étude préliminaire a pour objectif de cerner les parties du code qui prennent le plus de temps pour s'exécuter, dans le contexte applicatif spéficique à cette thèse présenté au paragraphe précédent.

Le graphique de la figure 4.5 obtenu avec *gprof* indique une répartition grossière du temps d'exécution sur les différentes parties du code. À l'évidence, presque les trois quarts du temps d'exécution de l'application sont écoulés dans le calcul de la boucle élémentaire. Le quart restant est partagé entre les initialisations globales de l'application, l'addition des variables nodales de chaque éléments à l'issu de la boucle élémentaire ainsi qu'à l'évaluation des forces et au calcul des nouvelles vitesses et accélérations.

Cette boucle élémentaire constitue bien le "cœur" de notre application pour le travail de cette thèse.

À l'intérieur de la boucle élémentaire, toutes les grandeurs relatives aux éléments sont évaluées. Le calcul de ces grandeurs nécessite des consultations d'une liste de tableaux de données au fur et à mesure de l'exécution des itérations sur les éléments.

Ces constatations nous ont menées vers une étude plus approfondie de l'organisation de la structure de données du code que nous considérons comme une piste

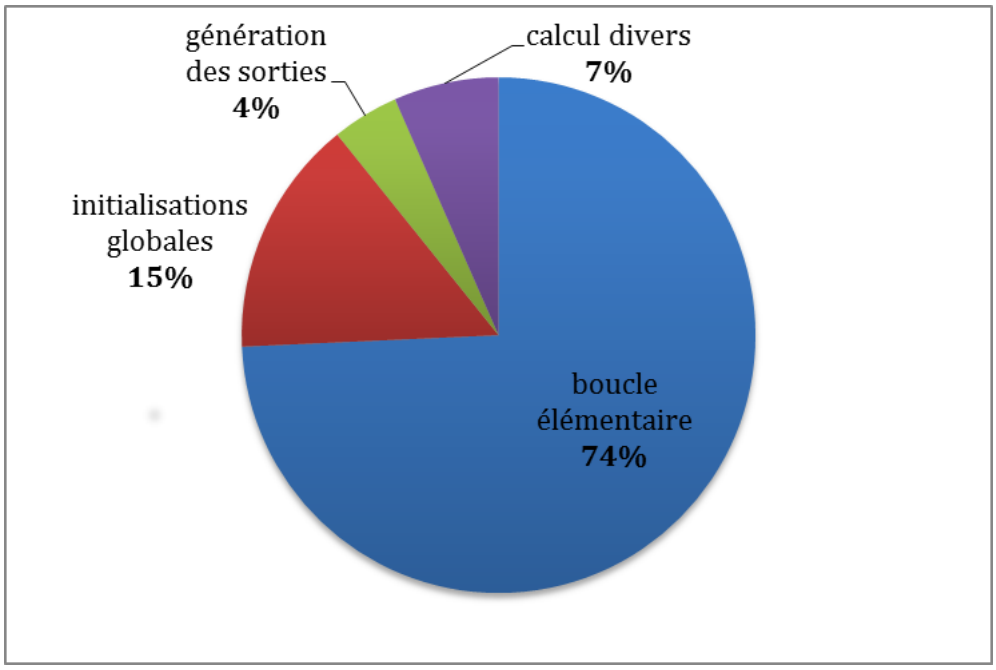


FIGURE 4.5 – Profiling du code EPX

prometteuse à optimiser pour accélérer l'exécution.

4.3.4 Structure de données

L'organisation de la structure de données du code EPX nous offre un modèle assez représentatif des structures de données de nombreux codes de mécanique, que nous nous proposons à présent de détailler pour en identifier les caractéristiques principales en termes de placement des données.

Avant de nous lancer dans la présentation détaillée du modèle de données, nous tenons à rappeler le vocabulaire que nous allons utiliser dans la suite du document lorsqu'on traite les éléments d'un maillage :

Notons que les termes "*maille*", "*élément*" et "*cellule*" sont équivalents.

Dans notre cas d'étude, un maillage est caractérisé par des éléments et des nœuds partagés (ou pas) entre ces éléments (cf. figure 4.6).
Lorsqu'une face (arête) appartient à deux éléments du maillage, nous considérons que ces deux éléments sont des **voisins**.

La structure de données du code EPX est présentée sous la forme d'une liste de tableaux. Chaque tableau est dédié au stockage d'une grandeur physique particulière.
Nous disposons ainsi de :

- tableaux de coordonnées nodales,
- tableaux de déplacements,
- tableaux de vitesses aux nœuds,
- tableaux pour les accélérations nodales,
- tableaux pour les forces internes aux nœuds,

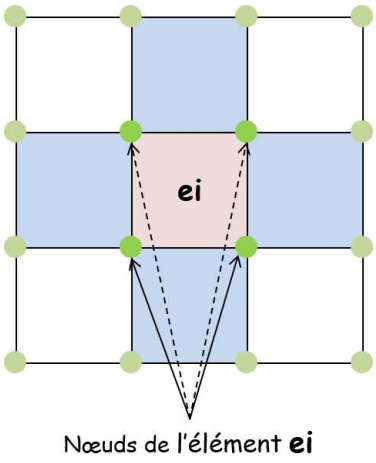


FIGURE 4.6 – Représentation d'un élément et ses nœuds dans maillage 2D

- tableaux pour les déformations élémentaires,
- tableaux pour les contraintes,
- etc.

Focalisons notre étude, par exemple, sur l'élément e_i représenté avec un cadre **noir gras** sur la grille représentée dans la figure 4.7.

Comme nous venons de le voir précédemment, l'ensemble des calculs relatifs à chaque élément se déroule à l'intérieur des itérations de la boucle élémentaire. Lors de l'exécution de cette boucle, les données nécessaires au déroulement des tâches élémentaires sont chargées au fur et à mesure à partir des tableaux constituant la structure de données.

Ainsi, le calcul relatif à l'i^{ème} élément engendre une consultation de plusieurs tableaux afin de charger les données demandées par le processeur. Ce chargement ne peut, malheureusement, pas s'effectuer uniquement à l'échelle de la case du tableau qui contient la donnée en question. En effet, d'après la politique de fonctionnement des mémoires, pour emmener la donnée en question dans son cache, le processeur procède au chargement de toute la ligne (ou l'ensemble de lignes) qui contient la case recherchée. De ce fait, plusieurs données seront inutilement stockées dans le cache de par leur appartenance au bloc mémoire contenant la donnée demandée. Ce mode de chargement engendre une hausse du nombre de défauts de cache et ralentit, par conséquent, l'exécution de la boucle élémentaire par rapport à une implémentation optimale utilisant efficacement l'espace de stockage. Ces constatations nous ont guidés vers la détermination d'une voie d'optimisation prometteuse pour l'accélération de la boucle élémentaire du code EPX (cf. chapitre 5).

4.4 Stratégie parallèle dans EUROPLEXUS

L'approche parallèle dans EPX a été conçue pour les machines de la génération *petaflopique*, à savoir des agrégats (*clusters*) de nœuds multi-cœurs à mémoire hiérarchisée et partagée. Les accélérateurs, de type GPU en particulier, n'ont pas été

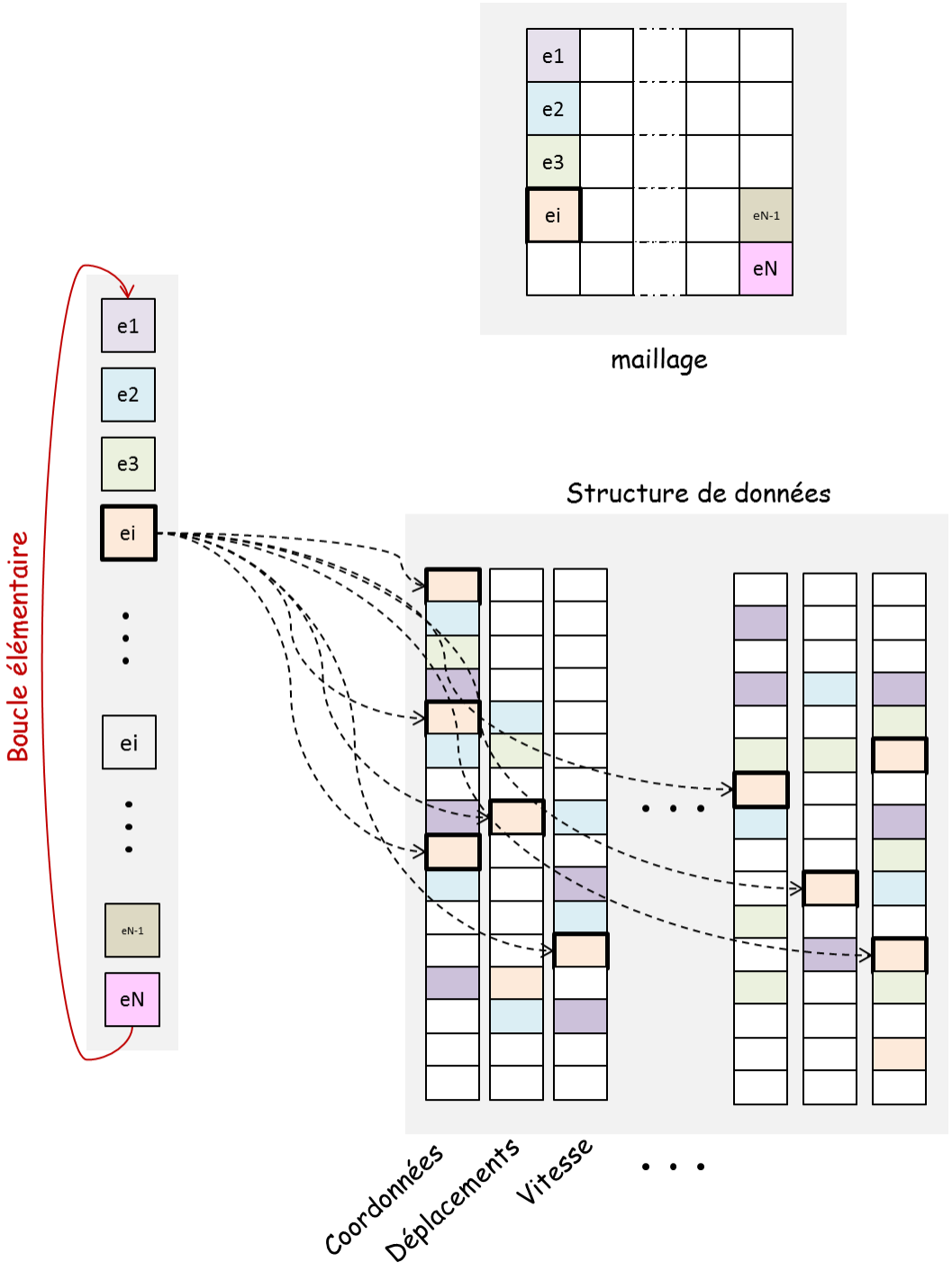


FIGURE 4.7 – Organisation de la structure de données du code EUROPLEXUS

pris en compte dans cette phase de conception, compte tenu de l'effort de développement spécifique hors de portée pour une application de la complexité d'EPX.
Les étapes principales du passage à l'échelle de l'application ont été franchies par l'intermédiaire du projet ANR RePDyn [2] [4] dont les partenaires sont CEA, EDF, LaMSID, ONERA, LaMCoS et INRIA.

4.4.1 Parallélisme à mémoire distribuée et mémoire partagée dans EPX

Le parallélisme dominant dans le code EPX est fondé sur une décomposition de domaine pour gérer la distribution des données entre les nœuds et les échanges explicites d'informations via la librairie MPI. Il est imposé par la nécessité de gérer en priorité la distribution de la mémoire sur les clusters de grande taille, prérequis pour mobiliser un grand nombre d'unités de calcul. La nature quasi-statique de l'équilibrage de la charge sous MPI contraint l'exécution du code par des barrières de synchronisation et des étapes de re-décomposition des sous-domaines inévitables pour suivre l'évolution du système simulé. De plus, la performance parallèle du solveur à mémoire distribuée est fortement influencée par les couplages pouvant intervenir entre les sous-domaines via les cinématiques. L'optimisation de cet aspect de la stratégie parallèle d'EPX, décrit notamment dans [56] ne fait pas l'objet du présent travail.

Au contraire, nous nous intéressons dans la suite à l'exploitation du parallélisme à mémoire partagée, complémentaire du précédent et correspondant à la structure multi-coeurs des noeuds de calcul interconnectés. Cette implémentation est traduite par l'ajout d'un parallélisme de boucles à mémoire partagée à l'intérieur des sous-domaines, pour exploiter efficacement les cœurs disponibles. Cette association est rendue pertinente dans le cas général par la complexité de la gestion des connexions cinématiques (contact unilatéral, interaction fluide-structure). Ces connexions sont certes cruciales pour les systèmes physiques considérées, mais génératrices de nombreuses communications et synchronisations dans le processus de résolution distribuée. Des solutions palliatives performantes ont été conçues et implémentées pour préserver autant que faire se peut l'extensibilité en fonction du nombre de sous-domaines [55]. Cependant, l'exploitation d'un grand nombre d'unité de calcul ne peut reposer que sur cette seule voie stratégique.

Au contraire, le recours à un parallélisme efficace en mémoire à l'intérieur d'un sous-domaine permet d'exploiter efficacement les calculateurs visés en repoussant les limites de l'extensibilité du formalisme à mémoire distribuée. Schématiquement, un sous-domaine se trouve alors affecté à un nœud de cluster, dont les cœurs internes sont exploités via une approche à base de parallélisme de boucles. Ce dernier élément stratégique revêt une importance majeure dans le passage à l'échelle global de l'application et fait l'objet des contributions du travail proposé dans ce document.

Il est noté que l'extensibilité d'EPX a été obtenue sur plus de 1000 cœurs avec les développements du projet ANR RePDyn, sur des modèles totalement représentatifs des applications industrielles du programme (cf. Figure 4.8). L'efficacité parallèle est toutefois restée très perfectible, notamment au niveau de l'exploitation des nœuds

2. http ://www.repdyn.fr/

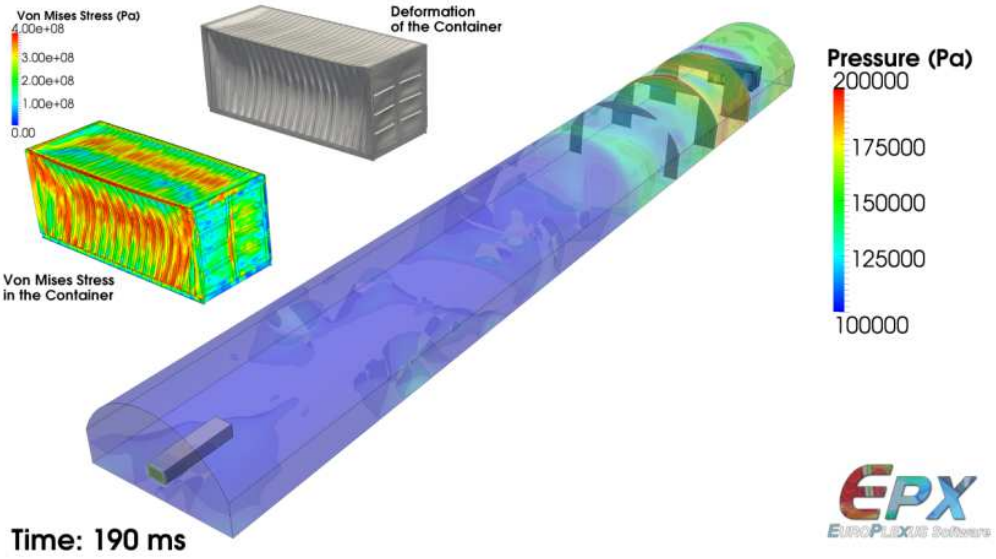


FIGURE 4.8 – Interaction entre une onde de choc guidée et un container métallique avec EPX

multi-cœurs internes aux sous-domaines, ce qui justifie pleinement les travaux de recherche engagés sur ce thème.

EPX est caractérisé par des modèles matériaux complexes et l'association de formulations différentes au sein d'un même modèle (éléments finis différents pour les structures, cohabitation entre approches lagrangiennes et eulériennes). Il en résulte des boucles de calcul produisant classiquement de nombreux accès à la mémoire dans les itérations et présentant des coûts hétérogènes d'une itération à l'autre. La boucle principale pour le calcul des forces internes et des flux de quantités eulériennes sur les faces des cellules est représentative de cette complexité et fait l'objet des développements proposés.

Cela ne nuit en rien à la généralité de l'approche, les concepts mis en œuvre pouvant aisément être généralisés aux autres boucles consommatrices du programme, souvent de nature moins complexe.

Comme introduit dans la section 4.3.2, on s'intéresse donc dans la suite à une situation où la boucle élémentaire principale est dominante, en forçant volontairement l'utilisation d'un modèle matériau dont le coût de traitement varie d'une cellule à l'autre en fonction de l'état local du système physique considéré et caractérisé par l'utilisation de nombreuses données, locales à une cellule ou prises sur ses voisins (i.e. partageant une face avec elle au sens du maillage).

4.4.2 Parallélisme à mémoire partagée

L'approche proposée pour le parallélisme à mémoire partagée est basée sur un équilibrage dynamique de charge dans un contexte de simulation en dynamique rapide. Cette version parallèle a été initialement implémentée à l'intérieur des sous-domaines traités dans le cadre de l'implémentation parallèle à mémoire distribuée,

mais peut-être évaluée et optimisée de manière autonome, conformément à la stricte complémentarité des deux sources de parallélisme dans EPX.
Dans la première phase de notre projet, nous avons procédé à une analyse des performances de la version parallèle à mémoire partagée déjà implémentée dans EPX pour mieux situer notre optimisation. Pour ce faire, nous avons mesuré le temps écoulé dans la boucle élémentaire parallélisée par la bibliothèque X-KAAPI.
La courbe de la figure 4.9 représente le temps réel écoulé dans la boucle élémentaire par le thread le moins rapide T_{Boucle} en fonction du nombre de threads impliqués dans l'exécution de la boucle. Les threads impliqués dans l'exécution de la région parallèle sont répartis de manière successive sur les cœurs des processeurs successifs. Le $i^{ème}$ thread s'exécute sur le $i^{ème}$ cœur.
La courbe de la figure 4.10 représente l'accélération relative des tâches élémentaires *Acc*. À partir de la courbe de la figure 4.9, nous constatons que T_{Boucle} diminue

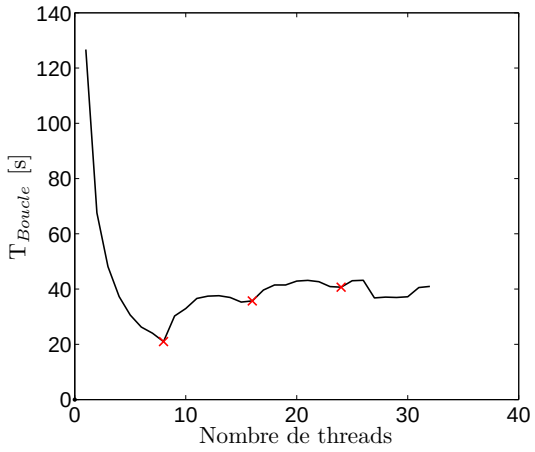


FIGURE 4.9 – Variation du temps d'exécution des tâches élémentaires en fonction du nombre de threads

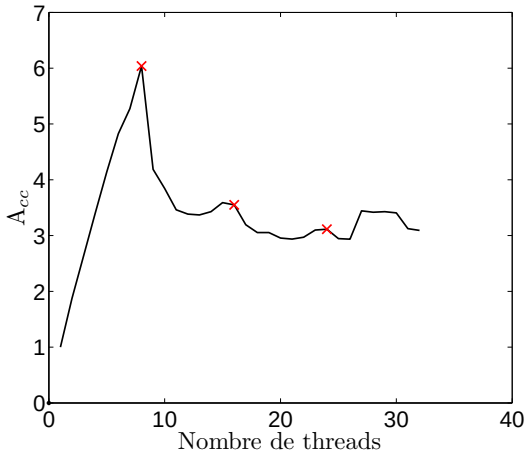


FIGURE 4.10 – Accélération des tâches élémentaires par rapport à l'exécution séquentielle

en augmentant le nombre de threads dans l'intervalle d'entier $[\![1; 8]\!]$ sachant que nous disposons d'un seul thread par cœur. Ensuite, en passant à 9 threads le temps d'exécution augmente anormalement pour se remettre à baisser légèrement avant de connaitre une deuxième augmentation pour un nombre de threads égal à 16. Ce phénomène s'est reproduit également pour un nombre de threads égal à 24.

