Simulation de la dynamique des dislocations à très grande échelle

Arnaud Etcheverry

université de BORDEAUX

THÈSE

PRÉSENTÉ À

L'UNIVERSITÉ DE BORDEAUX I

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

Par **Arnaud ETCHEVERRY**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Simulation de la dynamique des dislocations à très grande échelle

---

Soutenue le : 23 novembre 2015

Après avis des rapporteurs :

François-Xavier Roux	Professeur Paris VI
Marc Fivel	Directeur de recherche, CNRS

Devant la commission d'examen composée de :

Gerard L. Vignoles	Prof. Univ. Bordeaux I ...	Président du jury
François-Xavier Roux	Professeur Paris VI	Examinateur
Marc Fivel	Directeur de recherche, CNRS	Examinateur
Laurent Dupuy ...	Ingénieur de recherche CEA .	Examinateur
Laurent Colombet .	Chargé de recherche CEA ..	Examinateur
Olivier Coulaud ..	Directeur de recherche, INRIA	Directeur de thèse

– 2015 –

Remerciements

Pour ces travaux de thèse, je tiens tout d'abord à remercier mes encadrants pour m'avoir fait confiance. Je remercie donc tout particulièrement Olivier Coulaud, pour nos discussions et ces conseils afin d'orienter et faire avancer cette thèse. Je remercie également Laurent Dupuy, pour sa patience mais surtout sa passion en me transmettant ces connaissances pour appréhender le monde merveilleux et complexe des dislocations. Merci à tous les deux, pour m'avoir aidé à dérouler la bobine des dislocations, bien emmêlée à mon arrivée, pour m'avoir soutenu et redonné confiance dans les moments de doutes, et m'avoir initié à la recherche scientifique.
Ensuite, un grand merci aux copains, parce que ces trois années de thèse ont été trois années de bonheur grâce à eux. Toute l'équipe HiePacs, avec les indéboulonnables du premier jour, Stojce et Mawussi, et aussi Bérenger et Matthias pour des belles courses et les plans igloo de dernière minute ! Puis la relève avec JM, Pierre et Maria ! Julien permanent du baby foot, et tous les autres vrais permanents notamment Orel et Abdou, pour m'avoir fait découvrir une approche différente du sport... Merci à la love coloc, pour tous ces bons moments partagés. Merci Marlounette pour m'avoir mis la pression pour finir ce manuscrit et remplir ma vie de beaux projets !
Enfin merci à la famille, d'avoir fait l'effort d'écrire sur un tableau noir mon sujet de thèse pendant 3 ans rien que pour pouvoir articuler chaque terme. Merci de m'avoir soutenu, encouragé et fait confiance pour faire cette thèse mais aussi dans toutes mes décisions.
Ces travaux de thèse se terminent, avec un peu de nostalgie de ces belles années pleines d'apprentissages, de voyages et de rencontres mais les projets sont déjà là pour se tourner plein d'excitation vers l'avenir.

Simulation de la dynamique des dislocations à très grande échelle

Résumé :

Le travail réalisé durant cette thèse vise à offrir à un code de simulation en dynamique des dislocations les composantes essentielles pour permettre le passage à l'échelle sur les calculateurs modernes.

Nous abordons plusieurs aspects de la simulation numérique avec tout d'abord des considérations algorithmiques. Pour permettre de réaliser des simulations efficaces en terme de complexité algorithmique pour des grandes simulations, nous explorons les contraintes des différentes étapes de la simulation en offrant une analyse et des améliorations aux algorithmes.

Ensuite, une considération particulière est apportée aux structures de données. En prenant en compte les nouveaux algorithmes, nous proposons une structure de données pour bénéficier d'accès performants à travers la hiérarchie mémoire. Cette structure est modulaire pour faire face à deux types d'algorithmes, avec d'un côté la gestion du maillage nécessitant une gestion dynamique de la mémoire et de l'autre les phases de calcul intensifs avec des accès rapides. Pour cela cette structure modulaire est complétée par un octree pour gérer la décomposition de domaine et aussi les algorithmes hiérarchiques comme le calcul du champ de contrainte et la détection des collisions.

Enfin nous présentons les aspects parallèles du code. Pour cela nous introduisons une approche hybride, avec un parallélisme à grain fin à base de threads, et un parallélisme à gros grain de type MPI nécessitant une décomposition de domaine et un équilibrage de charge.

Finalement, ces contributions sont testées pour valider les apports pour la simulation numérique. Deux cas d'étude sont présentés pour observer et analyser le comportement des différentes briques de la simulation. Tout d'abord une simulation extrêmement dynamique, composée de sources de Frank-Read dans un cristal de zirconium est utilisée, avant de présenter quelques résultats sur une simulation cible contenant une forte densité de défauts d'irradiation.

Mots clés :

dynamique des dislocations, problème à N-corps, structure de données, hiérarchie mémoire, méthode multipôle rapide, parallélisme hybride, mémoire partagée, OpenMP, mémoire distribuée, MPI, scalabilité

Discipline :

Informatique

Hybrid parallelism on large scale dislocation dynamic simulation

Abstract :

This research work focuses on bringing performances in 3D dislocation dynamics simulation, to run efficiently on modern computers.

First of all, we introduce some algorithmic technics, to reduce the complexity in order to target large scale simulations.

Second of all, we focus on data structure to take into account both memory hierachie and algorithmic data access. On one side we build this adaptive data structure to handle dynamism of data and on the other side we use an Octree to combine hierachie decompostion and data locality in order to face intensive arithmetics with force field computation and collision detection.

Finnaly, we introduce some parallel aspects of our simulation. We propose a classical hybrid parallelism, with task based openMP threads and domain decomposition technics for MPI.

Keywords :

simulation, dislocation dynamics, 3D, n-body problem, data structure, cache efficient, memory hierarchie, fast multipol method, parallelism,hybrid, openMP task, shared memory, distributed memory, MPI, scalability

Discipline :

Computer science

Table des matières

Liste des Algorithmes

Liste des tableaux

Table des figures

Introduction générale

La production d'électricité par Réacteur à Eau Pressurisée (REP) doit être extrêmement contrôlée, afin d'éviter tout accident et risque de contamination. Nous pouvons observer dans la Figure 1, trois éléments critiques faisant barrière entre le combustible et le milieu extérieur : l'enceinte en béton, la cuve en acier et dans le cœur du réacteur, les gaines en alliage de Zirconium des assemblages de combustible. La compréhension et l'analyse du vieillissement de ces pièces sont donc capitales pour assurer la sûreté de l'installation.