Si nous récapitulons les valeurs remarquables des nombres de threads pour lesquels nous avons noté les maxima locaux de T_{Boucle} nous obtenons : 9, 17, 25. Ces valeurs correspondent respectivement à l'utilisation des 8 cœurs du processeur et d'un cœur supplémentaire hébergé sur un autre processeur libre, à l'utilisation de deux processeurs et d'un cœur hébergé sur un processeur libre et à l'utilisation de 3 processeurs et un cœur appartenant à un $4^{ième}$ processeur libre.

Nous avons mesuré le temps mis par chaque thread pour effectuer le travail qui lui a été attribué. Nous reportons, dans la table 4.2, le temps écoulé par chaque thread dans l'exécution de la boucle parallèle sur 9 cœurs du nœud pollux dont 8 sont sur le même socket.

TABLE 4.2 – Temps écoulés par thread pour l'exécution de la boucle élémentaire d'EPX sur 9 cœurs du nœud Pollux

Indice du thread	$\mathbf{T_{Th}(s)}$
0	22, 340
1	22, 102
2	22, 102
3	21, 867
4	21, 911
5	22, 009
6	22, 763
7	22, 980
8	30, 26

TABLE 4.3 – Temps écoulé par thread dans la boucle parallèle du programme Test_Pi sur 9 cœurs

Indice du thread	$\mathbf{T_{Th}(s)}$
0	11, 493
1	11, 504
2	11, 519
3	11, 486
4	11, 478
5	11, 488
6	11, 490
7	11, 985
8	15, 980

Nous notons que pour les tests effectués dans la cadre de cette étude les itérations des boucles parallèles ont été réparties de manière statique en bloquant les vols KAAPI pour être sûr que tous les threads ont la même quantité de travail à effectuer.

À partir des valeurs de cette table, nous constatons que le $9^{\text{ième}}$ thread met un temps plus long pour effectuer son travail. Il ralentit par conséquent les autres threads situés sur le premier socket. La même constatation a été notée pour les tests impliquant 17 et 25 threads.

Pour déceler la source de cette anomalie, nous avons refait le même test pour 9 threads sur un programme *Test_Pi* qui calcule simplement la valeur de π de manière répétitive dans une boucle parallèle. En mesurant le temps T_{Th} mis par chaque thread pour effectuer les itérations qui lui ont été attribuées, nous avons reproduit le même comportement observé dans EPX.
Nous reportons dans la table 4.3 les valeurs de T_{Th} pour les 9 threads qui ont participé à l'exécution de la boucle.

Sur la base des résultats obtenus avec ce dernier test, et du fait que les 9 threads avaient exactement la même quantité de travail à effectuer, notre premier soupçon retenu est la politique de gestion de la fréquence des processeurs.

4.4.3 Variation de la fréquence des processeurs

La plupart des processeurs actuels sont dotés d'un mécanisme de changement de fréquence. Ce mécanisme est communément appelé *Dynamic Frequency Scaling*. Il permet de réduire le nombre d'instructions à exécuter par seconde pour les processeurs sous-utilisés afin de réduire leur consommation énergétique.
Les architectures actuelles comportent différents niveaux de réglage de fréquence. Généralement, le mode configuré par défaut par le constructeur est le mode *ondemand*. Ce mode consiste à utiliser une fréquence minimale au départ, ensuite, la faire varier à la demande selon la charge du processeur. Ainsi, un processeur chargé à 100% atteint sa fréquence maximale. Ce mode est certes efficace pour augmenter la durée de vie du matériel, cependant, il engendre un ralentissement des processeurs peu chargés.
En ce qui concerne la configuration des processeurs de notre nœud, nous nous sommes rendu compte qu'ils sont initialement configurés en mode *ondemand*. Cette configuration explique la lenteur des cœurs qui appartiennent à des processeurs sous-utilisés par rapport à ceux des processeurs pleinement chargés.
Nous avons modifié ce mode vers une configuration permettant d'avoir une fréquence maximale pour tous les processeurs indépendamment de leur charge. Le mode qui assure ce comportement est le mode *performance*. Bien qu'il est moins efficace en terme d'économie d'énergie, il est primordial pour les études de performances.
Nous avons repris les mêmes tests avec la nouvelle configuration de la fréquence en mode performance. Nous avons réalisé que l'écart des temps d'exécution entre les 8 threads s'exécutant sur le premier processeur et le $9^{\text{ième}}$ thread qui s'exécute sur le processeur à part s'est manifestement réduit dans le cas du programme *Test_Pi* ainsi que dans cas du code EPX. Les nouvelles valeurs sont reportées dans les tables 4.4 et 4.5.
Nous conservons cette configuration pour toutes les expérimentations effectuées dans le cadre de cette thèse.
Cependant, bien que la fixation de la fréquence nous a permis d'avoir des résultats stables pour le cas du programme *Test_Pi*, l'amélioration de la scalabilité du code EPX n'a pas été très notable. Ainsi, nous dirigeons notre axe de recherche vers l'analyse des accès mémoire qui possèdent une influence importante sur les performances. À ce niveau de notre analyse, il est probable que le passage au deuxième socket engendrant un échange de données à travers les mémoires distribuées des processeurs de notre plateforme de calcul affecte les performances et pourrait faire partie des causes de l'anomalie observée au niveau de la courbe de scalabilité (cf. figure 4.9).

4.5 Choix d'un cas de calcul de démonstration

Pour répondre aux attentes évoquées ci-dessus, nous considérons une situation d'intérêt industriel, à savoir le dimensionnement mécanique d'une cuve métallique

TABLE 4.4 – Temps écoulés par thread dans la boucle parallèle du programme Test_Pi en mode *performance*

Indice du thread	$\mathbf{T}_{Th}$**(s)**
0	11,409
1	11,407
2	11,385
3	11,381
4	11,382
5	11,387
6	11,388
7	11,386
8	11,415

TABLE 4.5 – Temps écoulés par thread dans la boucle élémentaire du programme EPX en mode *performance*

Indice du thread	$\mathbf{T}_{Th}$**(s)**
0	22,809
1	23,007
2	23,085
3	22,981
4	23,982
5	22,987
6	22,988
7	23,010
8	26,015

soumise à une explosion interne. Cette situation est caractéristique de la sûreté des réacteurs nucléaire de 4ème génération. Le fluide interne correspond à un liquide (eau ou sodium), surplombé d'un ciel de pile (air ou argon). Un troisième composant gazeux intervient dans le système sous la forme d'un gaz d'explosif initialement comprimé au niveau du cœur du réacteur et exerçant le chargement mécanique sur le système. On donne dans la Figure 4.11, une illustration de cette gamme de simulation à l'échelle réacteur. Cette simulation consiste en la détente d'une bulle comprimée formée au niveau du cœur dans la cuve du réacteur, impactant les structures immergées environnantes et provoquant un soulèvement brutal de la surface libre du caloporteur.

Dans la configuration technologique réelle, de nombreuses structures sont immergées à l'intérieur de la cuve. Le calcul des forces internes dans ces éléments de structures est négligeable devant l'effort de résolution associé au fluide interne. Ils contribuent au coût de calcul global principalement par les liaisons cinématiques d'interaction fluide-structure qu'ils portent. Comme ce dernier point ne fait pas partie des objectifs portés prioritairement par le présent travail doctoral, ces structures internes sont négligées.

En suivant le même raisonnement, on considère l'enveloppe extérieure rigide dans les calculs à venir, si bien que le modèle de calcul est composé uniquement du fluide interne.

Comme attendu, la représentation est eulérienne. La formulation mêle l'approche éléments finis pour le calcul des forces internes via l'équation de conservation de la quantité de mouvement et l'approche volumes finis pour l'expression des flux de masse et éventuellement d'énergie à travers les faces des cellules.

De même, la loi d'état considérée présente les caractéristiques requises pour tester les capacités des approches de gestion de la mémoire et de parallélisme évoquées dans les chapitres précédents. Il s'agit d'une loi multi-matériaux comportant un liquide (eau) et deux gaz (gaz parfait pour l'air et gaz polytropique pour l'explosif dans le cas présent) en transformation adiabatique. Cette loi est désignée sous le nom ADCR dans EPX.

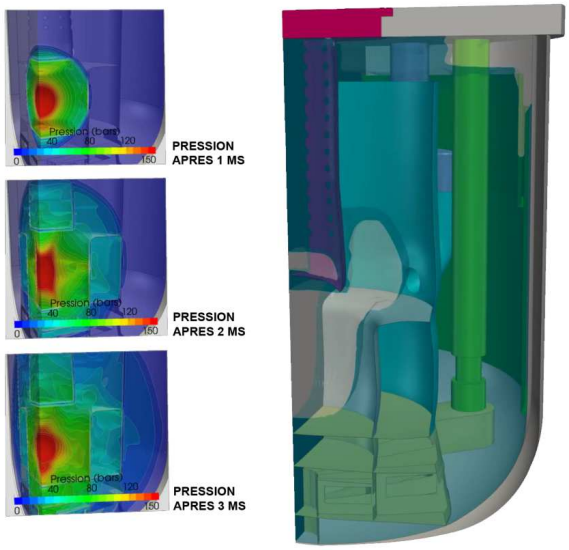


FIGURE 4.11 – Simulation de l'accident de référence dans un réacteur de IV$^{\text{ème}}$ génération

La densité totale et la fraction massique de chaque composant du mélange est obtenue à partir des flux de masse par composant. Ces flux sont connus une fois le champ de vitesse nodal défini via le schéma d'intégration explicite.
On obtient la pression via un système itératif ajustant les fractions volumiques de chaque composant pour équilibrer les pressions partielles : on produit ainsi comme recherché un algorithme consommateur en opérations arithmétiques et en accès mémoire. De plus, le coût du traitement d'une cellule varie suivant qu'elle est remplie d'un unique matériau ou d'un mélange de plusieurs des matériaux, ce qui répond à l'objectif d'hétérogénéité des coûts par itérations dans la boucle élémentaire. On donne finalement sur la Figure 4.12 une illustration du modèle de calcul mis en œuvre pour les mesures et les démonstrations à venir.

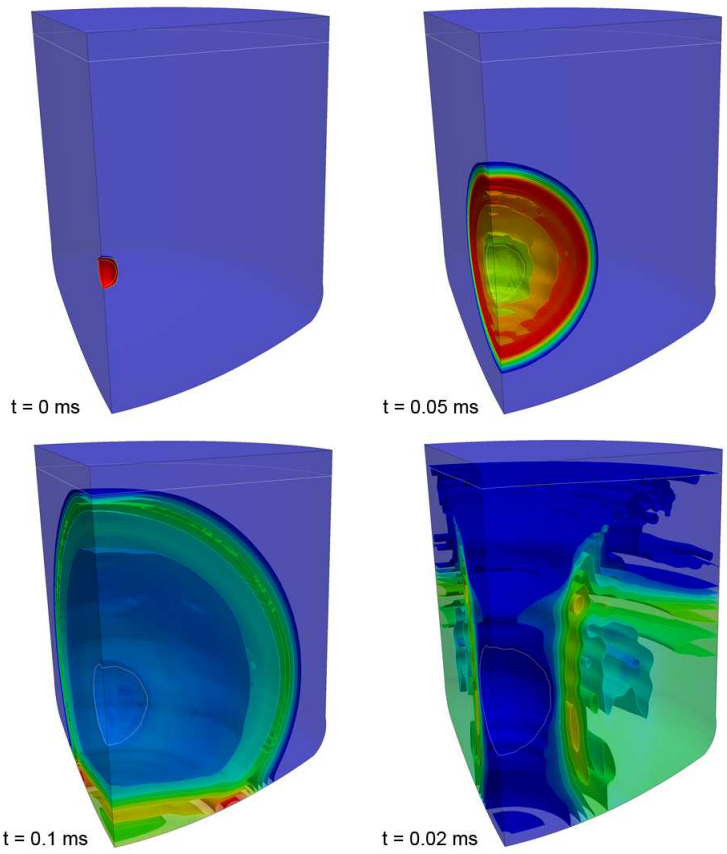


FIGURE 4.12 – Simulation de l'essai MARA2 avec EPX

Chapitre 5

Optimisation de la localité : Réorganisation de la structure des données

Dans le présent chapitre, nous présentons l'approche de réorganisation de la structure de données que nous avons développé. Nous commençons par un exposé du principe de l'approche dans la première section de ce chapitre. Ensuite, nous détaillons son implémentation dans le code EPX. Nous consacrons la dernière partie de ce chapitre pour l'évaluation de cette implémentation.

Sommaire

5.1 Objectifs

L'enjeu consiste à implémenter une stratégie de réorganisation de la structure de données du code EPX qui nous permet d'avoir le maximum de données utiles dans le cache. Nous avons vu que dans la structure de données de référence, même si la problématique de la localité géométrique des nœuds et des éléments a été bien traitée par des algorithmes spécifiques au niveau du maillage, la préservation de la localité spatiale des données en mémoire est encore perfectible.

Ainsi d'après la structure du code EPX, nous avons noté que la boucle élémentaire est le lieu de nombreux chargements de données à partir des tableaux globaux par champs physiques. Il en résulte un ralentissement potentiel des calculs élémentaires dû aux chargements multiples des données à partir de la mémoire distante au cours de l'exécution des tâches élémentaires.

De ce fait, notre objectif consiste à mettre en place une approche *cache-aware* qui permet de :

- contrôler la taille des blocs de données utilisés par chaque unité de calcul,
- avoir le maximum de données utiles dans le cache pour le calcul d'un ensemble d'éléments successifs,
- travailler sur des blocs de données contigus pour une meilleure localité dans le cache,
- effectuer le moins d'interventions possible sur le code,
- limiter les modifications aux routines externes,
- définir une stratégie d'ordonnancement de boucle bénéficiant de la localité des données pour l'exécution parallèle avec la bibliothèque XKAAPI,
- assurer l'indépendance des blocs de données accédées pour un meilleur gain en mode parallèle.

5.2 Approche `Par_gpe`

L'organisation de la structure de données du code EPX s'appuie sur le modèle SOA (cf. section 2.5.2 page 39). Dans ce modèle, les données relatives à un élément du maillage proviennent de différents tableaux de grandeurs physiques. Ce modèle est fonctionnel pour les actions portant sur des vecteurs de grandeurs particulières (calcul des vitesses et des accélérations ou traitement des liaisons cinématiques dans EPX par exemple), mais il peut être significativement amélioré les phases intensives de calculs sur les éléments.

La réorganisation que nous proposons est inspirée du modèle hybride SOAOS basé sur la réorganisation des données sous forme de *tableaux de structures* (modèle AOS) et ce pour un ensemble de structures d'où l'aspect hybride de notre modèle.

Dans notre cas, *une structure* au sens du modèle de réorganisation des données désigne le paquet de données nécessaires pour effectuer un calcul élémentaire. Conséquemment, dans le cadre de l'approche hybride que nous adoptons, nous regroupons dans un même paquet des données nécessaires pour effectuer plusieurs calculs (itérations) élémentaires en utilisant des données locales.

Nous débutons par une catégorisation des données pour en déduire le contenu

des paquets élémentaires que nous rassemblons dans des *groupes*.

Un *groupe* est un ensemble de paquets contenant les données nécessaires pour effectuer le calcul de elt_{gr} éléments.

Ainsi, notre approche ne se limite pas au traitement d'un seul élément par groupe mais elle permet d'avoir les structures d'un ensemble d'éléments à la fois. De la sorte, l'approche *Par_gpe*, nous permet de profiter des apports du modèle AOS d'une part et des avantages du modèle SOA en chargeant elt_{gr} champs de grandeurs physiques dans le même groupe d'une autre. Nous détaillons les étapes que nous avons suivies pour parvenir à mettre en œuvre cette approche.

5.2.1 Classification des données

À l'issu d'une opération de discrétisation et de maillage, les éléments obtenus possèdent des caractéristiques distinctives. Le déroulement d'une simulation en éléments finis est basé sur l'étude des éléments à travers la variation de leurs caractéristiques au cours du temps. Cette évolution se traduit par des calculs élémentaires.

Ce raisonnement nous a conduit à un questionnement sur la liste des données nécessaires et suffisantes pour mener un calcul élémentaire à terme.

Pour cette liste d'ingrédients, nous nous sommes placés dans le cas le plus général de simulation en éléments finis et nous avons focalisé notre étude sur un élément e_i entouré de 4 voisins *Ev1, Ev2, Ev3* et *Ev4* (cf. figure 5.1). Cette analyse nous a permis de distinguer trois familles de données essentielles pour le déroulement d'un calcul élémentaire.

La première famille que nous appelons **famille nodale** comporte toutes les données relatives aux nœuds de l'élément (les tableaux N_i sur la figure 5.1). Elle contient, notamment, les incréments des déplacements nodaux, les vitesses nodales, les accélérations nodales, etc ...

La deuxième famille, la **famille élémentaire**, contient toutes les informations qui se rapportent à l'élément en soi (les tableaux E_i sur la figure 5.1). On y intègre les déformations qui affectent la géométrie de l'élément en question, les efforts surfaciques exercés sur l'élément (les contraintes), etc ...

Partant du fait qu'un élément partage des nœuds et des faces avec des éléments voisins, un échange éventuel entre un élément et son voisinage doit être pris en compte dans le cas général intégrant les représentations eulériennes et ALE (un tel échange est nul en représentation lagrangienne). Il se produit sous la forme de flux de masse ou de flux d'énergie à travers les faces communes entre l'élément et ses voisins (les tableaux V_i sur la figure 5.1). Nous regroupons ces informations décrivant les échanges entre l'élément et son voisinage dans la **famille de voisinage**.

5.2.2 Construction des groupes

L'approche *Par_gpe* consiste à créer un espace de travail local. Ceci permet aux unités de calcul qui participent à l'exécution de l'application d'effectuer leurs calculs élémentaires respectifs sans avoir besoin de charger des données supplémentaires à partir des tableaux globaux.

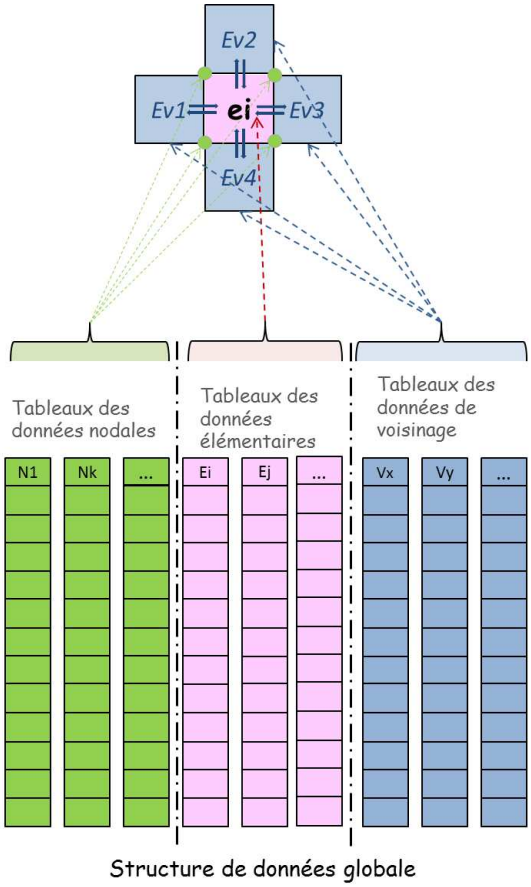


FIGURE 5.1 – Catégorisation des données en familles

Ainsi, pour assurer ce passage du milieu global vers l'espace local, nous avons procédé à une copie des données issues des trois familles qui caractérisent l'élément en tenant compte de son voisinage (cf. section 5.2). Cette opération de copie est effectuée pendant la phase d'initialisation en amont de la boucle élémentaire.
Nous désignons par :

- **nelem** le nombre total d'éléments du maillage à étudier ;
- **ngr** le nombre total de groupes à construire à partir des Nelem éléments ;
- $\mathbf{elt_{gr}}$ le nombre d'éléments par groupe.

Le nombre d'éléments par groupe est un paramètre réglable selon les caractéristiques et la complexité du système simulé.

À partir de la valeur de elt_{gr} et par la donnée de *nelem*, nous déduisons le nombre de groupes *ngr* que l'on peut construire :

$$ngr = \lfloor \frac{nelem}{elt_{gr}} \rfloor + Rest \tag{5.1}$$

Tels que :

- Pour $X \in \mathbb{R}$, $\lfloor X \rfloor$ représente la partie entière de X
-

$$Rest = \begin{matrix} 0 & \text{si} \lfloor \frac{nelem}{elt_{gr}} \rfloor \times elt_{gr} = \text{nelem} \\ 1 & sinon \end{matrix} \tag{5.2}$$

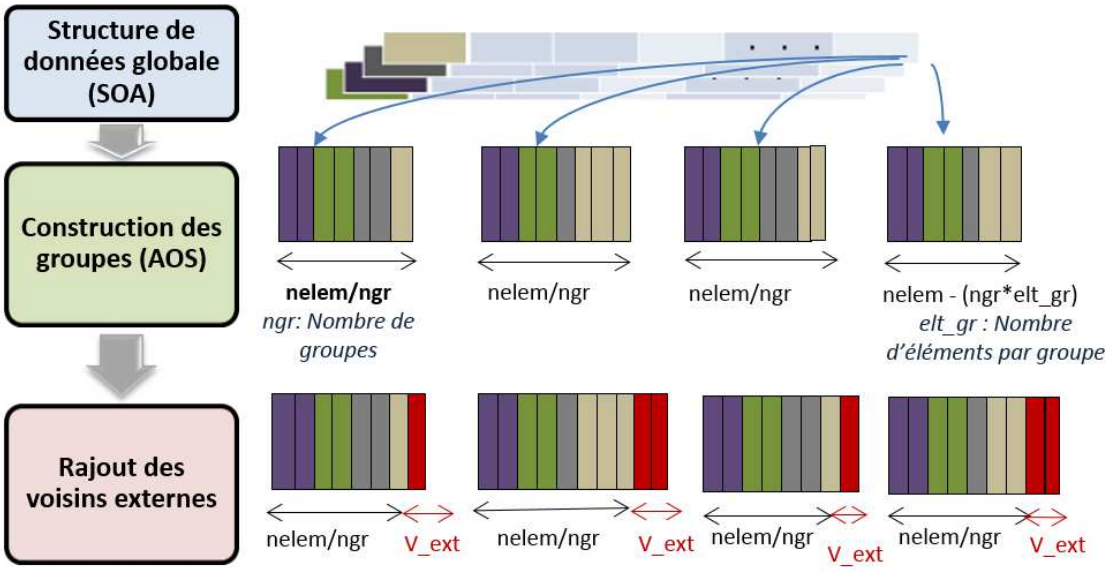


FIGURE 5.2 – Construction des groupes à partir de la structure de données globale

D'après les équations 5.1 et 5.2, la taille du dernier groupe est inférieure ou égale à la taille des autres groupes.

Les positions des éléments dans les groupes correspondent à l'ordre de leur utilisation lors de l'exécution des itérations de la boucle élémentaire. Dans cet ordre, le chargement des données relatives à un élément engendre également le chargement du contenu de ses cases voisines en mémoire. Ces dernières seront accédées par le processeur dans les instants suivants, respectant ainsi le principe de la localité spatiale du cache.

Dans le but de conserver la généricité du programme, le nombre de champs par élément est déduit à partir du type de ce dernier et de sa géométrie. Cette propriété élargit le champs d'application de notre approche pour différentes simulations en éléments finis.
L'hétérogénéité des géométries des éléments à traiter est prévue et prise en compte au moment de la construction des groupes.
Le nombre de face par élément ainsi que le nombre de ses nœuds sont connus pour chaque élément du groupe. Cette information est exploitée pour la détection des voisins de l'élément.

5.2.3 Extension des groupes

À ce stade nous sommes parvenus à construire des groupes respectant la localité spatiale en mémoire et adaptés à différentes simulations en éléments finis de par la généricité de l'approche `Par_gpe`. Cependant, ces groupes ne sont pas indépendants entre eux. En effet, les voisins des éléments ne font pas forcément partie des éléments locaux et il est possible qu'ils appartiennent à d'autres groupes. Par conséquent, des échanges entre les éléments du groupe courant et des éléments externes peuvent

avoir lieu et nuire à la localité des traitements élémentaires.
Pour éviter ces échanges inter-groupes, nous aimerions inclure les voisins externes dans les groupes avec lesquels ils établissent des échanges et rendre les communications locales. Pour ce faire, la méthode la plus triviale consiste à dupliquer ces voisins externes dans les groupes qui les utilisent.
Cette méthode permet d'éviter les accès distants dus aux échanges avec des éléments externes, surtout lors d'un traitement des groupes par plusieurs threads à la fois ; mais par ailleurs, elle engendre un gaspillage de la mémoire puisque les données dupliquées ne seront pas toutes exploitées lors du calcul des échanges avec les éléments voisins.
Pour limiter cette redondance des données, nous avons introduit dans les groupes des *pseudo-éléments* que nous appelons V_{ext}. Ils représentent les voisins externes avec lesquels les éléments locaux au groupe vont échanger au cours du calcul.
Ces pseudo-éléments sont caractérisés uniquement par leurs données provenant de la famille de voisinage. Les données nodales et les données élémentaires ne sont pas incluses dans la liste des données de V_{ext}. Cette restriction est justifiée par le fait que les éléments externes qui assurent le rôle de voisins dans un groupe donné font effectivement partie d'un autre groupe et ils y sont entièrement définis et calculés en tant qu'éléments locaux.
Nous désignons par *indice global* la position de l'élément dans la liste globale de tous les éléments du maillage.
Nous désignons par i_{deb} l'indice du premier élément du groupe dans la liste globale des éléments du maillage et par i_{fin} l'indice global de son dernier élément. Les définitions de ces deux bornes sont données par le pseudo-algorithme 3.

Algorithme 3 : Définition des bornes du groupe

1 **Entrées :** nombre d'éléments par groupe elt_{gr}, nombre d'éléments total *nelem*, nombre de groupes *ngr* ;
2 **Pour** $j \in [\![1; ngr]\!]$ **faire** ;
3 $\quad i_{deb}(j) = elt_{gr} \times (j-1) + 1$
4 $\quad i_{fin}(j) = \min\{i_{deb} + elt_{gr} - 1, nelem\}$
5 **Fin Pour**

La détermination de la liste des voisins d'un élément e_i, est effectuée en itérant sur les différentes faces de ce dernier. Pour chaque face, nous notons par idx_v l'indice global du voisin de e_i.

- **si** $idx_v \in [\![i_{deb}; i_{fin}]\!]$ **alors** le voisin est un élément local au groupe. Dans ce cas l'élément d'indice idx_v sera doublement exploité une fois chargé dans le cache.
- **sinon** le voisin est un V_{ext}.

La notation idx_{loc} désigne la position de ce dernier par rapport aux éléments de son groupe. Elle est déduite comme suit :

$$idx_{loc} = idx_g - i_{deb} + 1 \tag{5.3}$$

Le passage des indices globaux vers les indices locaux est lié au travail sur des structures locales favorisant la localité des accès mémoire. En revanche, le passage inverse est nécessaire pour accéder à des tableaux de la structure de données globale.

Algorithme 4 : Pseudo-code de la routine de calcul dans la version de référence

```
1 Entrées : nombre total d'éléments nelem, tableaux de la structure de données globale
    glob
2 Initialisations globales
3         Pour j ∈ ⟦1; nelem⟧ faire ;
4                 Appel Calcul_elem(glob)
5         Fin Pour
6 Pas_T ← Pas_T +1
```

5.3 Version séquentielle de l'approche Par_gpe

5.3.1 Implémentation

L'activation de l'approche *Par_gpe* dans le programme EPX est optionnelle. Cette configuration offre la possibilité à l'utilisateur de choisir entre le modèle SOA représenté par la structure de données originelle (version de référence du code) et le modèle hybride SOAOS proposé par l'approche *Par_gpe*.
Les modifications majeures du programme sont localisées au niveau de la routine de calcul autour de la boucle élémentaire.
Une fois les initialisations achevées et avant d'entamer les calculs élémentaires, nous faisons appel à la routine **Glob_loc** responsable de la création des groupes par passage de la structure globale vers les structures locales (les groupes).
Pour chaque groupe créé, nous itérons sur ses éléments en utilisant leurs données locales pour effectuer les calculs élémentaires.
À l'achèvement de tous les calculs élémentaires du groupe courant, une étape de mise à jour des données de la structure globale est nécessaire avant de détruire la structure locale et passer au calcul des éléments du groupe suivant (routine **Loc_glob**).
Dans l'algorithme 4 nous décrivons l'organisation de la routine de calcul dans la version de référence du code. Dans cette version, tous les calculs élémentaires se déroulent dans les itérations de la boucle principale (lignes 3 $\rightarrow$ 5 du pseudo-code). L'ensemble des routines de *Calcul_elem* utilisent des données issues des tableaux globaux (*glob*).

Nous décrivons de manière simplifiée le peudo-code de cette même partie du programme EPX après l'implémentation de l'approche *Par_gpe* dans l'algorithme 5. Dans cette version, nous implémentons une boucle sur les groupes autour de la boucle élémentaire *Inner_loop*. Chaque itération de la boucle externe *Outer_loop* constitue un espace local pour l'exécution des tâches élémentaires sur des données contiguës en mémoire (*loc*). Elle commence par un chargement des données des éléments du groupe ainsi que celles de leurs voisins externes dans la structure de données locale (cf. Algorithme 6 de la routine *Glob_loc(glob)*). Ensuite, nous appliquons la routine *calcul_elem* sur les éléments du groupe courant. Une fois les calculs élémentaires terminés, nous mettons à jour la structure de données globale (*Loc_glob(loc)*) avec les données que nous venons de calculer en local et nous passons au calcul du groupe suivant.

Algorithme 5 : Pseudo-code de l'implémentation de l'approche Par_gpe

```
1  Entrées : nombre d'éléments par groupe elt_gr, nombre d'éléments total nelem, nombre
     de groupes ngr, tableaux de la structure de données globale glob
2  Initialisations globales
3  Allocation dynamique des tableaux locaux
4  Pour j ∈ [[1; ngr]] faire ;
5          i_deb(j) = elt_gr × (j − 1) +1
6          i_fin(j) = min {i_deb(j) + elt_gr - 1, nelem}
7          loc ← Glob_loc(glob)
8          Pour j ∈ [[i_deb; i_fin]] faire ;
9                  Appel Calcul_elem(loc)
10         Fin Pour
11         glob ← Loc_glob(loc)
12 Fin Pour
13 Désallocation des tableaux locaux
14 Pas_T ← Pas_T +1
```

5.3.2 Étude expérimentale

5.3.2.1 Influence de l'affinité CPU

Dans les architectures NUMA, la politique d'allocation mémoire possède une influence significative sur les performances des applications.
Lorsque cette allocation n'est pas contrôlée, les données sont allouées par défaut sur le banc mémoire du processeur qui les a utilisées en premier. C'est la politique du *first-touch*.
Parmi les politiques disponibles sous Linux nous citons la politique *interleave* de l'outil *numactl* qui consiste à allouer une page par banc mémoire de manière cyclique.
Le placement des données sur un banc mémoire désigné par l'utilisateur est possible en utilisant la stratégie *membind* de *numactl*[1]. Cette configuration permet d'utiliser uniquement la mémoire du nœud désigné par l'utilisateur. Cependant, bien que cette stratégie soit efficace pour le choix du banc mémoire à utiliser elle ne permet pas le contrôle total de l'exécution de l'application. En fait, pour des raisons matérielles consistant à améliorer la durée de vie des unités de calcul, le système d'exploitation a la possibilité de migrer les threads ce qui peut poser des problèmes de localité. Pour remédier à ce problème, la politique *physcpubind* permet de préciser le processeur sur lequel l'application doit s'exécuter. L'association de cette dernière stratégie avec la stratégie *membind* permet ainsi de gérer l'affinité mémoire de manière plus précise en définissant à la fois le processeur sur lequel le programme va s'exécuter du début jusqu'à la fin mais aussi le placement mémoire des données qu'il va utiliser.
Nous étudions l'influence de la stratégie d'allocation mémoire sur le temps d'exécution du code EPX. Dans cette série d'expériences nous faisons varier la taille du bloc de données utilisées et nous mesurons le temps d'exécution du programme. Le jeu de données utilisé dans ce test simule la propagation d'une onde à partir d'un

1. http ://linux.die.net/man/8/numactl

Algorithme 6 : Pseudo-code de la routine Glob_loc

Entrées : elt_{gr}, *nelem*, *ngr*, tableaux de la structure de données globale *glob*, i_{deb}, i_{fin}, adresse des faces de chaque élément *fac*,
Pour $i \in [\![i_{deb}; i_{fin}]\!]$ **faire** ; *! Calcul et sauvegarde des indices locaux :*
    $idx_{loc} =$ i - i_{deb} +1 ;
    itag(i) $= idx_{loc}$;
Fin Pour
! Initialisations :
$V_{ext} = 0$, FF $= 0$, LL $+ 1$;
Pour $i \in [\![i_{deb}; i_{fin}]\!]$ **faire** ;
    $idx_{loc} =$ i - i_{deb} +1 ;
    $cach_{vois}(idx_{loc}) =$ i ;
    NbFac = fac(i + 1) - fac(i) ;
    $fac_{gr}(idx_{loc}) =$ FF ;
    Pour $j \in [\![1; NbFac]\!]$ **faire** ;
        vois = elt(Fac(i) + j -1) ;
        Si *(vois == 0)* **Alors** ;
            Elt_{loc}*(FF + j - 1) = 0 ;*
            Return ;
        FinSi ;
        Si *(itag(vois) == 0)* **Alors** ;
            $V_{ext} = V_{ext} + 1$;
            $cach_vois(elt_{gr} + V_{ext}) =$ *vois* ;
            itag(vois) $= i_{fin}$ - $i_{deb} + V_{ext} + 1$;
        FinSi ;
        Si *(vois* > 0*)* **Alors** ;
            elt_{gr}*(FF + j - 1) = itag(vois) ;*
        Else ;
            elt_{gr}*(FF + j - 1) = - itag(vois) ;*
        FinSi ;
    Fin Pour ;
    FF = FF + NbFac ;
Fin Pour ;
$fac_{gr}(i_{fin}$ - $i_{deb} + 1 + 1) =$ FF ;
Pour *chaque grandeur physique* G_i ***faire*** ;
    Pour $i \in [\![i_{deb}; i_{fin}]\!]$ **faire** ;
    ! Pointer dans le tableau global de G_i *et définir le nombre total de ses champs par élément* N_{champs} ;
        loc(LL : LL + N_{champs} - 1) $= G_i$(i) ;
        $IP_i(idx_{loc}) =$ LL ;
        LL = LL + 1 ;
    Fin Pour ;
    Pour $j \in [\![1; V_{ext}]\!]$ **faire** ;
        loc(LL) = glob($cach_{vois}(elt_{gr} + j)$*) ;*
        LL = LL + 1 ;
    Fin Pour ;
Fin Pour.

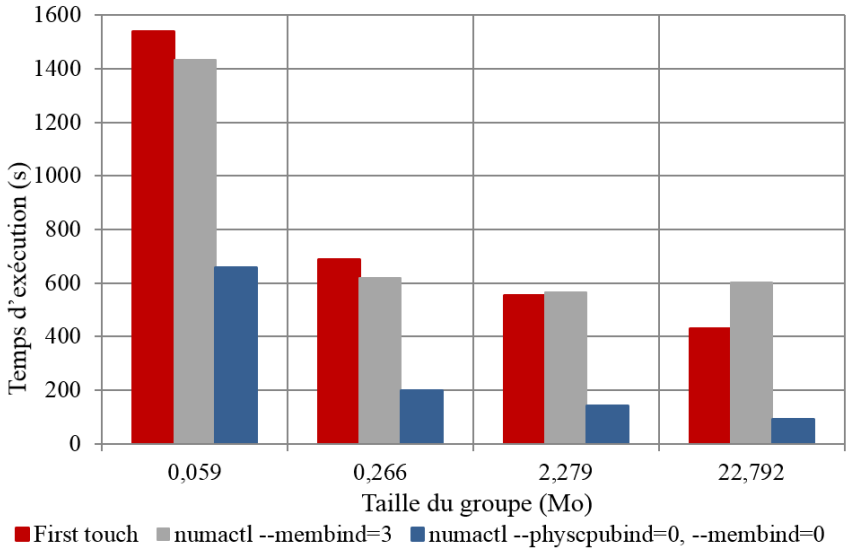


FIGURE 5.3 – Influence de la politique d'allocation sur les temps d'exécution d'une simulation EPX séquentielle

point soumis à une haute pression à travers un milieu fluide comportant 10^6 mailles. Nous comparons dans la figure 5.3 les performances obtenues pour trois stratégies d'allocation différentes fournies par l'outil *numactl* :

- stratégie d'allocation du *first touch* utilisée par défaut.
- `numactl -membind=3 europlexus test.epx` : consiste à utiliser le banc mémoire du noeud n° 3 lors de l'exécution de l'application *europlexus* sur le jeu de données *test.epx*.
- `numactl -membind=0 -physcpubind=0 europlexus test.epx` : désigne le processeur n° 0 du nœud n° 0 pour exécuter l'application *europlexus* et le banc mémoire du nœud n°0 pour y allouer les données demandées par le processeur.

Pour les différentes tailles de groupes utilisées, nous notons que les performances de la politique *first-touch* sont comparables à celles de la politique *membind*. L'absence du contrôle du placement des tâches dans ces deux cas est à l'origine de la proximité des résultats notée dans les différents tests effectués.
En revanche, l'écart dans les temps d'exécution entre ces deux politiques d'une part et la politique `-membind=0 -physcpubind=0` d'autre part est notable. Cette dernière nous a permis de réduire le temps d'exécution de plus de 25% en moyenne.
Dans la suite de notre étude séquentielle, nous adoptons cette stratégie pour les différentes expérimentations.