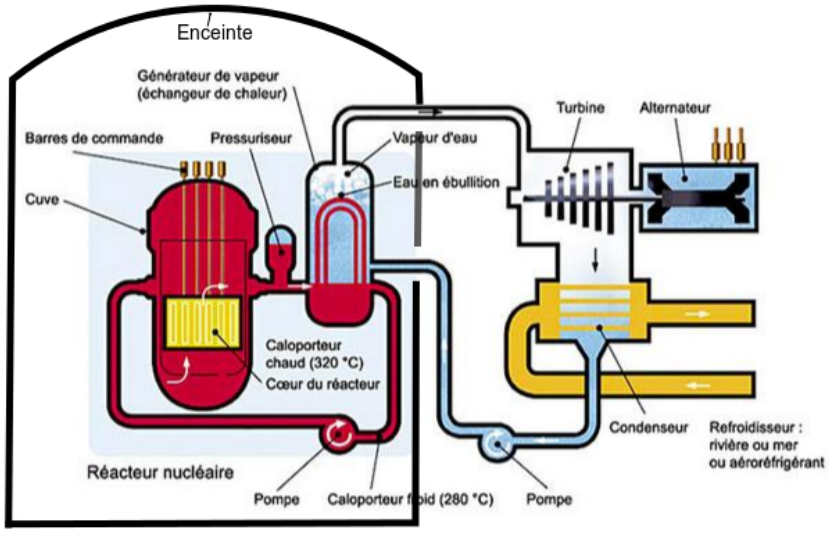


FIGURE 1 – Représentation d'un REP. (*Source : CEA*)

Soumis à un environnement extrême avec une forte pression ($\backsimeq\ 150Bar$), une température élevée ($\backsimeq\ 320°C$) et un flux neutronique important, les propriétés mécaniques des matériaux évoluent progressivement au cours de la vie des assemblages ou du réacteur. On observe ainsi, généralement, une hausse de la limite d'élasticité accompagnée d'une perte de ductilité comme le montre la Figure 2. On peut aussi, selon la nature du matériau, observer une fragilisation ou une localisation de la déformation, qui peuvent toutes les deux conduirent à la rupture du composant.

La compréhension du vieillissement de ces matériaux est donc au cœur des activités du Département des Matériaux du Nucléaire du CEA Saclay, et passe par la modélisation des mécanismes physiques impliqués. Cette démarche est intrinsèquement multi-échelle puisqu'il s'agit de comprendre en quoi l'endommagement primaire, causé par les colli-

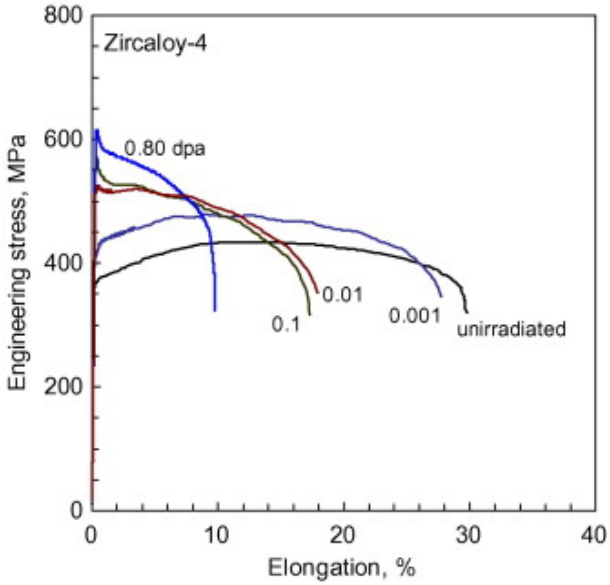


FIGURE 2 – Courbes de traction d'un alliage de zirconium industriel (Zircalloy-4) après différents niveaux d'exposition aux neutrons [41]. On observe une hausse de la limite d'élasticité de près de 200 MPa entre l'état non irradié et l'état fortement irradié ($0.8dpa$) mais aussi une diminution importante de l'allongement à rupture du matériau. Le *dpa*, ou déplacement par atome, est une estimation du nombre de fois où chaque atome du matériau a été déplacé de sa position de référence sous l'effet du flux de neutrons.

sions successives des neutrons avec des atomes (chaque collision s'effectuant à l'échelle du nanomètre sur quelques nanosecondes), va modifier le comportement mécanique d'un composant de plusieurs mètres après plusieurs années de fonctionnement.

La mise en oeuvre de cette démarche multi-échelle nécessite le chainage de nombreux codes, qui vont de l'échelle atomique avec des codes ab initio à l'échelle macroscopique avec des codes éléments finis. La simulation de la Dynamique des Dislocations (DD), qui est l'objet de cette thèse, en est un constituant. Elle a pour objet l'étude du comportement collectif des défauts cristallins que sont les dislocations, dont l'évolution au cours du temps contrôle la plupart des propriétés mécaniques du matériau.

La DD se situe, comme le montre la Figure 3, entre la Dynamique Moléculaire (DM) et les simulations de plasticité cristalline traitées par éléments finis ou méthodes FFT. De la DM sont extraites des informations comme par exemple, les lois de mobilité des dislocations individuelles, tandis que la DD a pour objectif de fournir des lois de comportement à l'échelle du monocristal ($\backsimeq 10\mu m$) pouvant être utilisées dans des codes de plasticité cristalline impliquant plusieurs milliers de cristaux ($\backsimeq 1mm$).

L'étude du comportement des dislocations est une composante importante de la métallurgie physique, dont l'acte de naissance remonte à l'année 1934 avec les travaux publiés séparément par Orowan [80], Polanyi [85] et Taylor [99] (voir Figure 4). Outre une large panoplie d'essais mécaniques, elle s'appuie sur l'observation directe des dislocations rendue possible par l'utilisation de la Microscopie Electronique en Transmission (MET) comme le montre la Figure 5, et qui a permis à Hirsch, Horne et Whelan de confirmer leur existence

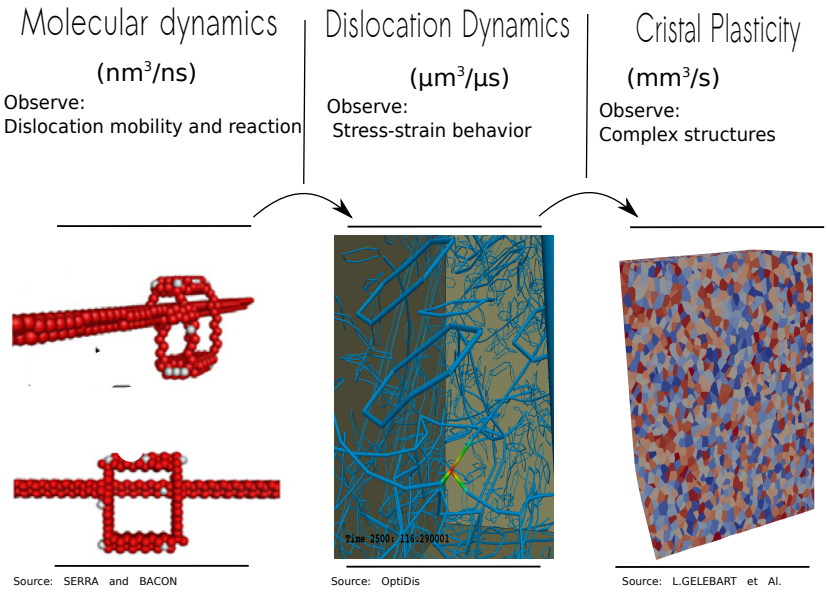


FIGURE 3 – Schéma de la modélisation multi-échelle pour les matériaux : différentes échelles caractéristiques de temps et d'espace pour différentes études.

en 1956 [58]. Si l'importance du rôle des dislocations est reconnu, la complexité de leurs comportements individuel et collectif n'ont pas permis à ce jour de modéliser avec succès le comportement mécanique des matériaux dans leur ensemble [67] et constitue un défi majeur des simulations par DD tant du point de vue physique que numérique.

Une analyse rapide permet de mesurer l'ampleur du défi numérique à relever pour simuler la déformation d'un grain cubique de $10\mu m$ de côté sur environ $100\mu s$, ce qui correspond à des grandeurs physiques standards :

Prenons tout d'abord une densité typique de dislocations ρ de $10^{14}m/m^{-3}$, qui correspond à une longueur totale de $0,1m$ dans notre grain de $10\mu m$. La longueur de discrétisation nécessaire pour discrétiser ces dislocations peut être estimer à un dixième de la distance moyenne [1] entre les dislocations soit $10^{-8}m$, ce qui donne au final 10^7 segments dans notre grain hors irradiation.

La prise en compte des boucles induites par l'irradiation, observées dans le fer, les aciers austénitiques ou les alliages de zirconium, augmente le nombre de segments puisque ces boucles sont des dislocations à part entière, qu'elles sont nombreuses (de 10^{21} à $10^{23}m^{-3}$) et de petite taille (de $1nm$ à $10nm$) ce qui nécessite de réduire d'autant plus la longueur de discrétisation. Le rajout d'une densité moyenne de 10^{22} boucles par mètre cube de taille $5nm$ et une discrétisation de longueur $1nm$, impose alors de simuler le comportement de près de 3.10^8 segments. Le pas de temps étant de l'ordre de $0,1ns$, 10^6 itérations sont également nécessaires pour atteindre le temps de simulation visé. Au cours de chaque itération, les segments de dislocations interagissent mutuellement entre eux et longue distance (via un algorithme de complexité quadratique) et présentent un aspect extrêmement dynamique du point de vue de leur connectivité.

1. Pour une densité de dislocation ρ, la distance moyenne entre les dislocations est de l'ordre de $1/\sqrt{\rho}$.

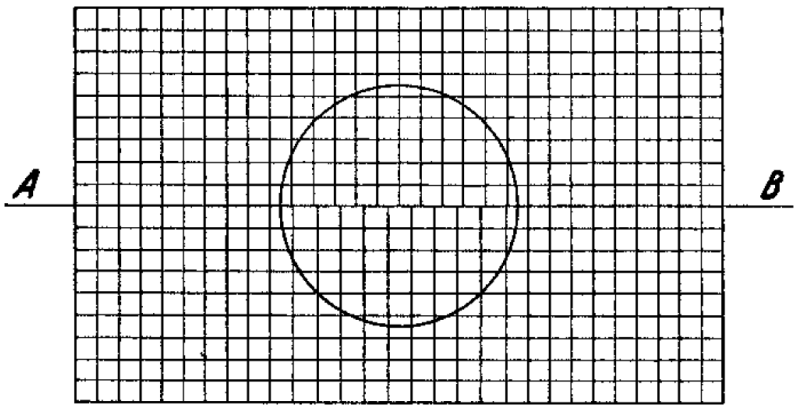


FIGURE 4 – Première représentation par Polanyi en 1934 de la déformation d'un réseau cristallin cubique en présence d'une ligne de dislocation de type *coin* vue de face [85]. Ce n'est que cinq années plus tard que Burgers [19] proposera une analyse géométrique complète de ces défauts en les caractérisant par un vecteur qui porte désormais son nom.

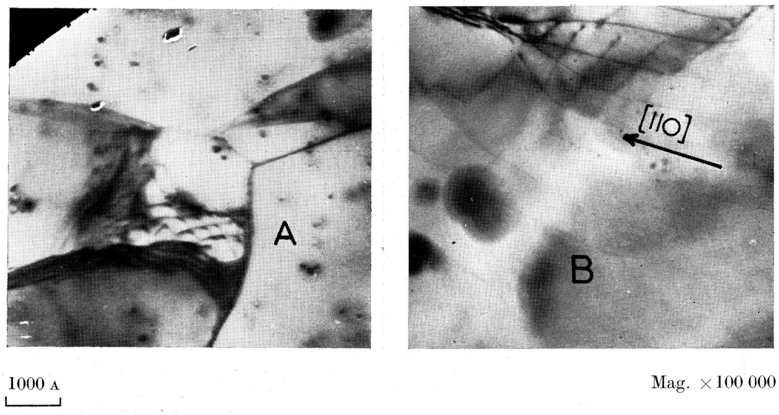


FIGURE 5 – Premiers clichés d'un réseau de dislocations effectués par Hirsch, Horne et Whelan en 1956 [58]

Ces éléments justifient pleinement le recours au calcul à hautes performances, mais restent hors d'atteinte à l'heure actuelle des codes de DD les plus avancés [8, 104]. Ce challenge a donc amené l'INRIA, le CNRS et le CEA à s'associer autour du projet OptiDis, financé par l'Agence Nationale de la Recherche et dans lequel s'inscrit cette thèse.

Première partie

Simulation en dynamique des dislocations

Nous allons, dans un premier temps, aborder les bases de la théorie des dislocations pour comprendre les mécanismes à simuler. A partir de là, nous présenterons les étapes nécessaires à la réalisation d'une simulation en dynamique des dislocations avec les défis algorithmiques associés. Nous aborderons ensuite les architectures des calculateurs modernes en terme de gestion mémoire et de parallélisme. Nous introduirons enfin les codes de simulation existants, avec leurs capacités et leurs limitations pour placer ces travaux de recherche dans un contexte de simulation hautes performances.

1 Théorie des dislocations

1.1 Défauts cristallins

Un matériau est dit cristallin lorsque sa structure présente un caractère périodique et ordonné à l'échelle atomique. Ces matériaux sont composés d'une maille élémentaire 6 et sa répétition par translation dans toutes les directions définit son réseau cristallin. Les trois structures cristallines les plus fréquemment rencontrées pour dans les matériaux de structure du nucléaire sont le réseau cubiqué centré (BCC *Body Centered Cubic*), le réseau cubique à face centrée (FCC *Face Centered Cubic*) ainsi que le réseau hexagonal (HCP *Hexagonal Close Packed*). Le premier est constitutif de l'acier de la cuve du réacteur, le second des aciers austénitiques qui maintiennent les assemblages à l'intérieur du coeur, et le troisième des assemblages en alliages de zirconium. Chaque grain, c'est à dire chaque cristal, a sa propre orientation et une taille de l'ordre de quelques microns. Le matériau est alors l'agrégation de ces grains.

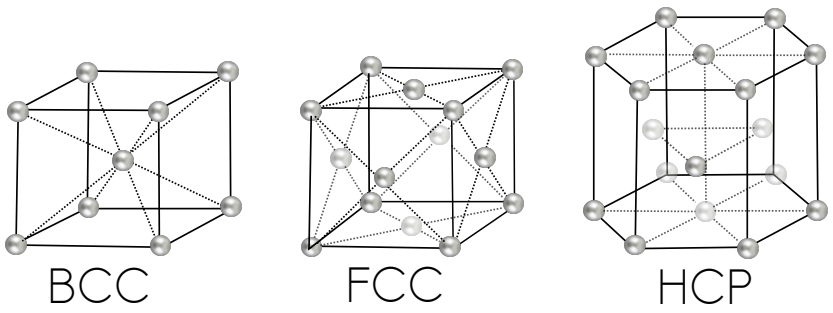


FIGURE 6 – Exemple de trois types de mailles élémentaires.

Les propriétés physico-chimiques des matériaux dépendent fortement de la structure cristalline, mais aussi des *défauts* qu'ils contiennent [60]. Ces défauts peuvent être :

- *ponctuels* comme des lacunes (absence d'un atome sur un site du réseau), des interstitiels (un atome en excès par rapport au réseau parfait) voire des atomes d'un autre type comme dans le cas des alliages ;
- *linéaires* comme c'est le cas des dislocations ;
- *surfaciques* comme les surfaces des grains ;
- *volumiques* comme par exemple des précipités de carbure dans les aciers.