5.3.2.2 Évaluation du temps d'exécution séquentiel

L'objectif de cette expérience est de définir la taille de groupe optimale pour un meilleur gain en performance en environnement séquentiel et de comparer les performances du modèle SOA de la version de référence du code EPX (désignée par *Reference* dans la suite du document) avec le modèle hybride SOAOS de la version *Par_gpe*.

TABLE 5.1 – Nombre de groupes par pas de temps en fonction de leurs tailles en Mo

elt_{gr}	**Taille (Mo)**	**Groupes de Mara_mdm**	**Groupes de Mara_big**
100	0,059	2717	21730
940	0,25	289	2312
1.000	0,266	272	2173
3.000	0,968	91	725
5.000	1,521	55	435
7.000	1,595	39	311
10.000	2,279	28	218
15.000	3,418	19	145
20.000	4,558	14	109
25.000	5,698	11	87
30.000	6,837	10	73
32.000	7,293	9	68
35.000	7,977	8	63
40.000	9,116	7	55
45.000	10,256	7	49
50.000	11,396	6	44
70.000	15,954	4	32
100.000	22,792	3	22
271.615	61,907	1	8

Pour ce faire, nous testons deux variétés du jeu de données Mara2 : le jeu de données *Mara_mdm* comportant 271.615 éléments équivalents à environ 62 Mo de données nécessaires pour effectuer la totalité des calculs élémentaires et un jeu données basé sur un maillage plus raffiné *Mara_big*, composé de 2.172.920 éléments (près de 495 Mo).
Afin de mieux situer les tailles des groupes par rapport aux capacités de stockage des niveaux de la mémoire cache de notre plateforme de calcul, les tailles des groupes seront présentées en Mo. Cependant, étant donné que les groupes n'ont pas tous le même nombre de voisins externes à cause de la variabilité de la distribution spatiale de leurs éléments sur la grille du maillage, nous reportons une valeur moyenne de la taille en Mo pour chaque taille de groupe elt_{gr}.
Nous reportons dans la table 5.1 quelques tailles de groupes en éléments et leurs équivalents en Mo pour les deux jeux de données de mara2 présentés précédemment. Les colonnes *Groupes de Mara_mdm* et *Groupes de Mara_big* représentent le nombre total de groupes que l'on peut construire à partir de chaque taille de groupe respectivement pour le jeu de données Mara_mdm et Mara_big.

Nous étudions, la variation du coût de construction d'un groupe en fonction de sa taille. La figure 5.4 représente le temps moyen écoulé dans la routine `Glob_loc` (cf. Algorithme 6) pour construire un groupe divisé par la taille de ce dernier. Ces résultats sont donnés pour la version séquentielle et la version exécutée avec 32

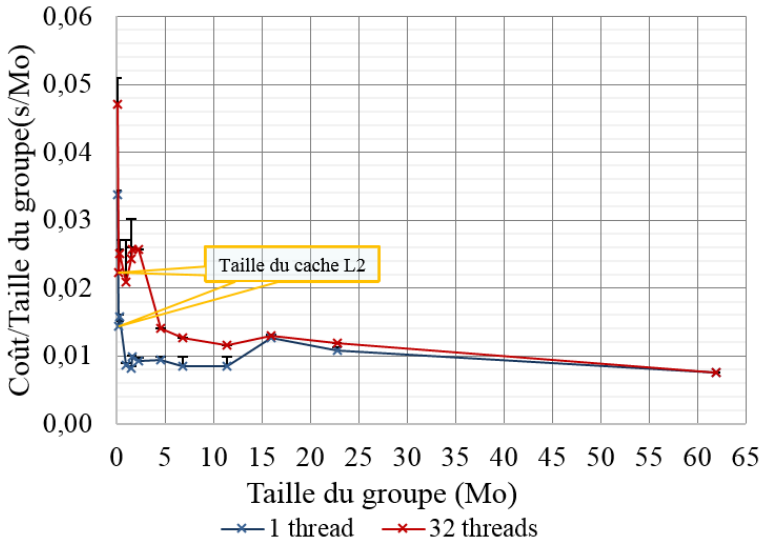


FIGURE 5.4 – Variation du coût de création des groupes en fonction de leurs tailles

threads. Nous remarquons que le ratio donné pour la version séquentielle est légèrement moins élevé que celui obtenu avec la version 32 threads. Cet écart entre les deux versions est du à l'intensification du trafic mémoire lorsqu'un grand nombre de threads tentent d'accéder simultanément à la mémoire partagée. Nous notons que, pour le groupe de taille égale à la taille du jeu de données (61 Mo), les performances des deux versions sont identiques. En effet, dans ce cas, nous n'avons qu'un seul groupe ; par conséquent, l'exécution à 32 threads est réduite à une exécution séquentielle. Ces deux courbes, montrent que le coût de construction des groupes de petites tailles est considérablement élevé (atteint 50 ms/Mo) par rapport à celui des groupes de tailles élevées qui se stabilise autour de 10 ms/Mo.

Afin d'évaluer le gain en performance apporté par la version `Par_gpe` dans un environnement d'exécution séquentiel, nous étudions l'évolution du temps *T_boucle* écoulé dans la boucle élémentaire en fonction de la taille du groupe. Dans cette étude, nous comparons la version de `Reference` avec la version `Par_gpe` basée sur l'implémentation d'une boucle sur les groupes autour de la boucle sur les éléments. Le temps d'exécution mesuré pour cette dernière version comprend le temps de construction du groupe ainsi que la mise à jour des éléments du groupe dans la structure de données d'origine du code EPX (routine `Loc_glob` de l'algorithme 5). Dans les courbes des figures 5.5 et 5.6, nous comparons les performances en séquentiel de la version `Par_gpe` avec ceux de la version `Reference` respectivement pour les deux jeux de données *MARA_mdm* et *MARA_big*.
Nous constatons que pour les groupes de tailles inférieures à 0,25 Mo, la version de `Reference` est plus performante que la version `Par_gpe` avec un écart de près de 35%. En effet, si nous considérons le groupe de taille égale à 0,059 Mo, nous remarquons qu'il est 4 fois plus petit que la taille du cache L2 (taille du cache L2 = 0,256 Mo). Ainsi, pour atteindre la taille du cache L2 en utilisant des groupes plus petits que ce dernier, nous multiplions inutilement les coûts de construction des

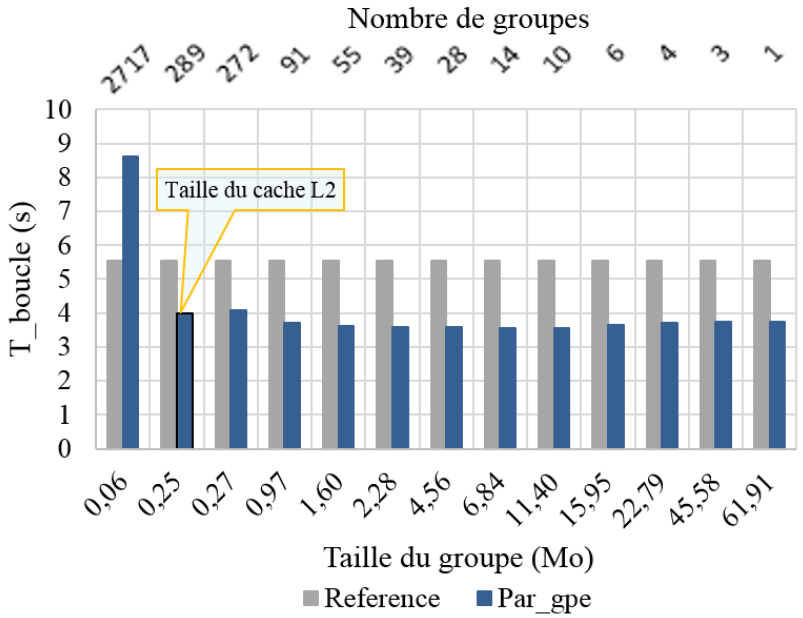


FIGURE 5.5 – Variation du temps d'exécution séquentiel de la boucle élémentaire pour MARA_mdm en fonction de la taille des groupes.

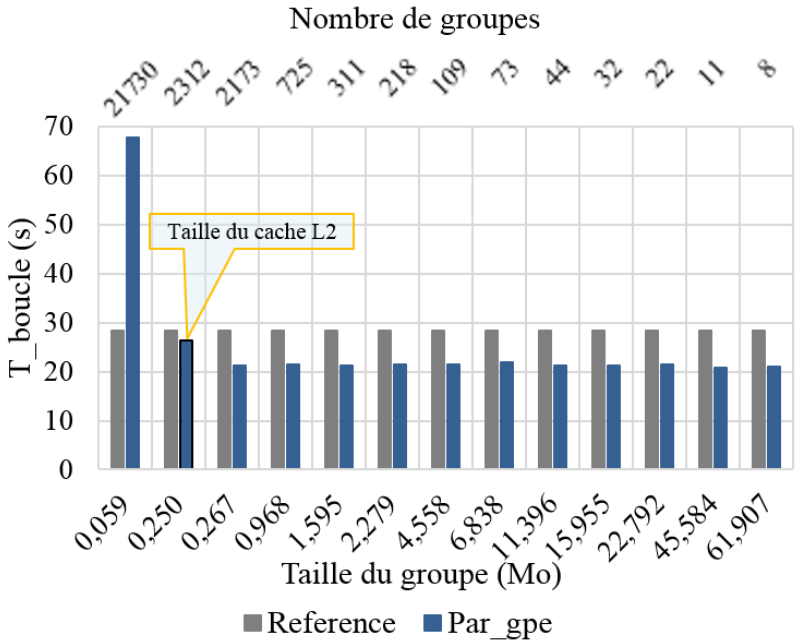


FIGURE 5.6 – Variation du temps d'exécution séquentiel de la boucle élémentaire pour MARA_big en fonction de la taille des groupes.

groupes.
En revanche, pour les groupes de tailles supérieures à 0,25 Mo, la version de `Par_gpe` est plus performante que la version `Reference` avec un écart de près de 30%. Le réarrangement des données dans des groupes contribue dans ce cas à une utilisation plus efficace du cache. Nous notons ce gain de performance également pour le cas d'une taille de groupe égale à la taille du jeu données. Ce gain pour cette taille particulière de groupe prouve que le modèle SOAOS adopté dans la version `Par_gpe` est mieux adapté que le modèle SOA de la version de `Reference` pour le code EPX. Globalement, ces résultats prouvent qu'à partir d'une taille de groupe de l'ordre de la taille du cache L2, les coûts de construction des groupes restent suffisamment faibles pour assurer un gain de performance significatif dans le cadre d'une exécution séquentielle.

5.4 Version parallèle de l'approche `Par_gpe`

5.4.1 Implémentation

Algorithme 7 : Pseudo-code de la parallélisation de *Outer_loop*

```
1  Entrées : nombre d'éléments par groupe elt_gr, nombre d'éléments total nelem, nombre
     de groupes ngr, tableaux de la structure de données globale glob
2        SUBROUTINE CALCUL(A)
3   !        Initialisations globales
4   !        ALLOCATE(local arrays)
5   !        Initialization step
6   !          ...
7              Err = KAAPIF_SET_GRAIN(...)
8              Err = KAAPIF_SET_POLICY(...)
9              Err = KAAPIF_SET_DISTIBUTION(...)
10             Err = KAAPIF_FOREACH(...,
11  &                          KAAPI_GROUPS, ...)
12  !          ...
13  !        DEALLOCATE(local arrays)
14       END SUBROUTINE CALCUL
15       Pas_T ← Pas_T +1
```

La mise en œuvre de la version parallèle de notre implémentation `Par_gpe` s'appuie sur l'utilisation de la bibliothèque de programmation parallèle XKAAPI. Cette implémentation consiste à faire collaborer plusieurs threads pour exécuter les itérations de la boucle sur les groupes que nous désignons par *Outer_loop*. La parallélisation de *Outer_loop* obéit à une structure `FOREACH` classique. La routine à paralléliser est passée en argument de la fonction `KAAPIF_FOREACH` (cf. ligne 10 de l'algorithme 7). Les calculs élémentaires de la *Inner_loop* se déroulent au sein de la routine `KAAPI_GROUPS` (cf. Algorithme 8).
Dans la section parallèle, chaque thread accède uniquement aux données locales de son groupe pour effectuer les calculs relatifs aux éléments de ce groupe. Ainsi, l'exé-

Algorithme 8 : Pseudo-code de la routine KAAPI_GROUPS

```
1 Entrées : nombre d'éléments par groupe elt_gr, nombre d'éléments total nelem, nombre
    de groupes ngr, tableaux de la structure de données globale glob
2 !            Initialisations globales
3       Pour j ∈ [[i_deb; i_fin]] faire ;
4                   Appel Calcul_elem(loc)
5       Fin Pour
```

cution de la boucle élémentaire *Inner_loop* reste désormais séquentielle au niveau de chaque itération sur les groupes.
En outre, l'inclusion des routines `Glob_loc` et `Loc_glob` dans la section parallèle implique, que chaque thread prend en charge la construction de son propre groupe ainsi que la mise à jour des données de ses éléments. Les lignes 7 → 10 représentent les fonctions nécessaires pour la configuration de l'environnement XKAAPI. Ces fonctions seront détaillées dans la section 6.1 page 100. La fonction `KAAPIF_SET_POLICY` contrôle le type d'ordonnancement à appliquer lors de la parallélisation de la boucle. L'utilisateur peut définir une distribution initiale des itérations de la boucle parallèle sur les cœurs disponibles en utilisant la fonction `KAAPIF_SET_DISTRIBUTION`. Dans le cas où l'ordonnanceur est configuré à *static*, cette distribution initiale est maintenue jusqu'à la fin de la section parallèle. En revanche, si l'ordonnancement appliqué est de type *dynamique*, dans ce cas, la charge de travail initialement distribuée sur les threads est dynamiquement réajustée par vol de tâches : lorsqu'un thread termine l'exécution de ses itérations sur les groupes, il se transforme en voleur et il commence à désigner de manière aléatoire ses victimes jusqu'à ce qu'il réussisse à trouver du travail chez une d'entre elles. Dans notre implémentation, les vols ne peuvent avoir lieu qu'à l'échelle des groupes : un voleur ne peut pas voler des itérations de la boucle élémentaire.

5.4.2 Étude expérimentale

5.4.2.1 Évaluation du temps d'exécution parallèle

Les résultats des tests analysés dans cette partie correspondent aux 4 versions suivantes, parallélisées avec XKAAPI :

- **Reference-static** : Version parallèle sans activation de l'approche `Par_gpe` basée sur une distribution statique de la charge sur les threads disponibles.
- **Reference-steal** : Version parallèle sans activation de l'approche `Par_gpe` basée sur un équilibrage dynamique de la charge par vol de travail.
- **1-foreach-static** : Version parallèle de l'approche `Par_gpe` basée sur une distribution statique de la charge sur les threads disponible.
- **1-foreach-steal** : Version parallèle de l'approche `Par_gpe` basée sur un équilibrage dynamique de la charge par vol de travail.

Nous comparons les performances de ces versions pour différents nombre de threads et en variant la taille des groupes pour les deux jeux de données *Mara_mdm* (cf. figures 5.7, 5.9 et 5.11) et *Mara_big* (cf. figures 5.8, 5.10 et 5.12).
Dans les histogrammes des résultats, l'axe principal des abscisses représente les

différentes tailles des groupes en Mo. L'axe des abscisses secondaire, en haut de chaque figure, représente le nombre total de groupes que l'on peut construire à partir de chaque taille.

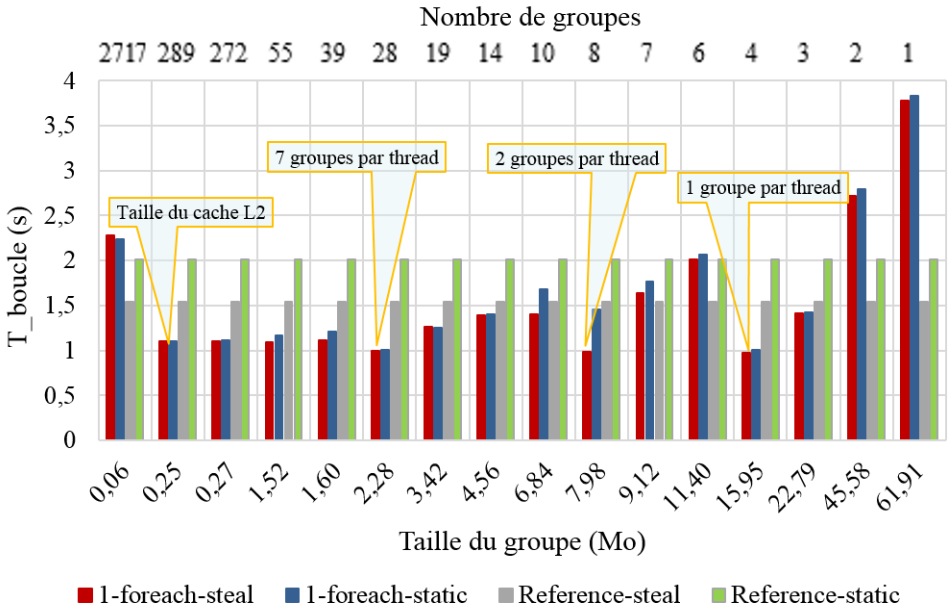


FIGURE 5.7 – Variation du temps d'exécution de la boucle élémentaire en fonction de la taille des groupes avec 4 threads pour une simulation du jeu de données MARA_mdm

Nous comparons la version `Reference-steal` et la version `Reference-static` pour une exécution avec 4, 8 et 32 threads dans le cas du jeu de données *Mara_mdm* et *Mara_big*. Ces résultats montrent que l'utilisation d'un ordonnancement dynamique par vol de travail permet de réduire le temps d'exécution de la section parallèle de près de 20% par rapport à une distribution statique de la charge.
D'autre part, l'utilisation de l'approche `Par_gpe` apporte dans la majorité des cas étudiés un gain de performance significatif comparé aux versions de référence. Nous notons, également, la supériorité de la version implémentant un ordonnancement par vol de travail `1-foreach-steal` par rapport à la version statique `1-foreach-static`. L'écart entre ces deux versions est fortement influencé par le déséquilibre de la charge au niveau de la répartition des groupes sur les threads ce qui se traduit par une évolution instable en dents de scie avec la variation de la taille des groupes. Par exemple, pour une exécution avec 32 threads, lorsque le nombre total de groupes est égal à 63 (cf. Figure 5.12), un seul thread parmi les 32 threads qui participent à l'exécution aura en charge l'exécution des calculs relatifs à 2 groupes ; en revanche, les 31 autres threads auront 1 seul groupe chacun. Cette répartition oblige les threads à passer à l'état inactif et attendre la fin de l'exécution du thread le plus lent. Le vol de travail permet d'atténuer cet effet de dents de scie en rééquilibrant dynamiquement la charge sur les threads dès qu'ils passent à l'état inactif. Les temps les plus bas sont obtenus lorsque la charge est équitablement répartie sur les threads participant