Il faut noter que la présence de tels défauts est non seulement inévitable du fait des modes d'élaboration des matériaux, voire des lois de la thermodynamique, mais elle est fréquemment cherchée par les métallurgistes pour renforcer certaines propriétés mécaniques comme, par exemple la limite d'élasticité. Les dislocations jouent un rôle central dans ce tableau, car ce sont leurs interactions mutuelles ou avec les autres types de défauts qui contrôlent le comportement mécanique d'un matériau. La modélisation de ces phénomènes est précisément l'enjeu de la DD.

1.2 Dislocations

Les dislocations sont des défauts linéaires qui correspondent à une discontinuité du réseau cristallin comme illustré dans la représentation proposée par Polanyi en 1934 (voir 4). Elles se caractérisent par leur vecteur de Burgers $\vec{b}$ [19] qui mesure l'amplitude de cette discontinuité selon la méthode décrite dans la Figure 7. Le vecteur de Burgers s'apparente à l'intensité dans un circuit électrique puisque (i) il se conserve tout au long d'une ligne de dislocation, (ii) il change de sens dès lors que le sens arbitraire de parcours de la ligne est inversé et (iii) il suit la loi des nœuds en s'additionnant lorsque plusieurs dislocations s'intersectent. De ce fait, une ligne de dislocation ne peut s'arrêter à l'intérieur d'un grain, elle doit soit sortir à la surface du grain ou former avec d'autres lignes de dislocation un circuit fermé.

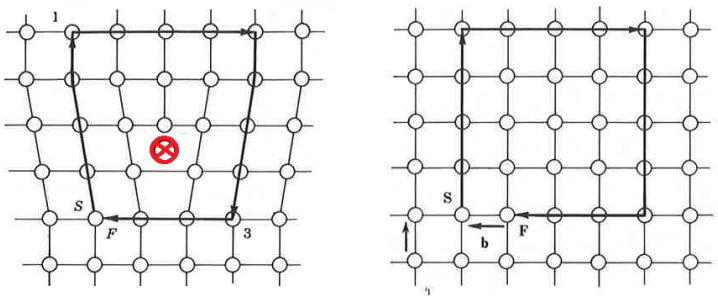


FIGURE 7 – Mise en évidence du vecteur de Burgers avec une dislocation coin sur un circuit fermé de Volterra. Nous suivons dans ce travail la convention *FS/RH* utilisée dans l'ouvrage de Hirth et Lothe [59]. Après avoir orienté arbitrairement la dislocation, un circuit suivant la *règle de la main droite* (RH = right-hand) est choisi autour de celle-ci (figure de gauche). Effectuant ce même circuit dans la configuration de référence (figure de droite), le vecteur de Burgers est alors défini comme le vecteur reliant les points F (F = finish) à S (S = start).

Muni d'un vecteur de Burgers $\vec{b}$ et du vecteur $\vec{\xi}$ tangeant à la dislocation, on peut définir *localement* le caractère de la dislocations en trois catégories :

- lorsque $\vec{b}$ et $\vec{\xi}$ sont orthogonaux, on dit que la dislocation présente un caractère *coin*,
- lorsque $\vec{b}$ et $\vec{\xi}$ sont colinéaires, on dit que la dislocation présente un caractère *vis*,
- dans les autres cas on parle de caractère mixte.

Une dislocation étant la plupart du temps courbées et de formes complexe (voir 5), son caractère de la dislocation évolue le long de sa ligne.

La présence d'une dislocation, caractérisée par son vecteur de Burgers et sa géométrie, introduit une déformation du réseau cristallin (voir Figure 8) et induit, en conséquence, un champ de contrainte σ. Les champs de déformation et de contrainte peuvent être modélisés avec précision dans le cadre de l'élasticité linéaire 0.2.2. Cette analyse montre que le champ de contrainte est à longue portée, puisqu'il décroit en $1/r$ autour de la dislocation, et qu'il dépend fortement du caractère de la dislocation. L'état de contrainte en tout point du matériau est donc la somme de la contribution individuelle de toutes les dislocations qu'il contient, appelée *contrainte interne* et de la contrainte extérieure appliquée.

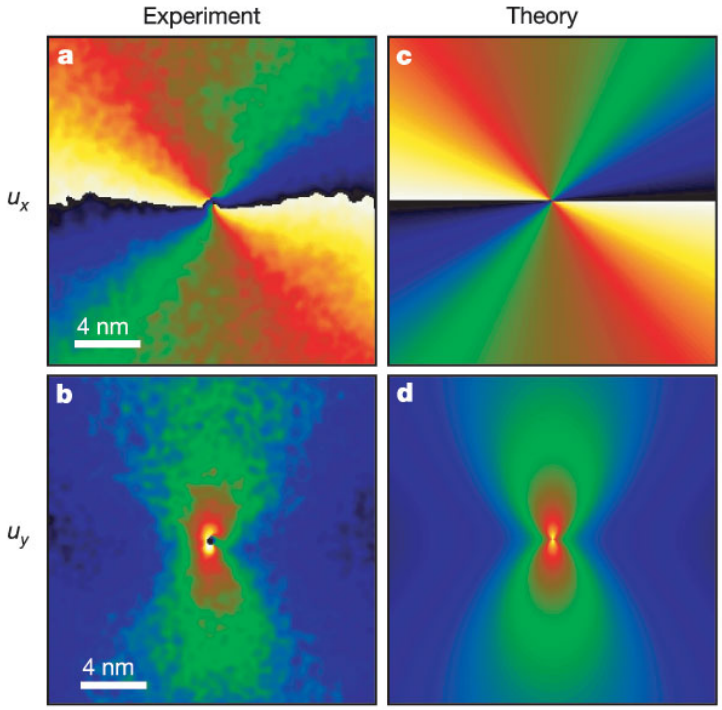


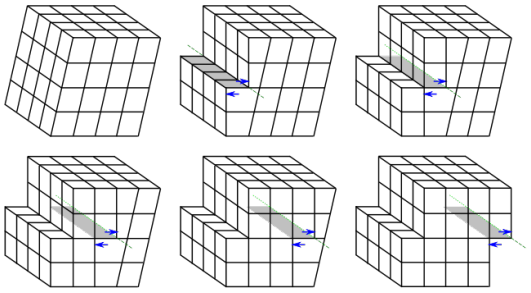
FIGURE 8 – Visualisation du champ de déplacement $u(u_x, u_y)$ autour d'une dislocation vis. Le champ de déplacement mesuré expérimentalement (à gauche) est comparé au champ calculé (à droite) dans le cadre de la théorie élastique [62].

Sous l'effet de la contrainte, chaque élément de dislocation ressent alors une force par unité de longueur que Peach et Koehler ont théorisé en 1950 [82] et qui porte désormais leur nom.