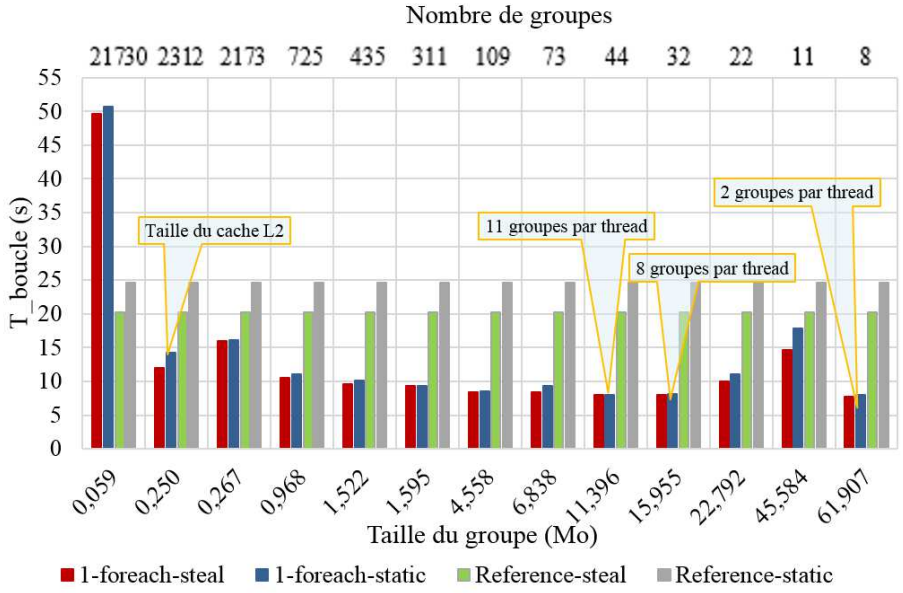


FIGURE 5.8 – Variation du temps d'exécution de la boucle élémentaire en fonction de la taille des groupes avec 4 threads pour une simulation du jeu de données MARA_big

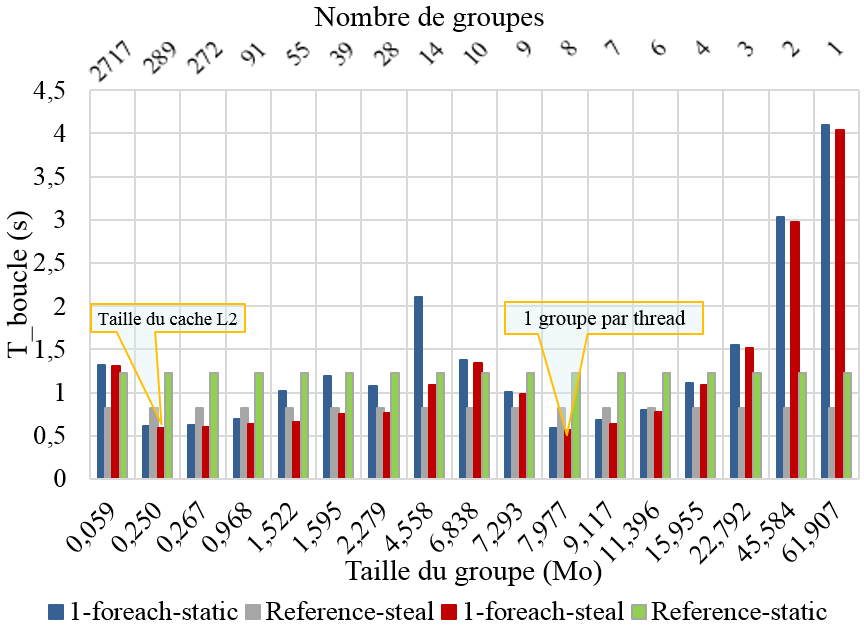


FIGURE 5.9 – Variation du temps d'exécution de la boucle élémentaire en fonction de la taille des groupes avec 8 threads pour une simulation du jeu de données MARA_mdm

à l'exécution de la boucle sur les groupes.

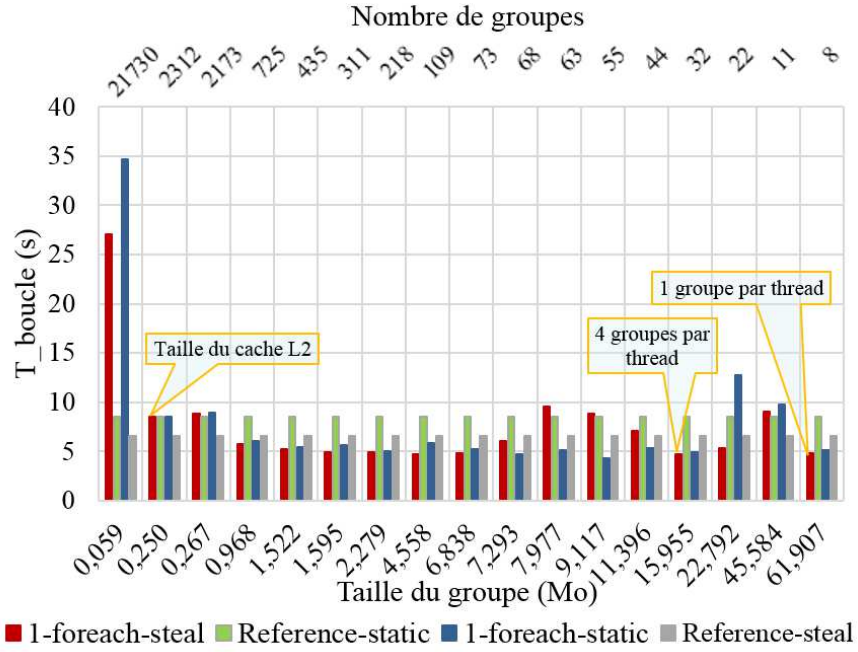


FIGURE 5.10 – Variation du temps d'exécution de la boucle élémentaire en fonction de la taille des groupes avec 8 threads pour une simulation du jeu de données MARA_big

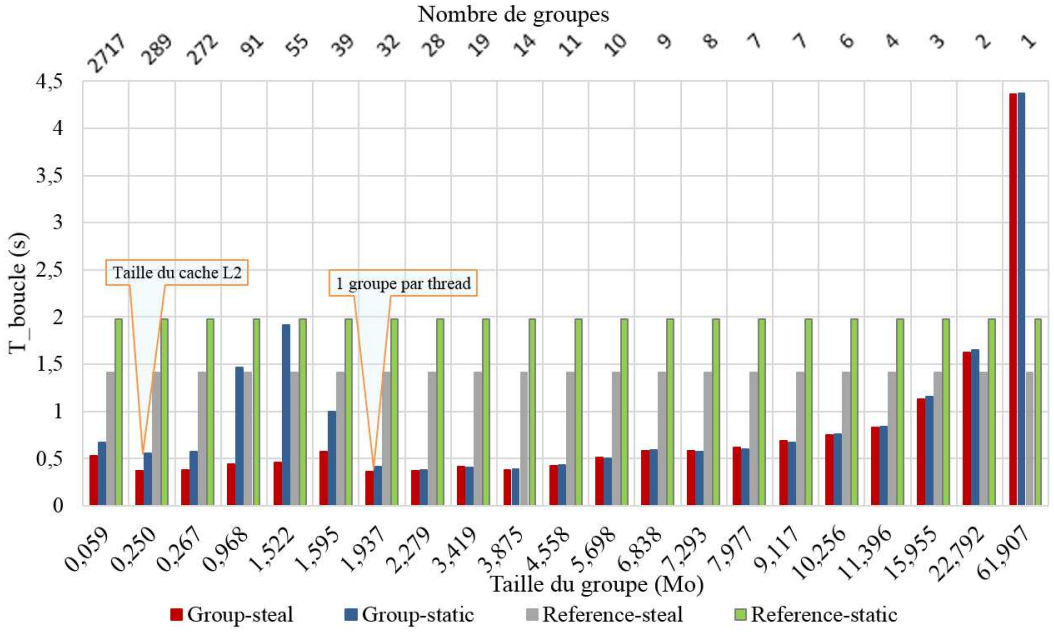


FIGURE 5.11 – Variation du temps d'exécution de la boucle élémentaire en fonction de la taille des groupes avec 32 threads pour une simulation du jeu de données MARA_mdm

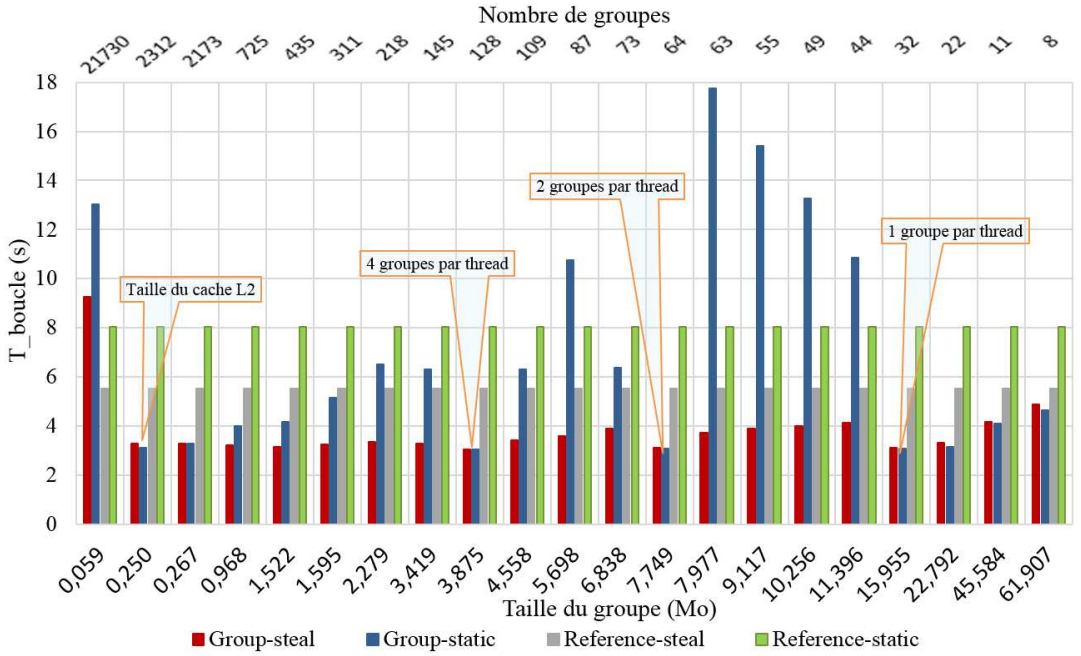


FIGURE 5.12 – Variation du temps d'exécution de la boucle élémentaire en fonction de la taille des groupes avec 32 threads pour une simulation du jeu de données MARA_big

L'augmentation linéaire de *T_boucle* à partir d'une taille qui correspond à un seul groupe par thread (Nombre de groupes = nombre de threads) résulte du manque de parallélisme pour un nombre de thread qui dépasse le nombre de groupes à exécuter. Lorsque le nombre de groupes est égal à 1 (taille de groupe = taille totale du jeu de données), nous retrouvons les temps d'exécution séquentiels (cf. Figures 5.7, 5.9 et 5.11 et la figure 5.5).

5.4.2.2 Évaluation de l'accélération

Nous retenons, à partir de cette série de tests, que les meilleures performances correspondent globalement aux versions implémentant un ordonnancement dynamique par vol de travail et que au sein de ces versions, les meilleures temps d'exécution sont obtenus pour des groupes de tailles proches de la taille du cache L2. Ainsi, fixons la taille du groupe à 0,25 Mo et nous évaluons l'accélération (*speedup*) de la boucle sur les groupes *Outer_loop* et celle de la boucle élémentaire *Inner_loop* par rapport à l'accélération de la version de `Reference`. Nous notons que l'accélération de *Outer_loop* avec 32 threads est de l'ordre de 22 pour le jeu de données *Mara_big* (cf. Figure 5.13). Elle est presque 2 fois plus élevée que l'accélération de la boucle élémentaire de la version `Reference`. L'accélération de la boucle interne *Inner_loop* est légèrement plus élevée que celle de la boucle *Outer_loop* qui comprend les copies à partir de (et vers) la mémoire RAM.

Les accélérations obtenues avec le jeu de données *Mara_mdm* (cf. Figure 5.14) conservent le classement établi pour le jeu de données *Mara_big* avec une supé-

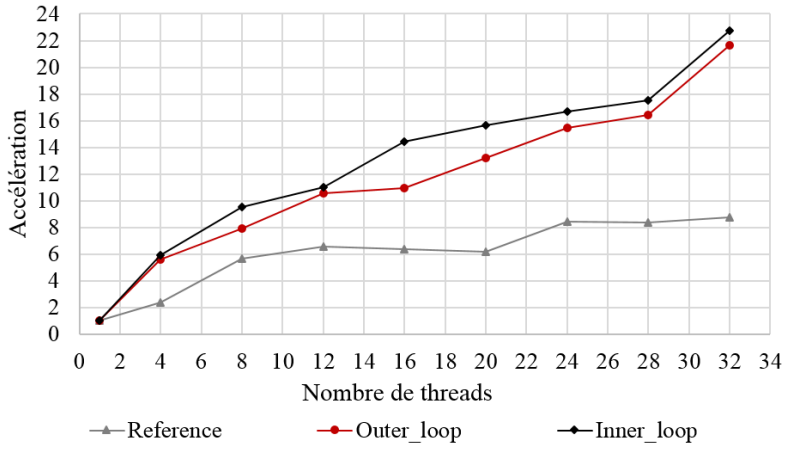


FIGURE 5.13 – Accélération de la boucle élémentaire par rapport à la version séquentielle pour une simulation avec le jeu de données MARA_big

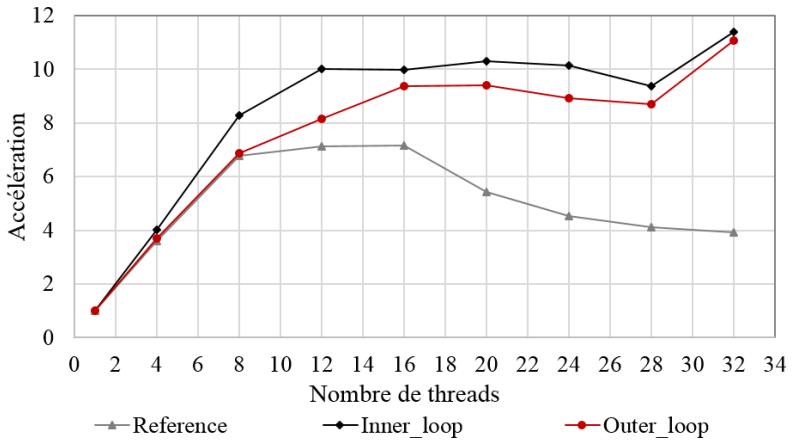


FIGURE 5.14 – Accélération de la boucle élémentaire par rapport à la version séquentielle pour une simulation avec le jeu de données MARA_mdm

riorité de la version `Par_gpe` par rapport à la version de `Reference`. En revanche, les accélérations de ce jeu de données sont moins élevées que celles que nous avons notées pour le jeu de données *Mara_big*. Cet écart entre les deux jeux de données s'explique par la taille réduite de *Mara_mdm* ce qui limite le parallélisme à extraire de la boucle parallèle *Outer_loop* et amortit par conséquent l'accélération.

5.4.2.3 Évaluation du nombre de défauts de cache

Nous comparons le nombre moyen de défauts de cache L2 commis par cœur dans les routines `Calcul` et `Calcul_elem` de la version `Par_gpe` avec celui de la version `Reference`.
Nous considérons les mêmes conditions expérimentales que les expériences précédentes et nous fixons la taille des groupes à 0,25 Mo ($\simeq$ *Taille du cache L2*). Nous traitons dans cette expérience le cas d'une exécution avec avec 32 threads des jeux

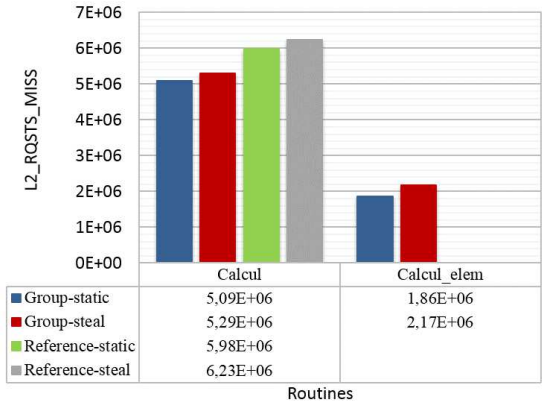


FIGURE 5.15 – Mesure du nombre de défauts de cache dans les routines de calcul du code EPX pour une exécution avec 32 threads de la simulation Mara_mdm avec une taille de groupe égale à 0,25 Mo.

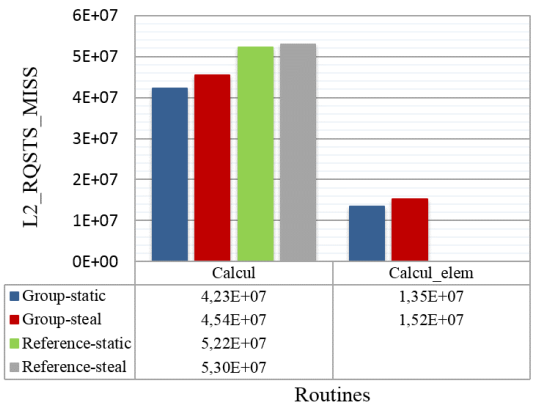


FIGURE 5.16 – Mesure du nombre de défauts de cache dans les routines de calcul du code EPX pour une exécution avec 32 threads de la simulation Mara_big avec une taille de groupe égale à 0,25 Mo.

de données *Mara_big* et *Mara_mdm*. Les nombres des défauts de cache L2 reportés dans les histogrammes des figures 5.15 et 5.16 sont obtenus par l'outil de mesure de performance LIKWID [2] [129]. Dans notre étude, nous nous intéressons à la lecture de la métrique *L2_RQSTS_MISS* qui mesure combien de fois les requêtes de données à partir du cache L2 ont engendré un défaut de cache.

L'approche `Par_gpe` permet de réduire le nombre de défauts de cache de près de 15% au niveau de la routine `Calcul` par rapport à la `Reference`. Environ 35% de ces défauts de cache sont commis au niveau de la boucle élémentaire *Inner_loop* de la routine `Calcul_elem`. Le reste de ces défauts est du à la création des groupes (dans la routine `Glob_loc`) et à la mise à jour des données élémentaires dans la structure de données d'origine (routine `Loc_glob`).

La comparaison des résultats des versions implémentant un ordonnancement statique et celles basées sur un ordonnancement par vol de travail montre une légère supériorité des versions statiques par rapport aux versions avec vol. Nous expliquons cet écart de performances par les coûts supplémentaires introduits par la gestion de la cohérence entre le cache L2 de la victime et le cache L3 suite à la migration des données lors des vols.

5.5 Conclusion

Au vu des analyses précédentes, l'utilisation de l'approche *Par_gpe* permet d'améliorer de manière significative les performances du code EPX. Les meilleures résultats sont obtenus pour des tailles de groupes proches de la taille du cache L2. Cette taille particulière donne lieu à la construction d'un nombre élevé de groupes favorisant un équilibrage dynamique de charge à grain fin sans engendrer, néanmoins, des surcoûts supplémentaires que les groupes de tailles plus petites peuvent

2. https ://code.google.com/p/likwid/

entraîner. Les performances obtenues avec des groupes de tailles plus grandes sont moins stables et fortement dépendantes du jeu de données à simuler.

Chapitre 6

Parallélisme imbriqué

Dans le présent chapitre, nous proposons d'étendre la parallélisation de notre approche `Par_gpe` *pour mieux profiter de la localité des données au niveau de chaque groupe. Nous commençons par présenter ce nouveau niveau de parallélisme. Ensuite, nous menons une étude paramétrique afin d'évaluer les performances de notre implémentation. Nous concluons ce chapitre par une discussion autours des résultats obtenus.*

Sommaire

6.1 Principe et implémentation

Dans la première implémentation parallèle que nous avons proposée dans le chapitre 5 (version *1-foreach*), les calculs relatifs aux éléments de chaque groupe se déroulaient de manière séquentielle au niveau de chaque cœur : chaque thread s'exécutant sur un cœur du processeur prend en charge l'exécution d'un groupe à partir de sa création (`Glob_loc`) jusqu'à sa destruction (`Loc_glob`).
Les meilleures performances de la version *1-foreach* ont été obtenues pour des groupes qui tiennent dans le cache L2. En revanche, pour les groupes de grandes tailles, nous avons noté une dégradation des performances due au manque de parallélisme engendré par la diminution du nombre de groupes à traiter en parallèle. Ainsi, dans cette nouvelle version parallèle, l'enjeu consiste à diminuer le temps des traitements séquentiels en accélérant davantage les calculs élémentaires à l'intérieur de chaque groupe. Pour ce faire, nous avons opté pour la parallélisation de la boucle élémentaire (lignes 8 → 10 de l'algorithme 5) (cf. Figure 6.1). Cette parallélisation consiste à faire participer une équipe de threads au niveau de chaque groupe dans l'exécution des itérations de la boucle élémentaire tout en préservant la localité des données au niveau du cache L3 partagé entre les cœurs travaillant sur le même groupe.
Dans la suite du document, nous désignons par version *2-foreach* cette deuxième implémentation parallèle (cf. Algorithme 9).

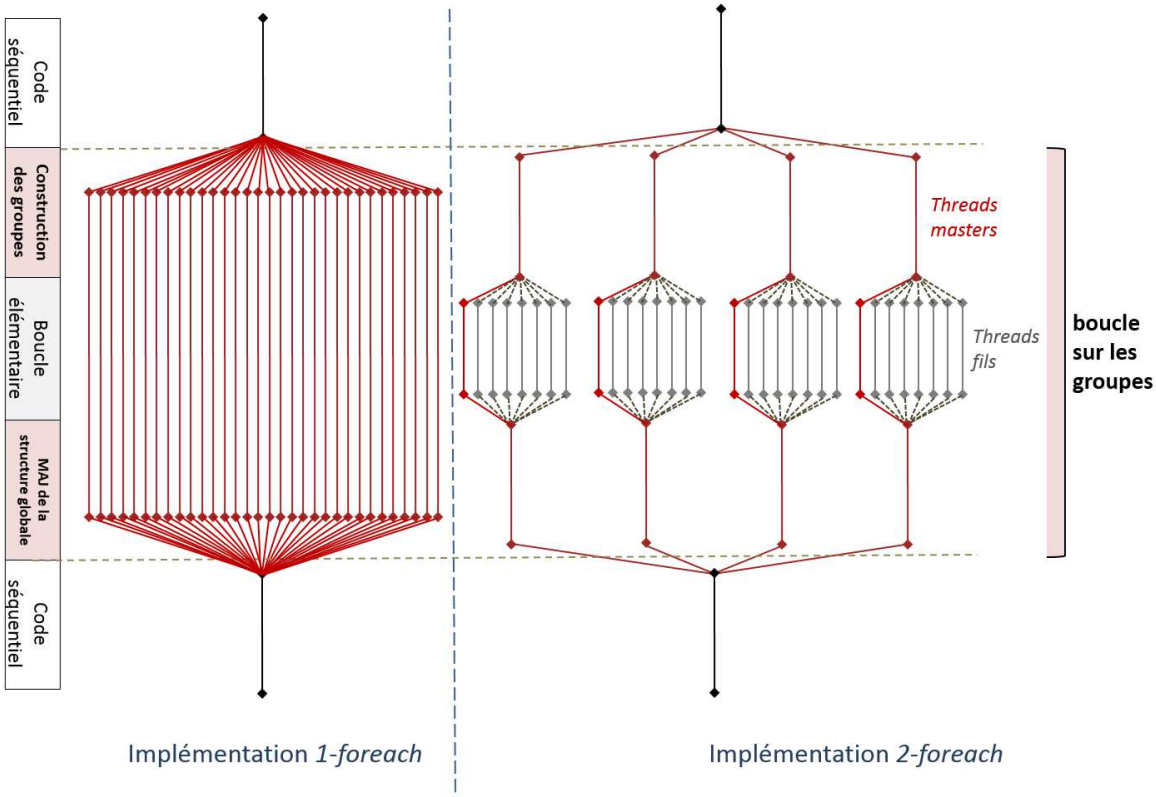


FIGURE 6.1 – Parallélisation multi-niveaux de la boucle sur les groupes

Algorithme 9 : Pseudo-code du second niveau de parallélisme

```
1  Entrées : nombre d'éléments par groupe elt_gr, nombre d'éléments total nelem, nombre
     de groupes ngr, tableaux de la structure de données globale glob
2  SUBROUTINE KAAPI_GROUPS(A)
3  !          Initialisations globales
4  !          process [Lbound, Ubound] inclusive
5               DO Loop_gr = Lbound, Ubound
6                   i_deb = 1 + elt_gr × (Loop_gr − 1)
7                   i_fin = min(i_deb + elt_gr - 1, nelem)
8                   ALLOCATE(local arrays)
9                   CALL Glob_loc (A → A')
10                  Err = KAAPIF_SET_GRAIN(...)
11                  Err = KAAPIF_SET_POLICY(...)
12                  Err = KAAPIF_SET_DISTIBUTION(...)
13                  Err = KAAPIF_FOREACH(...,
14 &                                kaapi_inner, ...)
15 !                ...
16                  CALL Loc_glob(A' ← A)
17                  DEALLOCATE (local arrays)
18 !                ...
19              ENDDO
20 END SUBROUTINE KAAPI_GROUPS
```

Pour contrôler le placement des threads qui travaillent sur le même groupe, nous configurons la variable d'environnement `OMP_PLACES` (cf. section 2.3.5.4 du chapitre 2) de manière à ce que les threads de *Outer_loop* soient attribués aux processeurs multi-cœurs du nœud du calcul.
Afin de lier les threads de *Inner_loop* aux bancs numa des processeurs considérés, nous configurons la variable d'environnement `OMP_PLACES` comme suit :

$$\texttt{OMP\_PLACES} = \texttt{"}\{\text{numa}(0)\}, \{\text{numa}(1)\}, \{\text{numa}(2)\}, \{\text{numa}(3)\}\texttt{"}$$

Chaque banc numa correspond à un domaine de localité.

6.2 Évaluation expérimentale

Dans cette partie, nous évaluons les performances de l'approche `Par_gpe` dans le cas de l'implémentation *2-foreach*.

6.2.1 Définition de la taille du grain séquentiel

6.2.1.1 Étude théorique

Lorsqu'un thread maître prend en charge l'exécution des calculs relatifs à un groupe, il est opportun que les données de ce dernier soient proche en mémoire aux données du groupe suivant qui sera traité par ce même thread. De plus, dans le cadre d'une distribution statique du travail sur les threads parallèles, l'enjeu consiste

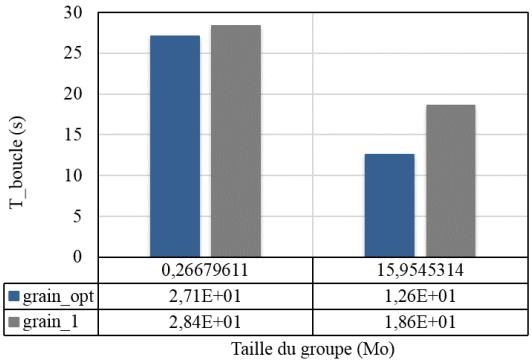


FIGURE 6.2 – Variation du temps d'exécution de la boucle élémentaire en fonction de la taille du groupe pour deux valeurs de grain séquentiel

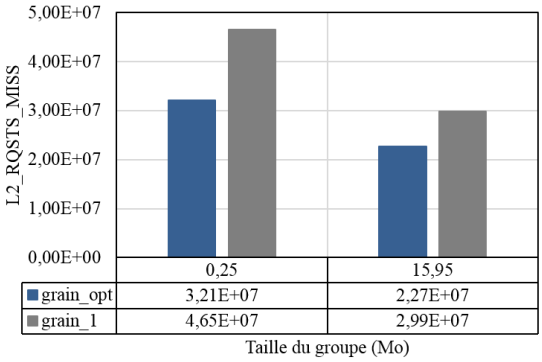


FIGURE 6.3 – Variation du nombre de défauts de cache L2 en fonction de la taille du groupe pour deux valeurs de grain séquentiel

à éviter la multiplication des coûts de création des tâches. Pour ce faire, il est nécessaire d'effectuer cette distribution de manière à ce que chaque thread ait dès le début de la section parallèle la liste des tâches à exécuter et que ces tâches utilisent des données de groupes contiguës. De ce fait, si nous considérons ngr groupes et m threads maîtres exécutant ces groupes en parallèle, alors une distribution statique optimale dans le cache de ces groupes consiste à définir le grain séquentiel de la boucle sur les groupes seq_grain_{gr} comme suit :

$$seq\_grain_{gr} = \frac{ngr}{m} \tag{6.1}$$

Dans cette implémentation *2-foreach*, il est également nécessaire de définir un grain séquentiel pour le niveau interne de parallélisme seq_grain_{elt}. Ce grain est défini en fonction du nombre de threads par équipe (f threads fils + 1 thread maître) au niveau de chaque *PLACE*. Il est calculé comme suit :

$$seq\_grain_{elt} = \frac{nb\_elt\_gr}{f+1} \tag{6.2}$$

Nous notons que cette définition des tailles des grains est applicable dans le cas d'un ordonnancement statique. Pour un ordonnancement par vol de travail, cette définition est à éviter car elle empêche l'équilibrage dynamique de charge.

6.2.1.2 Validation expérimentale

Afin d'évaluer l'impact de la taille du grain sur les performances, nous comparons le temps d'exécution de la boucle élémentaire *T_boucle* pour un grain séquentiel égal à 1 par rapport à celui obtenu avec un grain défini selon les formules 6.1 et 6.2 et ce pour les deux niveaux de boucles. Nous désignons par *grain_opt* cette taille théoriquement optimale et nous vérifions son efficacité sur le jeu de données *mara2_big*. Ce test, de part sa grande taille, nous permet de manipuler un nombre plus important de groupes de tailles comparables avec les tailles des niveaux du cache de chaque processeur.

L'exécution de *Outer_loop* est prise en charge par 4 threads maîtres distribués sur les 4 sockets disponibles (`OMP_PLACES` = "{numa(0)}, {numa(1)}, {numa(2)}, {numa(3)}"). Au niveau de chaque socket, 4 threads (3 threads fils + 1 thread maître) exécutent les itérations de *Inner_loop*. L'ordonnancement des deux niveaux parallèles est statique.
Les résultats reportés dans la figure 6.2 correspondent à la taille optimale de groupe retenue dans le chapitre 5 (0,25 Mo) et à une taille égale à 15,95 Mo qui s'approche de celle du cache L3 partagé entre les cœurs du processeur travaillant sur le même groupe de données.
Nous reportons dans l'histogramme de la figure 6.3 le nombre de défauts de cache L2 *L2_RQSTS_MISS* commis au niveau de la boucle élémentaire dans le cas d'un grain séquentiel égal à 1 comparé au nombre de défauts de cache L2 moyen par cœur commis dans cette même région du code pour un grain séquentiel égal à *grain_opt*.
À partir de la figure 6.2, nous notons une diminution du temps d'exécution dans les versions utilisant un grain séquentiel de taille *grain_opt* par rapport à l'utilisation d'un grain de taille 1. Ce gain de performance résulte de la baisse des surcoûts de création des tâches par rapport à l'utilisation d'un grain fin (*grain_1*) d'une part et de la baisse du nombre de défauts de cache L2 d'autre part comme le montrent les résultats de la figure 6.3.
Dans la suite du manuscrit, les tailles des grains des différents tests sont égales à *grain_opt*.

6.2.2 Version 2-foreach versus version 1-foreach

6.2.2.1 Accélération des calculs élémentaires

Dans cette partie, nous évaluons l'apport de la parallélisation de la boucle élémentaire *Inner_loop* à l'intérieur de la boucle sur les groupes *Outer_loop*.
Dans la version *2-foreach*, nous utilisons 16 threads parmi lesquels nous désignons 4 maîtres et 4 équipes de 4 threads pour l'exécution de *Inner_loop*. Dans la version *1-foreach*, les itérations sur les groupes sont distribuées sur les 16 threads disponibles. Nous reportons dans les tables 6.1 et 6.2 le temps moyen d'exécution d'un groupe pour les deux versions parallèles en utilisant les jeux de données *Mara_mdm* et *Mara_big* respectivement. Pour la version *1-foreach*, nous calculons le temps moyen mis par chaque thread pour effectuer les calculs relatifs à un groupe sans compter les coûts de construction de ce dernier. Les mesures effectuées pour la version *2-foreach* représentent le temps moyen mis par équipe de threads (1 thread maitre + f threads fils) pour effectuer les calculs relatifs à un groupe.

Nous notons que la parallélisation de *Inner_loop* permet de réduire le temps d'exécution moyen des itérations élémentaires au niveau d'un groupe de près de 70% par rapport à la version *1-foreach*. Dans le cas des groupes de tailles égales à 15,95 Mo, la parallélisation de *Inner_loop* nous a permis de réduire le rapport entre le temps moyen de calcul d'un groupe par la taille de ce dernier de près de 0,069 s/Mo comparé à la version *1-foreach*. Pour les groupes de tailles égales à 0,25 Mo, ce rapport est réduit de 0,1132 s/Mo par rapport à l'implémentation *1-foreach*. Cet écart entre les performances des deux tailles pour la version *2-foreach* est dû

TABLE 6.1 – Temps d'exécution des itérations élémentaires par groupe dans les versions *1-foreach* et *2-foreach* en utilisant le jeu de données *Mara_mdm*

Numéro du thread	**Temps d'exécution moyen par groupe**	
	Taille du groupe = 0,25 Mo	**Taille du groupe = 15,95 Mo**
Version *1-foreach*	0,0254	1,0127
Version *2-foreach*	0,0053	0,3989

TABLE 6.2 – Temps d'exécution des itérations élémentaires par groupe dans les versions *1-foreach* et *2-foreach* en utilisant le jeu de données *Mara_big*

Numéro du thread	**Temps d'exécution moyen par groupe**	
	Taille du groupe = 0,25 Mo	**Taille du groupe = 15,95 Mo**
Version *1-foreach*	0,0339	1,5323
Version *2-foreach*	0,0056	0,4286

au manque de parallélisme dans le cas des groupes de petites tailles (0,25 Mo) par rapport au parallélisme que l'on peut extraire à partir des groupes de grandes tailles (15,95 Mo).
Nous poursuivons notre analyse pour voir si ce gain obtenu au niveau des calculs élémentaires ne sera pas ôté par les coûts de construction et de mise à jour de groupes à l'échelle de *Outer_loop*.

6.2.2.2 Évaluation des temps d'exécution de *Outer_loop*

Dans cette série de tests, nous conservons les mêmes conditions expérimentales que la série précédente et nous étudions l'évolution des temps d'exécution de la section parallèle en fonction de la taille des groupes.
Nous reportons dans la figure 6.4 les temps écoulés dans *Outer_loop* pour une exécution avec 16 threads de la simulation *Mara_mdm*. Les temps mesurés comprennent :

- le temps maximal mis par thread pour effectuer les itérations de *Inner_loop*. C'est le temps mis par le thread fils le plus lent pour exécuter les itérations de la *Inner_loop* ;
- le temps mis par le thread maître le plus lent pour la construction des groupes (*Glob_loc*) ;
- le temps écoulé dans la mise à jour de la structure globale par le thread maître le plus lent (*Loc_glob*).

Nous comparons, dans l'histogramme de la figure 6.4, les performances de la version *2-foreach* avec celles des versions *1-foreach* et *Reference*.
Ces résultats montrent que la version *2-foreach* est moins efficace que les deux autres versions pour les différentes tailles de groupes considérées. En effet, lors de l'utilisa-

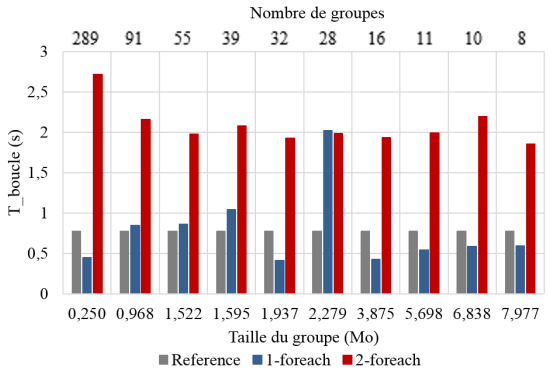
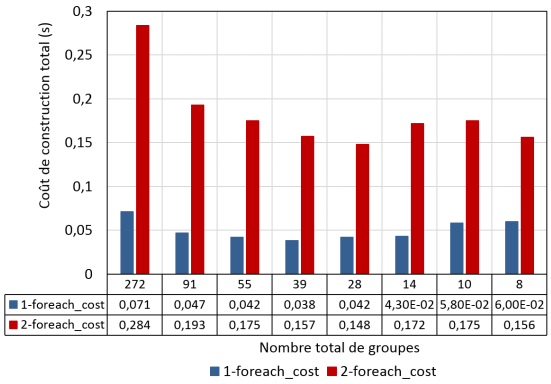


FIGURE 6.4 – Variation du temps d'exécution de la boucle élémentaire en fonction de la taille du groupe avec 16 threads pour une simulation du jeu de données Mara_mdm



	272	91	55	39	28	14	10	8
1-foreach_cost	0,071	0,047	0,042	0,038	0,042	4,30E-02	5,80E-02	6,00E-02
2-foreach_cost	0,284	0,193	0,175	0,157	0,148	0,172	0,175	0,156

FIGURE 6.5 – Variation des coûts moyens de construction des groupes pour une simulation du jeu de données Mara_mdm avec 16 threads

tion de groupes de petites tailles, le parallélisme à extraire du niveau interne (boucle sur les éléments du groupe) est moins avantageux que le parallélisme de la boucle sur les groupes ce qui amplifie inutilement les surcoûts de parallélisme des calculs élémentaires. Quant à l'écart de performance constaté pour les groupes de grandes tailles, il est dû au manque de parallélisme au niveau de *Outer_loop* induit par le nombre réduit de groupes de par leurs tailles élevées.
Nous comparons dans l'histogramme de la figure 6.5 les coûts moyens de création des groupes par thread pendant un pas de temps T_i dans la version *1-foreach* avec ceux de la version*2-foreach*. Ces valeurs représentent la moyenne des temps de création de groupes cumulés par thread maître dans la version *2-foreach*. Pour la version *1-foreach*, les temps mesurés représentent la moyenne des temps écoulés par thread (chaque thread construit le groupe sur lequel il va travailler).
Nous notons que le coût de construction des groupes dans la version *2-foreach* est de près de 4 fois plus important que celui de la version *1-foreach*. Effectivement, dans la version *2-foreach*, la construction des groupes est gérée uniquement par les 4 threads maîtres ; cependant, dans la version *1-foreach*, elle est prise en charge par les 16 threads disponibles (cf. Figure 6.1).