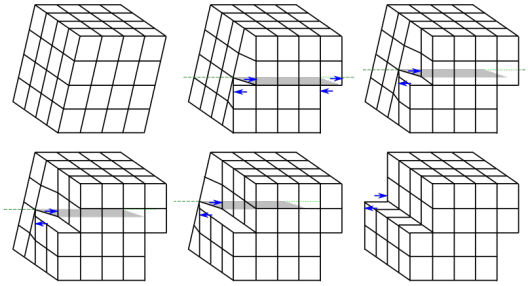
$$\vec{f}^{PK}(x) = [\sigma(x).\vec{b}] \wedge \vec{\xi}$$

Cette force est susceptible de mettre les dislocations en mouvement comme indiqué aux Figures 9a et 9b, et provoquer ainsi une déformation *plastique* irréversible du matériau dont l'amplitude et la direction correspondent aux vecteurs de Burgers de chaque dislocation. Le type de mouvement et la vitesse de déplacement des dislocations dépendent de nombreux facteurs intrinsèques comme leur vecteur de Burgers, leur caractère, mais aussi de la nature du matériau ou de la température. Le lien unissant ces paramètres à la vitesse reste à ce jour un domaine de recherche important pour les physiciens et les métallurgistes. Il est toutefois possible de distinguer deux régimes :

— un régime de *glissement* dans lequel les dislocations se déplacent dans un plan contenant à la fois leur vecteur de Burgers $\vec{b}$ et leur vecteur ligne $\vec{\xi}$. On parle alors de *plan de glissement*. Le glissement est le mode de déplacement privilégié par les dislocations notamment aux températures basses et modérées.
— un régime de *montée* dans lequel le mouvement se fait orthogonalement au plan de glissement, mais qui nécessite pour se faire un flux d'atomes interstitiels ou de lacunes vers la dislocation. Ce dernier étant thermiquement activé, le régime de montée n'est observé qu'aux températures élevées (typiquement au-dessus de $0.5T_f$ où T_f est la température de fusion du matériau) et ne sera donc pas pris en compte dans notre étude.



(a) Représentation du glissement d'une dislocation coin.



(b) Représentation du glissement d'une dislocation vis.

Le mouvement des dislocations donne lieu à une très grande variété de mécanismes physiques, étudiés depuis de nombreuses années dans le cadre de la théorie des dislocations (le lecteur peut se référer au livre de Bacon et Hull [60] pour une description complète de ces mécanismes), que les simulations par DD doivent prendre en compte. Parmi les principales interactions, notons les phénomènes de multiplication, d'annihilation, de formation et destruction de jonctions ou encore l'empilement des dislocations aux joints de grains ou autour des précipités. Le mécanisme de multiplication le plus emblématique est celui proposé par Frank et Read en 1950 [47] qui est détaillé dans la figure 10. Pour que ce phénomène opère, il faut que deux points B et C d'une ligne de dislocation soient ancrés (1) comme par exemple sur des précipités dans le grain (non représentés ici). Sous l'effet de la contrainte appliquée, la ligne commence alors à former un arc en réponse à cette force. Plus la ligne se courbe, plus la courbure augmente jusqu'à atteindre un maximum (2). A ce moment, le rayon vaut $\frac{d}{2}$ et la contrainte appliquée est maximale, $\tau \backsimeq \frac{2\mu b}{d}$, avec μ le module d'élasticité du matériau et b l'amplitude du vecteur de Burgers. Si la contrainte subie dépasse cette valeur critique, la ligne continue d'elle même à s'arquer (3). Finalement, les deux bras s'enroulent autour des points ancrés B et C jusqu'à entrer en collision (4). Cette collision impliquant des éléments de ligne de même vecteur de Burgers, mais de direction opposé, ceux-ci s'annihilent laissant à la fois une nouvelle boucle et reformant la source de Frank-Read initiale (5).

Les propriétés mécaniques d'un grain dépendent de la combinaison des nombreux mécanismes d'interaction entre dislocations qui s'y déroulent. En les prenant en compte de façon explicite, des simulations de DD doivent permettre d'en faire une analyse statistique, d'identifier le ou les mécanismes prépondérants et ainsi de prédire quantitativement la

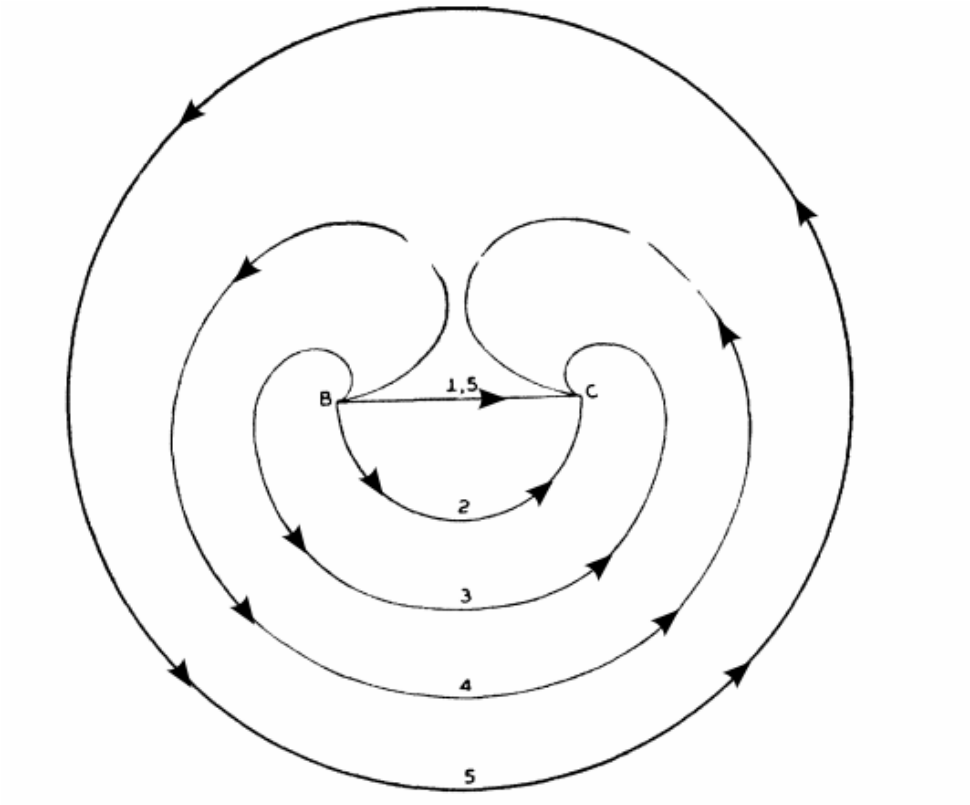
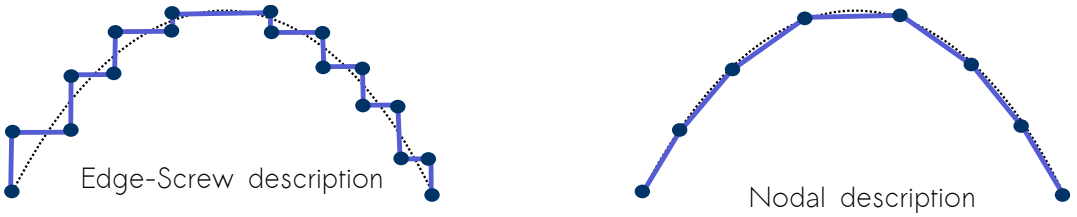


FIGURE 10 – Première représentation du mécanisme de multiplication des dislocations imaginé par Frank et Read en 1950 [47].

déformation plastique d'un matériau. Les principaux aspects théoriques sur la dislocations ayant été présentés, nous allons aborder dans la section suivante les principales étapes algorithmiques nécessaires pour réaliser des simulations de DD.