Pour étudier le comportement de la version *2-foreach* dans un cas qui répond au compromis nombre de groupes/taille de groupes, nous nous plaçons dans le même cadre expérimental que la série de tests précédente et nous considérons le jeu de données *Mara_big*. La taille de ce jeu de données nous permet d'avoir suffisamment de groupes de grandes tailles et par conséquent, suffisamment de parallélisme pour les deux niveaux de boucles.
Nous reportons dans l'histogramme de la figure 6.6 les performances obtenues pour cette série de tests.
Nous notons que la version *2-foreach* est moins performante que la version *1-foreach* lorsque la taille des groupes est faible (inférieure à 1,595 Mo (7000 éléments par groupe)). En effet, la lenteur de la version *2-foreach* est due au fait que nous parallélisons 4 fois moins la création des groupes par rapport à la version *1-foreach*. En

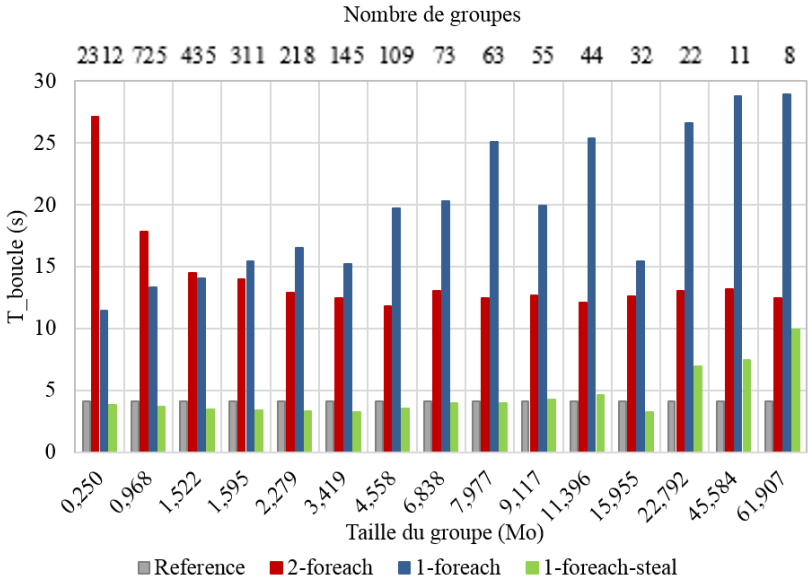


FIGURE 6.6 – Variation du temps d'exécution de la boucle élémentaire en fonction de la taille du groupe avec 16 threads pour une simulation du jeu de données Mara_big

revanche, quand le nombre de groupes diminue, la version *1-foreach* devient pénalisée par le manque de groupes : cette version a besoin de plus de parallélisme au niveau des groupes que la version *2-foreach*.

6.2.2.3 L'approche *Par_gpe* et les architectures Xeon Phi

L'exploitation du parallélisme multi-niveaux offert par la version *2-foreach* pour les groupes de grandes tailles du jeu de données *Mara_big* nécessite l'utilisation d'un grand nombre d'unités de calcul accédant à un même niveau de cache pour le traitement d'un groupe donné (itération de *Outer_loop*). Dans le cas de notre architecture multi-cœurs, les threads travaillant sur le même groupe se partagent le cache L3 à partir duquel ils chargent les données dont ils ont besoin dans leurs caches privés L2. Nous analysons dans cette partie l'influence du travail des threads sur les données du même groupe de la version *2-foreach* sur les performances de l'utilisation du cache L2 par rapport à l'implémentation *1-foreach*. Nous comparons pour les deux versions parallèles, *1-foreach* et *2-foreach*, le nombre de défauts de cache L2 moyen par cœur commis dans la routine `Calcul_elem` responsable des calculs élémentaires (cf. figure 6.7). Nous rappelons que cette routine ne comprend pas la construction des groupes. Pour la version *2-foreach*, ces mesures concernent tous les threads de l'équipe (1 thread maître et f threads fils).
La comparaison du nombre de défauts de cache de ces deux versions montre une augmentation de près de 25% de cette valeur dans la version *2-foreach* par rapport à la version *1-foreach*. En effet, lors de l'exécution des itérations de *Inner_loop*, chaque membre de l'équipe de threads fils travaillant sur le même groupe charge dans son cache L2 les données nécessaires pour le calcul des itérations élémentaires qui lui ont été attribuées. Cependant, le modèle adopté dans notre approche *Par_gpe* obéit à une organisation hybride SOAOS. Cette organisation bien qu'elle conserve la

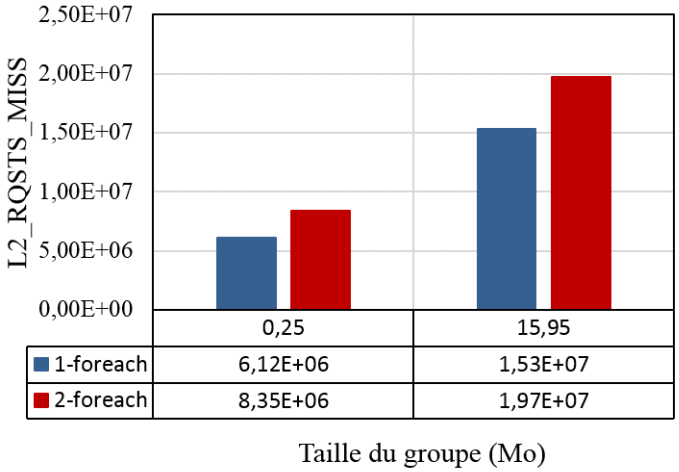


FIGURE 6.7 – Variation du nombre de défauts de cache L2 moyen par cœur au niveau de la routine élémentaire en fonction de la taille du groupe pour le jeu de données Mara_big

contigüité et l'indépendance des données pour l'ensemble des éléments d'un groupe, elle ne prend pas en compte la contigüité des données d'un élément appartenant au groupe. Ainsi, les données nécessaires pour effectuer les calculs élémentaires par thread ne forment pas un bloc contigu et elles ne sont pas indépendantes des données utilisées par les autres membres de l'équipe de threads travaillant sur ce même groupe. Cette dépendance entre les données engendre des défauts de cohérence entre les différents threads fils accédant au données d'un même groupe. La gestion de cohérence entre les niveaux de caches L2 privés par cœur pénalise l'éventuelle accélération espérée de l'implémentation *2-foreach*.
À ce sujet, les nouvelles architectures Xeon Phi de Intel disposant de plus de 60 cœurs par processeur et de près de 4 hyper-threads par cœur [57, 99] sont une alternative intéressante aux architectures multi-cœurs classiques de par l'accès direct des threads du même cœur au cache L2 de ce dernier. Ce mode de fonctionnement permet de baisser le nombre de défauts de cache de cohérence noté sur notre architecture multi-cœurs Pollux. Le portage de notre application sur ces architectures permet de profiter du parallélisme massif de l'implémentation *2-foreach* sans se confronter aux problèmes de gestion de cohérence entre les threads travaillant sur le même groupe et appartenant à des cœurs différents. Grâce au multithreading disponible sur les architectures Xeon Phi, notre approche *Par_gpe* peut être exploitée en distribuant les itérations de *Outer_loop* sur les cœurs des processeurs. Chaque groupe sera pris en charge par les threads physiques de chaque cœur. Ainsi, les threads fils de chaque itération sur les groupes accèdent directement aux données du groupes stockées dans le cache L2 de leur cœur.

6.3 Conclusion

La parallélisation de *Inner_loop* à l'intérieur des itérations de *Outer_loop* est le moyen avec lequel nous avons espéré améliorer davantage le gain de performance

réalisé dans la version *1-foreach*. Cependant, d'après notre étude expérimentale, nous avons constaté que cette implémentation est moins performante que l'implémentation *1-foreach* de l'approche `Par_gpe`. Les limites de cette version proviennent principalement de la restriction de la parallélisation de la construction des groupes au niveau des threads maîtres ; les threads fils n'y participent pas. De plus, étant donnée que l'utilisation des données d'un groupe est partagée entre les threads de l'équipe, la localité des calculs des groupes dans cette version est moins bonne que celle de la version *1-foreach*. Nous notons que, dans le cadre de cette implémentation *2-foreach*, l'utilisation d'un vol de travail pour l'équilibrage dynamique de la charge parmi les threads travaillant sur un même groupe n'aurait pas conservé la localité des données qui influe fortement les performances de notre application. L'utilisation des architectures Xeon Phi multi-cœurs semble être plus pertinente de par le grand nombre de threads physiques que l'on peut utiliser au niveau de chaque cœur et qui profitent d'un accès direct à la mémoire cache de ce dernier.

Quatrième partie

Conclusion et perspectives

Chapitre 7

Conclusion et perspectives

Les architectures parallèles actuelles intègrent au sein de leurs noeuds interconnectés des processeurs avec un nombre de cœurs à mémoire partagée de plus en plus important et répondant à des contraintes d'utilisation spécifiques, notamment en matière de gestion de la mémoire cache locale. Pour tirer parti de la puissance disponible, un parallélisme unique à mémoire distribuée, nécessaire pour gérer les communications inter-noeuds, présente le désavantage de ne pas s'adapter directement aux particularités locales des architectures multi-coeurs. De plus, les environnements de calcul à mémoire partagée proposent des outils et des techniques pour l'équilibrage de la charge entre les coeurs disponibles, qui se présentent toujours de manière plus délicate dans un contexte de mémoire distribuée à la structure plus complexe. Ainsi, des modèles de programmation tels que OpenMP, Intel TBB, Cilk et XKAAPI sont une réponse bien adaptée aux besoins logiciels et aux spécificités matérielles des architectures à mémoire partagée.

Au regard de ces problématiques, nous nous sommes intéressés, dans le cadre de ce travail, à développer une approche *Hardware-aware* qui prend en considération l'organisation hiérarchique des architectures parallèles à mémoire partagée. Notre approche `Par_gpe` offre un modèle d'optimisation de l'utilisation des espaces de stockage dans ce contexte de parallélisme.
Pour démontrer la pertinence de l'approche que nous proposons, nous l'avons implémentée dans le logiciel industriel de simulation en dynamique rapide des fluides et des structures EUROPLEXUS (EPX), en se concentrant sur le parallélisme à mémoire partagée, complémentaire d'une approche à mémoire distribuée développée et optimisée par ailleurs. De par le large panel d'applications qu'il permet de simuler, ce code est caractérisé par une structure de données très riche et des dépendances très complexes entre ses routines de calcul. Dans notre démarche d'optimisation de performances, nous nous sommes concentrés sur l'accélération de la boucle principale itérant sur les éléments du maillage. L'hétérogénéité des formulations et des matériaux des éléments pouvant co-exister dans un même modèle simulé engendre une grande variabilité entre les coûts des itérations de cette boucle.
Une première parallélisation de cette boucle avec la bibliothèque XKAAPI basée sur un ordonnancement dynamique par vol de tâches a été implémentée dans EPX. Cependant, malgré l'accélération atteinte par cette première implémentation parallèle, les performances ont été freinées par les coûts des accès fréquents et dispersés à une

structure de données complexe rendant l'exécution du code délicate à optimiser. À cause de cette structuration, une grande partie du temps d'exécution est écoulée dans des défauts de cache.

Les principaux travaux de cette thèse reposent sur la mise en place d'un modèle de structure de données assurant une meilleure localité des accès. Ce modèle, présenté dans le chapitre 5, consiste majoritairement à passer de la structure de données globale dans laquelle les champs physiques décrivant un élément donné sont stockées dans des tableaux séparés à une structure basée sur le stockage des données dans des structures indépendantes entre elles appelées *groupes*. Ces groupes sont construits à partir de la structure de données d'origine en copiant les données relatives aux calculs d'un certain nombre d'éléments du maillage dans des tableaux locaux. Ce nombre est un paramètre réglable en fonction de la taille des niveaux de la mémoire cache. Concrètement, l'implémentation de cette méthode de réorganisation des données à la volée revient à imbriquer la boucle élémentaire dans une autre boucle itérant sur des groupes locaux d'éléments. Lors de la parallélisation de cette boucle, les itérations sur les groupes sont distribuées sur les cœurs de l'architecture. Dans cette implémentation, l'exécution de la boucle interne se traite de manière séquentielle par chaque cœur. Nous avons montré dans ce document que les meilleurs résultats sont obtenus pour une taille de groupe égale à la taille du cache L2 privé par cœur. Pour cette taille particulière, l'utilisation d'un équilibrage dynamique de la charge des threads participant à l'exécution des itérations sur les groupes sous XKAAPI nous a permis de doubler l'accélération de la boucle élémentaire par rapport à une parallélisation avec XKAAPI de la version de référence du code EPX sans utilisation de l'approche `Par_gpe`.

La deuxième partie de cette thèse repose sur la parallélisation de la boucle élémentaire à l'intérieur de la boucle sur les groupes déjà parallélisée. D'après les résultats discutés dans le chapitre 6, le second niveau de parallélisme est moins performant que le parallélisme à un seul niveau que nous avons implémenté en premier. En revanche, le parallélisme imbriqué que nous avons mis en œuvre dans le code EPX pourrait être particulièrement intéressant sur les nouvelles architectures Xeon Phi de Intel qui intègrent des cœurs hyper-threadés au niveau de leurs unités de calcul. La parallélisation du niveau de boucle interne conviendrait parfaitement à la hiérarchie de ces architectures si les itérations de la boucle sur les groupes seraient traitées par les hyper-threads au niveau des cœurs.

Il est interessant de noter également que l'approche `Par_gpe` que nous avons implémentée dans le code EPX pourra être utilisée dans d'autres applications éléments finis dont l'algorithme repose sur une boucle élémentaire manipulant des structures de données de tailles importantes.

Perspectives

Pour un meilleur gain en performance avec l'approche `Par_gpe`, il est primordial de bien définir les paramètres d'exécution (type d'ordonnancement, nombre de threads, taille des groupes, ...) selon le jeu de données à simuler et en fonction des caractéristiques de l'architecture matérielle sur laquelle l'application va s'exécuter. Au jour d'aujourd'hui, dans le prototype que nous avons présenté dans ce travail

ce paramétrage est effectué manuellement par l'utilisateur. Étant donné que notre approche est dédiée à des applications industrielles de domaines très variés, il serait intéressant de mettre en œuvre une stratégie d'auto-paramétrage [81] permettant à l'utilisateur quelque soit son domaine de travail d'aboutir à la configuration optimale sans devoir pour autant être expert en HPC.

En ce qui concerne les coûts de construction des groupes discutés dans le chapitre 5, il serait intéressant d'essayer de les réduire en parallélisant les opérations de copie des grandeurs physiques pour chaque groupe (cf. Algorithme 6).

L'équilibrage dynamique de charge par vol de travail sous XKAAPI est actuellement contrôlé pour que les vols restent limités à un domaine de localité sur lequel travaillent les threads d'un même groupe. Cependant, il est nécessaire d'approfondir cette étude pour éviter la dégradation de la localité des données pour des éventuels vols inter-groupes.

Pour terminer, il est important de replacer les travaux réalisés dans cette thèse dans le cadre de l'association des stratégies parallèles à mémoires distribuée et partagée. Dans ce contexte, le modèle de calcul est décomposé en sous-domaine répartis sur les processus MPI disponibles, avec un ou plusieurs par noeuds de calcul, et l'approche proposée à mémoire partagée est mise en oeuvre à l'intérieur de chaque sous-domaine, intégrant les propriétés de l'architecture sur laquelle s'exécute le processus concerné. Dès lors, le nombre de sous-domaines (i. e. processus MPI) et le nombre de threads XKAAPI par processus deviennent des paramètres qui peuvent être introduits dans la procédure d'auto-paramétrage pour obtenir à la volée la meilleure adaptation du programme aux ressources disponibles. Ceci suppose d'intégrer au logiciel la capacité de changer de configuration d'exécution parallèle de manière transparente pour l'utilisateur. Des travaux préliminaires dans ce sens existent dans EPX et sont à étendre et consolider.

Bibliographie

[1] *Elementary fluid dynamics*. Oxford applied mathematics and computing science series, 1990.

[2] *Plexus - Notice théorique*, 03 1997.

[3] *Intel XScale (R) Core*, January 2004.

[4] Reaching petascale for advanced fluidstructure dynamics. Technical report, ANR Project, April 2013. ANR-FORM-090601-01-01.

[5] De Boelelaan A, Rob van Nieuwpoort, Rob V. Van Nieuwpoort, Jason Maassen, Jason Maassen, Gosia Wrzesinska, Gosia Wrzesinska, Thilo Kielmann, Thilo Kielmann, Thilo Kielmann, Henri E. Bal, and Henri E. Bal. Adaptive load-balancing for divide-and-conquer grid applications. *J. of Supercomputing*, 2004.

[6] Ern A. *Elements finis*. Dunod, 2005.

[7] B. Ackland, A. Anesko, D. Brinthaupt, S.J. Daubert, A. Kalavade, J. Knobloch, E. Micca, M. Moturi, C.J. Nicol, J.H. O'Neill, J. Othmer, E. Sackinger, K.J. Singh, J. Sweet, C.J. Terman, and J. Williams. A single-chip, 1.6-billion, 16-b mac/s multiprocessor dsp. *Solid-State Circuits, IEEE Journal of*, 35(3) :412–424, March 2000.

[8] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An evaluation of directory schemes for cache coherence. *SIGARCH Comput. Archit. News*, 16(2) :280–298, May 1988.

[9] SpirosN. Agathos, PanagiotisE. Hadjidoukas, and VassiliosV. Dimakopoulos. Task-based execution of nested openmp loops. In BarbaraM. Chapman, Federico Massaioli, MatthiasS. Muller, and Marco Rorro, editors, *OpenMP in a Heterogeneous World*, volume 7312 of *Lecture Notes in Computer Science*, pages 210–222. Springer Berlin Heidelberg, 2012.

[10] Hussein Al-Zoubi, Aleksandar Milenkovic, and Milena Milenkovic. Performance evaluation of cache replacement policies for the spec cpu2000 benchmark suite. In *Proceedings of the 42Nd Annual Southeast Regional Conference*, ACM-SE 42, pages 267–272, New York, NY, USA, 2004. ACM.

[11] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L Steele Jr, Sam Tobin-Hochstadt, Joao Dias, Carl Eastlund, et al. The fortress language specification. *Sun Microsystems*, 139 :140, 2005.

[12] George Almasi. Pgas (partitioned global address space) languages. In David Padua, editor, *Encyclopedia of Parallel Computing*, pages 1539–1545. Springer US, 2011.

[13] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The tera computer system. *SIGARCH Comput. Archit. News*, 18(3b) :1–6, June 1990.

[14] James Archibald and Jean-Loup Baer. Cache coherence protocols : Evaluation using a multiprocessor simulation model. *ACM Trans. Comput. Syst.*, 4(4) :273–298, September 1986.

[15] James Archibald and Jean-Loup Baer. Cache coherence protocols : Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems (TOCS)*, 4(4) :273–298, 1986.

[16] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '98, pages 119–129, New York, NY, USA, 1998. ACM.

[17] J.-L. Baer and W.-H. Wang. On the inclusion properties for multi-level cache hierarchies. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, ISCA '88, pages 73–80, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[18] Klaus-Jurgen Bathe, Ekkehard Ramm, and Edward L. Wilson. Finite element formulations for large deformation dynamic analysis. *International Journal for Numerical Methods in Engineering*, 9(2) :353–386, 1975.

[19] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Syst. J.*, 5(2) :78–101, June 1966.

[20] MichaelA. Bender and MichaelO. Rabin. Online scheduling of parallel programs on heterogeneous systems with applications to cilk. *Theory of Computing Systems*, 35(3) :289–304, 2002.