2 Les étapes algorithmiques

En fonction de la discrétisation des lignes de dislocations deux modèles de simulations se dégagent. On trouve tout d'abord l'approche vis-coin (Figure 11a) où les nœuds sont disposés uniquement sur des points du réseau du cristal et donc les segments sont soit vis soit coin ; et l'approche nodale (Figure 11b) où les segments peuvent avoir une orientation quelconque pour permettre notamment de mieux représenter la courbure des lignes.



(a) Discrétisation vis/coin.

(b) Discrétisation nodale.

FIGURE 11 – Discrétisation d'une ligne de dislocation.

Bien que nous nous concentrons dans ces travaux sur l'approche nodale, pour ces deux types de discrétisation, nous avons sur un pas de temps les mêmes étapes algorithmiques. La Figure 12 présente les différentes étapes avec le calcul des forces, le calcul des vitesses, les règles locales de collision, le remaillage puis la gestion des conditions aux bords.

Nous allons maintenant détailler ces différentes étapes, en commençant par introduire la méthode de discrétisation nodale.

2.1 Discrétisation spatiale

Une discrétisation nodale [8] propose une représentation des lignes de dislocation par des segments droits avec une orientation quelconque. Il s'agit d'une discrétisation de type éléments finis 1D de type P1 dans un espace 3D. Nous avons une continuité de ligne par les interconnexions mais une discontinuité tangentielle au niveau des nœuds. Comme représenté sur la Figure 13, l'interconnexion des segments par des nœuds, forme une ligne brisée approchant la courbure de la ligne de dislocation réelle.

Les extrémités d'un segment sont soit un nœud de discrétisation, soit un nœud physique (voir Figure 13). Un nœud physique, de manière générale, est un point de jonction entre plusieurs lignes. Il connecte donc des segments aux propriétés différentes. Il est le résultat d'une collision, et par la suite d'une jonction, entre plusieurs lignes. Suite à ces différentes réactions topologiques ces nœuds peuvent connecter de 2 à N segments. Pour simuler certains phénomènes, un nœud peut être ancré arbitrairement et n'être connecté qu'à un segment (nœuds B et C dans la Figure 10). Ce type de nœud ne peut pas exister dans un vrai matériau, mais son intérêt pratique et pédagogique est indéniable

Dans une simulation de DD, les degrés de liberté de chaque nœud du maillage est l'intersection des degrés de liberté de chacun des segments qui leur sont attachés. Ainsi, un nœud de discrétisation attaché à deux segments glissiles aura deux degrés de liberté situés dans le plan de glissement commun, tandis qu'un nœud topologique attaché à deux

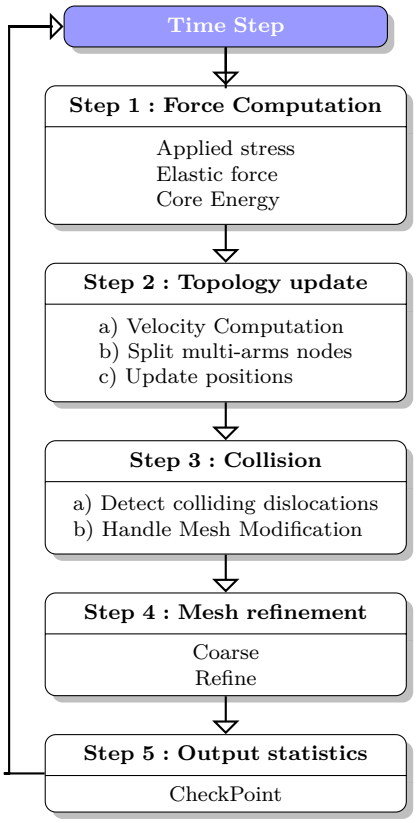


FIGURE 12 – Principales étapes d'un pas de temps.

segments sessiles [2] n'aura aucun degrés de liberé. Le nombre de degrés de liberté d'un nœud physique varie quant à lui de zero à deux.

Comme nous l'avons indiqué précédemment 0.1.2, La conservation du vecteur de Burgers est valable sur tous les nœuds [3]. C'est à dire que pour tout nœud N_i connecté à n segments ayant un vecteur de Burgers orienté de N_i vers N_j noté b_{ij}, nous avons :

$$\sum_{j=0}^{n} \vec{b}_{ij} = \vec{0}.$$

2.2 Calcul de la force nodale

Le calcul de force s'exerçant sur chaque nœud du maillage est généralement effectué dans le cadre de la théorie élastique linéaire *isotrope*. Cette force peut être calculée comme la somme de deux contributions [15]. La première est la projection de la force élastique de Peach-Koehler qui s'exerce le long de chaque segment, tandis que la deuxième dérive de l'énergie de coeur des dislocations que seules les méthodes atomistiques permettent d'estimer [74].

La force élastique de Peach-Koehler, s'exerçant sur chaque dislocation, découle de la contrainte locale σ, qui est la somme de deux contributions. On trouve d'un côté la

2. sessile = immobile
3. à l'exception des nœuds *arbitrairement* ancrés.

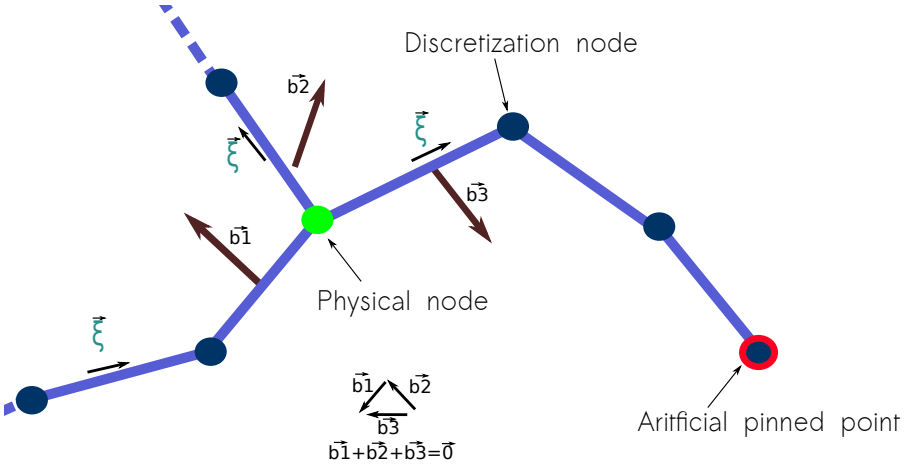


FIGURE 13 – Représentation de la topologie d'une dislocation.

contrainte appliquée ou extérieure σ_{ext} et la contrainte interne σ_{int} générée par la présence des segments de dislocations du grain. On a ainsi $\sigma = (\sigma_{ext}+\sigma_{int})$. Le calcul de la contrainte interne est basé sur la formulation de Mura [77] qui exprime la contrainte σ_{int} en tout point x sur une ligne fermée, par l'intégrale :

$$\sigma_{ij}(x) = C_{ijkl} \oint \epsilon_{lnh} C_{pqmn} G_{kp,q(x-x')} b_m dx'_h$$

où C_{ijkl} est le tenseur des modules élastiques du matériau et G la fonction de Green qui dans le cadre de la théorie isotrope s'écrie :

$$G_{ij}(x-x') = \frac{1}{8\pi\mu}[\delta_{ij}\partial_p\partial_p R - \frac{1}{2(1-\nu)}\partial_i\partial_j R]$$

où μ est le module de cisaillement isotrope et ν le coefficient de poisson et $R = ||x-x'||$.