[21] Paul Besl. A case study comparing aos (arrays of structures) and soa (structures of arrays) data layouts for a compute-intensive loop run on intel xeon processors and intel xeon phi product family coprocessors, 2013.

[22] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk : An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216, New York, NY, USA, 1995. ACM.

[23] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5) :720–748, September 1999.

[24] Timothy Brecht. On the importance of parallel application placement in numa multiprocessors.

[25] Greg Breinholt and Christoph Schierz. Algorithm 781 : Generating hilbert's space-filling curve by recursion. *ACM Trans. Math. Softw.*, 24(2) :184–189, June 1998.

[26] F. Broquedis, O. Aumage, B. Goglin, S. Thibault, P.-A. Wacrenier, and R. Namyst. Structuring the execution of openmp applications for multicore architectures. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–10, April 2010.

[27] F. Warren Burton and M. Ronan Sleep. Executing functional programs on a virtual tree of processors. In *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*, FPCA '81, pages 187–194, New York, NY, USA, 1981. ACM.

[28] Arthur R. Butz. Convergence with hilbert's space filling curve. *Journal of Computer and System Sciences*, 3(2) :128 – 146, 1969.

[29] M. Castro, L.G. Fernandes, C. Pousa, J. Mehaut, and M.S. de Aguiar. Numa-ictm : A parallel version of ictm exploiting memory placement strategies for numa machines. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8, May 2009.

[30] Michel Cazenave. *Méthode des éléments finis : Approche pratique en mécanique des structures*. Dunod, 2010.

[31] David Chaiken, John Kubiatowicz, and Anant Agarwal. Limitless directories : A scalable cache coherence scheme. *SIGPLAN Not.*, 26(4) :224–234, April 1991.

[32] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, 21(3) :291–312, 2007.

[33] Dominique Chamoret. Modélisation du contact : nouvelles approches numériques. *These de doctorat, Ecole Centrale de Lyon*, 2002.

[34] 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.

[35] 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.

[36] Trishul M. Chilimbi, Bob Davidson, and James R. Larus. Cache-conscious structure definition. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pages 13–25, Atlanta, GA, May 1999. ACM.

[37] Philippe Clauss and Vincent Loechner. Parametric analysis of polyhedral iteration spaces. *Journal of VLSI signal processing systems for signal, image and video technology*, 19(2) :179–194, 1998.

[38] UPC Consortium et al. Upc language specifications v1. 2. *Lawrence Berkeley National Laboratory*, 2005.

[39] G. Contreras and M. Martonosi. Characterizing and improving the performance of intel threading building blocks. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 57–66, Sept 2008.

[40] Jean-Charles Craveur. *Modélisation des éléments finis : Cours et exercices corrigés, 3e édition*. Dunod, 3 edition, 2008.

[41] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th National Conference*, ACM '69, pages 157–172, New York, NY, USA, 1969. ACM.

[42] Elizabeth Cuthill. Several strategies for reducing the bandwidth of matrices. In DonaldJ. Rose and RalphA. Willoughby, editors, *Sparse Matrices and their Applications*, The IBM Research Symposia Series, pages 157–166. Springer US, 1972.

[43] L. Dagum and R. Menon. Openmp : an industry standard api for shared-memory programming. *Computational Science Engineering, IEEE*, 5(1) :46–55, Jan 1998.

[44] Mattias De Wael, Stefan Marr, Bruno De Fraine, Tom Van Cutsem, and Wolfgang De Meuter. Partitioned Global Address Space Languages. *ACM Computing Surveys*, page 29, January 2016.

[45] VassiliosV. Dimakopoulos, PanagiotisE. Hadjidoukas, and GiorgosCh. Philos. A microbenchmark study of openmp overheads under nested parallelism. In Rudolf Eigenmann and BronisR. de Supinski, editors, *OpenMP in a New Era of Parallelism*, volume 5004 of *Lecture Notes in Computer Science*, pages 1–12. Springer Berlin Heidelberg, 2008.

[46] James Dinan, D. Brian Larkins, P. Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. Scalable work stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 53 :1–53 :11, New York, NY, USA, 2009. ACM.

[47] U.S. Dixit and P.M. Dixit. A study on residual stresses in rolling. *International Journal of Machine Tools and Manufacture*, 37(6) :837 – 853, 1997.

[48] Laurent d'Orazio, Fabrice Jouanot, Cyril Labbé, and Claudia Roncancio. Caches sémantiques coopératifs pour la gestion de données sur grilles. *23e Journées Bases de Données Avancées (BDA'2007)*, 2007.

[49] Michel Dubois and Faye A. Briggs. Effects of cache coherency in multiprocessors. *Computers, IEEE Transactions on*, C-31(11) :1083–1099, Nov 1982.

[50] S. J. Eggers and R. H. Katz. Evaluating the performance of four snooping cache coherency protocols. *SIGARCH Comput. Archit. News*, 17(3) :2–15, April 1989.

[51] E.Allen Emerson and Vineet Kahlon. Exact and efficient verification of parameterized cache coherence protocols. In Daniel Geist and Enrico Tronci, editors, *Correct Hardware Design and Verification Methods*, volume 2860 of *Lecture Notes in Computer Science*, pages 247–262. Springer Berlin Heidelberg, 2003.

[52] E.Allen Emerson and Vineet Kahlon. Rapid parameterized model checking of snoopy cache coherence protocols. In Hubert Garavel and John Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *Lecture Notes in Computer Science*, pages 144–159. Springer Berlin Heidelberg, 2003.

[53] Vincent Danjean Fabien Le Mentec, Thierry Gautier. The x-kaapiś application programming interface. part i : Data flow programming. Technical report, Project-Teams MOAIS, 2011.

[54] Vincent Faucher. *Reduction methods for fast transient structural dynamics applicated to the analysis of complex structures under impact.* Theses, École normale supérieure de Cachan - ENS Cachan, June.

[55] Vincent Faucher. Advanced parallel strategy for strongly coupled fast transient fluid-structure dynamics with dual management of kinematic constraints. *Advances in Engineering Software*, 67 :70–89, 2014.

[56] Vincent Faucher. *Numerical methods and parallel algorithms for fast transient strongly coupled fluid-structure dynamics.* Habilitation à diriger des recherches, INSA de Lyon ; Université Claude Bernard - Lyon I, June 2014.

[57] W Feinstein and M Brylinski. Structure-based drug discovery accelerated by many-core devices. *Current drug targets*, 2016.

[58] M. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, C-21(9) :948–960, Sept 1972.

[59] M. Frigo, C.E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 285–297, 1999.

[60] 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.

[61] André Fortin ; André Garon. *Les éléments finis : de la théorie à la pratique.* 2011.

[62] T. Gautier, F. Lementec, V. Faucher, and B. Raffin. X-kaapi : A multi paradigm runtime for multicore architectures. In *Parallel Processing (ICPP), 2013 42nd International Conference on*, pages 728–735, Oct 2013.

[63] Thierry Gautier, Xavier Besseron, and Laurent Pigeon. Kaapi : A thread scheduling runtime system for data flow computations on cluster of multiprocessors. In *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation*, PASCO '07, pages 15–23, New York, NY, USA, 2007. ACM.

[64] Paul Germain. *Mecanique Tome I.* Ecole polytechnique, 1986.

[65] Paul Germain. *Mecanique Tome II.* Ecole polytechnique, 1986.

[66] M. B. Giles, G. R. Mudalige, B. Spencer, C. Bertolli, and I. Reguly. Designing op2 for gpu architectures. *J. Parallel Distrib. Comput.*, 73(11) :1451–1460, November 2013.

[67] Nicolin Govender, Daniel N. Wilke, Schalk Kok, and Rosanne Els. Development of a convex polyhedral discrete element simulation framework for {NVIDIA} kepler based {GPUs}. *Journal of Computational and Applied Mathematics*, 270 :386 – 400, 2014. Fourth International Conference on Finite Element Methods in Engineering and Sciences (FEMTEC 2013).

[68] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM JOURNAL ON APPLIED MATHEMATICS*, 17(2) :416–429, 1969.

[69] Yi Guo, Jisheng Zhao, Vincent Cave, and Vivek Sarkar. Slaw : A scalable locality-aware adaptive work-stealing scheduler for multi-core systems. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 341–342, New York, NY, USA, 2010. ACM.

[70] Panagiotis E Hadjidoukas and Vassilios V Dimakopoulos. Support and efficiency of nested parallelism in openmp implementations.

[71] Danny Hendler and Nir Shavit. Non-blocking steal-half work queues. In *Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing*, PODC '02, pages 280–289, New York, NY, USA, 2002. ACM.

[72] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[73] H.D. Hibbitt, P.V. Marcal, and J.R. Rice. A finite element formulation for problems of large strain and large displacement. *International Journal of Solids and Structures*, 6(8) :1069 – 1086, 1970.

[74] Paul N Hilfinger, Dan Oscar Bonachea, Kaushik Datta, David Gay, Susan L Graham, Benjamin Robert Liblit, Geoffrey Pike, Jimmy Zhigang Su, and Katherine A Yelick. Titanium language reference manual, version 2.19. Technical report, UC Berkeley Tech Rep. UCB/EECS-2005-15, 2005.

[75] M.D. Hill and A.J. Smith. Evaluating associativity in cpu caches. *Computers, IEEE Transactions on*, 38(12) :1612–1630, Dec 1989.

[76] Weiwu Hu, Weisong Shi, and Zhimin Tang. Jiajia : A software dsm system based on a new cache coherence protocol. In Peter Sloot, Marian Bubak, Alfons Hoekstra, and Bob Hertzberger, editors, *High-Performance Computing and Networking*, volume 1593 of *Lecture Notes in Computer Science*, pages 461–472. Springer Berlin Heidelberg, 1999.

[77] Patrick Huerre. *Mécanique des fluides Tome 1*. 1998.

[78] Alexandru C. Iordan, Magnus Jahre, and Lasse Natvig. Tuning the victim selection policy of intel {TBB}. *Journal of Systems Architecture*, pages –, 2015.

[79] Seung jai Min, Costin Iancu, and Katherine Yelick. Hierarchical work stealing on manycore clusters. In *In Fifth Conference on Partitioned Global Address Space Programming Models*, 2011.

[80] M Kandemir, A Choudhary, J Ramanujam, and P Banerjee. Optimizing spatial locality in loop nests using linear algebra. In *Proc. 7th Workshop Compilers for Parallel Computers*, volume 426, page 430. Citeseer, 1998.

[81] Thomas Karcher, Christoph Schaefer, and Victor Pankratius. Auto-tuning support for manycore applications-perspectives for operating systems and compilers.

[82] Ken Kennedy, Charles Koelbel, and Hans Zima. The rise and fall of high performance fortran : An historical object lesson. In *Proceedings of the Third*

ACM SIGPLAN Conference on History of Programming Languages, HOPL III, pages 7–1–7–22, New York, NY, USA, 2007. ACM.

[83] Richard E. Ladner, Ray Fortna, and Bao-Hoang Nguyen. A comparison of cache aware and cache oblivious static search trees using program instrumentation, 2002.

[84] Jinpil Lee and M. Sato. Implementation and performance evaluation of xcalablemp : A parallel programming language for distributed memory systems. In *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, pages 413–420, Sept 2010.

[85] Sophie Léger. *MMéthode Lagrangienne actualisée pour des problémes hyper-élastiques en trés grandes déformations*. PhD thesis, Université Laval, 2014.

[86] Arnaud Legrand and Yves Robert. *Algorithmique Parallèle – Cours Et Exercices Corrigés*. Dunod, 2003.

[87] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the dash multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90, pages 148–159, New York, NY, USA, 1990. ACM.

[88] Wei Li and Keshav Pingali. *A singular loop transformation framework based on non-singular matrices*. Springer, 1993.

[89] Calvin Lin and Lawrence Snyder. Zpl : An array sublanguage. In Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, volume 768 of *Lecture Notes in Computer Science*, pages 96–114. Springer Berlin Heidelberg, 1994.

[90] Wai-Hung Liu and Andrew H. Sherman. Comparative analysis of the cuthill-lmckee and the reverse cuthillmckee ordering algorithms for sparse matrices. *SIAM Journal on Numerical Analysis*, 13(2) :198–213, 1976.

[91] David B Loveman. High performance fortran. *Parallel & Distributed Technology : Systems & Applications, IEEE*, 1(1) :25–42, 1993.

[92] Najib Mahjoubi. *Méthode générale de couplage de schéma d'intégration multi-échelles en temps en dynamique des structures*. PhD thesis, INSA Lyon, 2010.

[93] Michael McCool, James Reinders, and Arch Robison. *Structured Parallel Programming : Patterns for Efficient Computation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[94] B. Moon, H.V. Jagadish, C. Faloutsos, and J.H. Saltz. Analysis of the clustering properties of the hilbert space-filling curve. *Knowledge and Data Engineering, IEEE Transactions on*, 13(1) :124–141, Jan 2001.

[95] Eliakim Hastings Moore. On certain crinkly curves. *Transactions of the American Mathematical Society*, 1(1) :pp. 72–90, 1900.

[96] G.E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1) :82–85, Jan 1998.

[97] Stéphanie Moreaud and Brice Goglin. Impact of NUMA Effects on High-Speed Networking with Multi-Opteron Machines. In *PDCS*, Cambridge, United States, November 2007.

[98] Paul Germain ; Patrick Muller. *Introduction a la mecanique des milieux continus*. 1980.

[99] Perri J. Needham, Ashraf Bhuiyan, and Ross C. Walker. Extension of the {AMBER} molecular dynamics software to intel's many integrated core (mic) architecture. *Computer Physics Communications*, pages –, 2016.

[100] Nathan M Newmark. A method of computation for structural dynamics. *Journal of the Engineering Mechanics Division*, 85(3) :67–94, 1959.

[101] Giap Nguyen Nguyen. *Spacer-filling curves and their application in image processing*. Theses, Université de La Rochelle, November 2013.

[102] Diego Novillo. Openmp and automatic parallelization in gcc. In *In the Proceedings of the GCC Developers*, 2006.

[103] Robert W. Numrich and John Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2) :1–31, August 1998.

[104] Marc Palyart. *A Model-Based Approach for the Development of High-Performance Scientific Computing Software*. Theses, Université Paul Sabatier - Toulouse III, December 2012.

[105] David A. Patterson and John L. Hennessy. *Computer Architecture : A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[106] Chuck Pheatt. Intel&reg ; threading building blocks. *J. Comput. Sci. Coll.*, 23(4) :298–298, April 2008.

[107] Sabine Philippe. *Development of an Arbitrary Lagrangian Eulerian (ALE) formulation for the 3D simulation of flat rolling*. Theses, École Nationale Supérieure des Mines de Paris, June 2009.

[108] Laércio L Pilla, Christiane Pousa Ribeiro, Daniel Cordeiro, Abhinav Bhatele, Philippe OA Navaux, Jean-François Méhaut, and Laxmikant V Kalé. Improving parallel system performance with a numa-aware load balancer. 2011.

[109] Jean-Noel Quintin. *Dynamic Load-Balancing on Hierarchical Platforms*. Theses, Université de Grenoble, December 2011.

[110] Damien Leone ; Jean-Noel Quintin ; Bruno Raffin. History based work-stealing for dynamic numerical simulations. Technical report, LIG/INRIA MOAIS Team, 2011.

[111] Keith H. Randall. *Cilk : Efficient Multithreaded Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1998.

[112] James Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007.

[113] C.P. Ribeiro, J. Mehaut, A. Carissimi, M. Castro, and L.G. Fernandes. Memory affinity for hierarchical shared memory multiprocessors. In *Computer Architecture and High Performance Computing, 2009. SBAC-PAD '09. 21st International Symposium on*, pages 59–66, Oct 2009.

[114] Arch Robison, Michael Voss, and Alexay Kukanov. Optimization via Reflection on Work Stealing in TBB.

[115] Hans Sagan. *Space-filling curves*. Springer Science & Business Media, 2012.

[116] I. J. Schoenberg. On the peano curve of lebesgue. *Bull. Amer. Math. Soc.*, 44(8) :519, 08 1938.

[117] James E. Smith and James R. Goodman. A study of instruction cache organizations and replacement policies. *SIGARCH Comput. Archit. News*, 11(3) :132–137, June 1983.

[118] William Stallings. *Organisation et architecture de l'ordinateur*. Imp. la source d'or, 2003.

[119] Childress Stephen. *An Introduction to theroritical fluid machanics*. Americain Mathematical Society, 2000.

[120] G. Strang and G. J. Fix. An analysis of the finite element method. 1973.

[121] Taeweon Suh, Douglas M. Blough, and Hsien hsin S. Lee. Supporting cache coherence in heterogeneous multiprocessor systems. Technical report, in Proceedings of the Design Automation and Test in Europe (DATE, 2003.

[122] V. S. Sunderam. Pvm : A framework for parallel distributed computing. *Concurrency : Pract. Exper.*, 2(4) :315–339, November 1990.

[123] Yoshizumi Tanaka, Kenjiro Taura, Mitsuhisa Sato, and Akinori Yonezawa. Performance evaluation of openmp applications with nested parallelism. In Sandhya Dwarkadas, editor, *Languages, Compilers, and Run-Time Systems for Scalable Computers*, volume 1915 of *Lecture Notes in Computer Science*, pages 100–112. Springer Berlin Heidelberg, 2000.

[124] Marc Tchiboukdjian. *Algorithmes parallèle efficace en Cache : Application à la visualisation scientifique*. PhD thesis, université de Grenoble, 2010.

[125] Vincent Faucher Bruno Raffin Thierry Gautier, Fabien Lementec. X-kaapi : a multi paradigm runtime for multicore architectures. Technical report, INRIA, 2012.

[126] Xinmin Tian, Jay P. Hoeflinger, Grant Haab, Yen-Kuang Chen, Milind Girkar, and Sanjiv Shah. A compiler for exploiting nested parallelism in openmp programs. *Parallel Comput.*, 31(10-12) :960–983, October 2005.

[127] Ashkan Tousimojarad and Wim Vanderbauwhede. Steal locally, share globally. *International Journal of Parallel Programming*, 43(5) :894–917, 2015.

[128] Daouda Traoré. *Self-adaptive parallel algorithms and applications*. Theses, Institut National Polytechnique de Grenoble - INPG, December 2008.

[129] Jan Treibig, Georg Hager, and Gerhard Wellein. Likwid : A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of the 2010 39th International Conference on Parallel Processing Workshops*, ICPPW '10, pages 207–216, Washington, DC, USA, 2010. IEEE Computer Society.

[130] John von Neumann. Introduction to the first draft report on the edvac. Technical report, United States Army Ordnance Department and the University of Pennsylvania, 1945.

[131] I. Wald. Fast construction of sah bvhs on the intel many integrated core (mic) architecture. *Visualization and Computer Graphics, IEEE Transactions on*, 18(1) :47–57, Jan 2012.

[132] David W. Walker, David W. Walker, Jack J. Dongarra, and Jack J. Dongarra. Mpi : A standard message passing interface. *Supercomputer*, 12 :56–68, 1996.

[133] M.V. Wilkes. Slave memories and dynamic storage allocation. *Electronic Computers, IEEE Transactions on*, EC-14(2) :270–271, April 1965.

[134] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. pages 30–44, 1991.

[135] Michael Joseph Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[136] I-Chen Wu and H.T. Kung. Communication complexity for parallel divide-and-conquer. In *Foundations of Computer Science, 1991. Proceedings., 32nd Annual Symposium on*, pages 151–162, Oct 1991.