La contrainte diverge lorsque R tend vers 0, ce qui est le cas de deux segments proches voire connectés. Ceci n'est bien entendu pas satisfaisant du point de vue physique et illustre la difficulté, voire l'impossibilité des méthodes continues à décrire une structure atomique discrète. Ceci n'est également pas satisfaisant du point de vue numérique. Une première solution, classique, consiste à tronquer l'intégration lorsque R est inférieur à une certaine distance de la dislocation, appelée rayon de cœur, et à utiliser la formule corrective de Brown [14], proche d'un modèle de tension de ligne [67], pour remplacer les interactions tronquées. Une deuxième solution consiste, dans le formalisme élastique, à "étaler" la discontinuité de déplacement mesurée par le vecteur de Burgers, sur une certaine distance autour de la dislocation, en essayant de donner à cet étalement une certaine motivation physique. Peierls [84] et Nabarro [78] font ici figures de pionniers. Le formalisme "non-singulier" proposé par Cai dans [20] s'inscrit dans cette continuité et consiste à remplacer R par $R_a = \sqrt{R^2 + a^2}$ dans les formules précédentes. S'il ne présente pas plus d'intérêt du point de vue physique que ces prédécessuers, le formalisme de Cai est toutefois bien plus adapté aux exigences du calcul numérique et permettant d'obtenir une expression analytique non seulement pour la force de Peach-Koehler s'exerçant sur

chaque segment, mais aussi pour sa projection sur les nœuds [8]. Nous adopterons pour ces raisons le formalisme de Cai dans la suite de nos travaux.

La seconde force nodale dérive de l'énergie en excès des atomes proches du coeur de la dislocation. Selon différents calculs atomistiques, cette énergie représente quelques pourcents (typiquement 10%) de l'énergie par unité de longueur d'une dislocation [61, 67]. Si les modèles élastiques "non-singuliers" précédents tentent d'intégrer une partie de cette énergie de coeur, ils échouent de l'aveu même de Cai et Bulatov [15] à la prendre totalement en compte. Les calculs atomiques permettant d'étalonner ces énergies et forces de coeur sont toutefois encore peu nombreuses [4], aussi, suivant les travaux de Shi [96], nous avons choisi de prendre cette énergie proportionnelle à l'énergie élastique d'une dislocation rectiligne infinie par unité de longueur via un coefficient de proportionnalité α_{core} de quelques pourcents.

La complexité du calcul des forces est principalement due au calcul du champ élastique, généré par les dislocations entre elles. Le calcul de la force générée par la contrainte appliquée et la contribution de l'énergie de cœur ont une complexité linéaire. Par contre le calcul d'interaction entre deux segments pour un système à N segments est de complexité quadratique, $\mathcal{O}(N^2)$. Par conséquent, ce coût devient vite prohibitif pour réaliser des simulations sur un échantillon plus important de dislocations .

Pour y faire face, comme introduit dans [8], la communauté s'est tournée vers les solutions développées pour les simulations à N-corps, notamment en astrophysique avec les forces d'interaction gravitationnelle. La méthode des multipôles rapide, (*Fast Multipole Methode* FMM) introduite par Greegard et Rokhlin [55], est utilisée pour prendre en compte la totalité des interactions du système. Il s'agit d'une méthode hiérarchique qui repose sur une séparation du champ proche et du champ lointain. Pour cela, dans les cellules les plus proches chaque paire d'interaction est calculée directement tandis que pour le champ lointain des méthodes d'approximation sont employées comme nous y reviendrons dans la section 1.1. La méthode des multipôles rapides réduit l'algorithme à une complexité linéaire $\mathcal{O}(N)$.

2.3 Calcul de la vitesse

Pour le calcul des vitesses, de nombreuses lois de mobilités peuvent être écrites prenant en compte les spécificités des matériaux considérés (voir [5] pour une revue assez exhaustive des mécanismes physiques impliqués et des lois de mobilité qui en découlent). La grande majorité des simulations par DD effectuées jusqu'à présent, repose toutefois sur une loi de glissement visqueuse Newtonienne. Cette loi de mobilité simple et linéaire relie la vitesse $v(x)$ d'un élément de dislocation à la force de Peach-Koehler $F^{pk}(x)$ qui s'y applique par la formule :

$$\vec{v}(x) = \frac{\vec{F}^{pk}(x) \cdot \vec{s}(x)}{B} \vec{s}(x) \,,$$

4. Les travaux de Martinez [74] font ici office de rare exception à cette règle

avec B le coefficient de viscosité du matériau et $\vec{s}(x) = \vec{n} \wedge \vec{\xi}$ le vecteur unitaire normal à la ligne dans le plan de glissement n de la ligne.

Le calcul de la vitesse, sur chacun des degrés de liberté utilise un formalisme éléments finis avec des objets linéiques 1D et donc une fonction de forme 1D pour interpoler de manière linéaire la vitesse de déplacement le long d'un segment à partir de la vitesse aux nœuds.

Dans le cadre d'un régime linéaire visqueux amorti, les forces motrices et les forces visqueuses le long de la dislocation sont à l'équilibre :

$$\vec{F}^{driving}(\vec{x}) + \vec{F}^{drag}(\vec{x}) = \vec{0}.$$

La force visqueuse $\vec{F}^{drag}$ s'exprime dans le cas linéaire comme suit :

$$\vec{F}^{drag}(\vec{x}) = -B\vec{v}(\vec{x}).$$

De sorte que l'équilibre des forces vérifie la formulation faible suivante en tout nœud i :

$$\vec{F}_i^{driving} = \oint_{lines} N_i(\vec{x}) * \vec{F}^{drag}(\vec{x}) dl(\vec{x})$$

où $\vec{F}_i^{driving}$ est la force effective au nœud i calculée précédemment. On définit la matrice de viscosité globale B_{ij} avec les fonctions de forme N_i définies seulement entre les nœuds i et j connectés par un segment :

$$B_{ij} = \oint_{lines} N_i(\vec{x}) * B(\vec{x}) N_j(\vec{x}) dl(\vec{x}).$$

Pour déterminer la vitesse en chaque nœud i, on obtient finalement le système d'équations linéaires suivant :

$$\sum_j B_{ij} * \vec{v}_j = \vec{F}_i^{driving}$$

avec j les nœuds connectés au nœud i.

L'assemblage de toutes ces contributions élémentaires conduit à la résolution d'un système linéaire où B_{ij} est une matrice creuse, majoritairement bande, dans la mesure où seuls les termes B_{ij} reliant deux nœuds i et j connectés sont non nuls. Ce système linéaire global pour tout le réseau de dislocation doit être assemblé à chaque itération et croît en fonction du nombre de degrés de liberté.

2.4 Mise à jour des positions

Tout au long de la simulation, il faut mettre à jour les positions des nœuds $r_i(t)$ à partir du champ de vitesse v_i^t que l'on vient de calculer. La méthode la plus simple et la moins coûteuse en calcul est d'utiliser le schéma d'intégration explicite de type Euler :

$$\vec{r}_i(t + \delta t) = \vec{r}_i(t) + \vec{v}_i(t)\delta t.$$

Cette méthode ne demande qu'un calcul de force mais nécessite par contre une forte condition de stabilité. En effet, la vitesse de déplacement d'une dislocation peut aller

jusqu'à plusieurs centaines de mètres par seconde, pour des objets de la taille de quelques nanomètres, ce qui contraint à utiliser de petits pas de temps.

Pour permettre d'augmenter le pas de temps, Bulatov et Cai [8], propose une méthode semi-implicite avec un prédicteur et un correcteur. Le prédicteur reste le schéma explicite d'Euler qui prédit la position $\vec{r}_i^P(t+\delta t)$. Ensuite, un nouveau calcul du champ de vitesse $\vec{v}_i^P(t+\delta t)$ est effectué à partir des positions nouvellement calculées. Puis la position est calculée à nouveau par une formulation implicite :

$$\vec{r}_i(t+\delta t) = \vec{r}_i(t) + \frac{\vec{v}_i^t + \vec{v}_i^P(t+\delta t)}{2}\delta t$$

On évalue alors la variation entre le prédicteur et le correcteur pour ajuster la taille du pas de temps avec $max||\vec{r}_i(t+\delta t) - \vec{r}_i^P(t+\delta t)||$ (cf. [8]).

L'utilisation d'un schéma implicite ne peut être considérée, que si le gain sur la longueur du pas de temps, compense le coût de devoir recalculer le champ de force. Pour les schémas décrits précédemment, la possibilité de maintenir la stabilité est dépendante de la taille des objets simulés, de la contrainte appliquée et surtout de la fréquence des opérations discrètes (collisions, séparations, remaillage) qui rompent la continuité du processus. Différentes nouvelles méthodes implicites sont proposées [98, 52] et une attention particulière mérite d'être portée à ces méthodes qui permettent d'augmenter le pas de temps et donc d'accélérer la simulation en réduisant le nombre d'itérations.

2.5 Opérations discrètes

Les étapes de modification de la topologie du maillage s'inscrivent dans un processus physique continu. En représentant, les mécanismes en jeu, sous la forme d'un automate cellulaire comme dans la figure 14, nous voyons que le schéma principal de la simulation est une boucle entre un calcul de force/vitesse et le déplacement des nœuds sur un pas de temps δt.

Cependant, il faut intégrer des opérations discrètes supplémentaires pour capturer, d'une part la physique du contact des dislocations, mais aussi pour éviter des artefacts numériques, notamment avec la détection des collisions et les opérations de remaillage.

Ainsi, une itération se termine par la gestion de toutes ces règles topologiques. Ces opérations discrètes se déroulent sous la contrainte d'introduire le moins de biais possible dans la recherche de la topologie minimisant l'énergie élastique du système. Pour cela trois opérations sont nécessaires et au final, vont entraîner l'ajout et/ou la suppression de degrés de liberté dans le domaine de simulation :

- gestion de la séparation des nœuds physiques ;
- gestion des collisions entre les dislocations et avec la microstructure ;
- gestion du remaillage pour garder une discrétisation cohérente après le déplacement des nœuds.

2.5.1 Scission des nœuds physiques

La scission des nœuds physiques est une étape qui ne concerne que les nœuds physiques avec au moins trois connexions [15, 96]. Cette étape survient consécutivement à

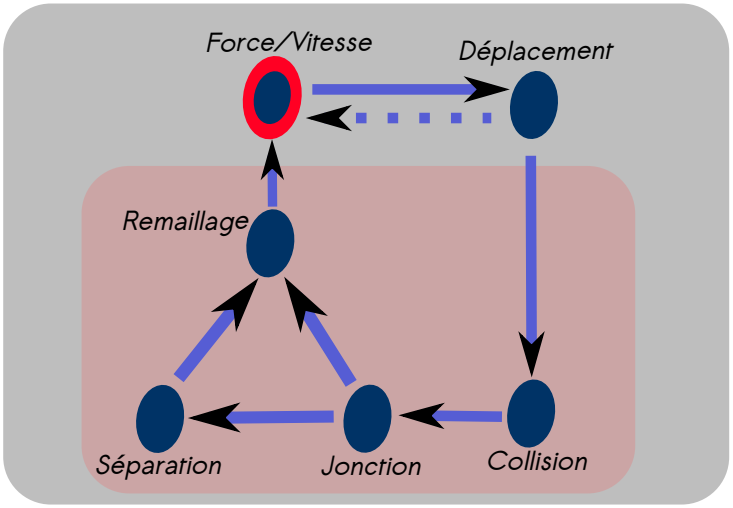


FIGURE 14 – Automate des opérations se produisant à chaque pas de temps.

la collision entre plusieurs dislocations conduisant à la formation d'un nœud physique multi-connecté, et traduit la propention du système à évoluer vers une configuration plus favorable énergétiquement. Considérons la situation présentée à la Figure 14 d'un nœud physique connecté à quatre segments.

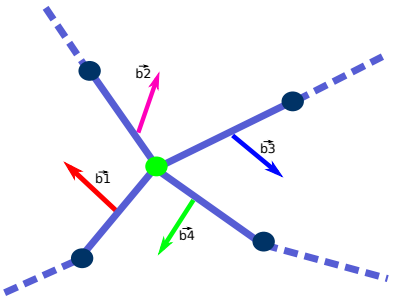
Le nombre de configurations possibles pour séparer un nœud physique avec N connexions est factoriel, et dépend parfois de façon complexe des différents plans de glissement des segments connectés. Des optimisations sont donc faites pour réduire au strict minimum le nombre de possibilités, supprimer les doublons éventuels et ne considérer que les configurations topologiquement admissibles. Suivant [15], la bifurcation vers l'une ou l'autre de ces configurations se fera sur la puissance dissipée $\dot{Q}$ qui est la somme des forces calculées sur chacun des M nouveaux nœuds multipliés par leur vitesse :

$$\dot{Q} = \sum_{i=0}^{M} \vec{V}_i . \vec{F}_i^{driving} ,$$

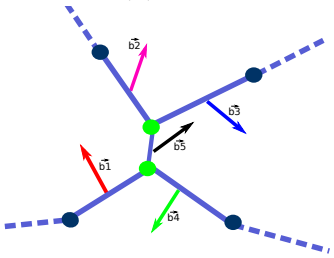
où $\vec{V}_i$ et $\vec{F}_i^{driving}$ sont respectivement la vitesse et la force des nouveaux nœuds physiques. L'ensemble des configurations retenues, y compris le cas où la structure reste inchangée, est donc systématiquement testé. De manière évidente, certaines configurations menant à des topologies aberrantes de chevauchement de segments, comme c'est le cas sur la Figure 15d. Suivant la configuration retenue, de nouveau nœuds et/ou de nouveaux segments sont alors créés.

2.5.2 Détection et traitement des collisions

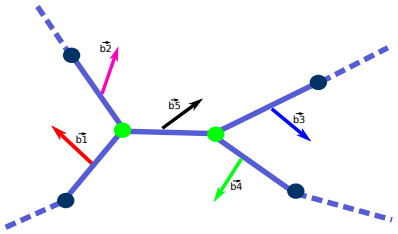
Les collisions résultant du déplacement des lignes sont un autre problème important à résoudre au cours d'une itération. Il s'agit de collisions entre lignes de dislocations ou bien de collisions avec d'autres éléments microstructuraux comme des joints de grains ou des précipités. L'idée de base est de calculer la distance minimum entre deux objets et de déterminer si au cours du pas de temps cette distance devient nulle ou suffisamment



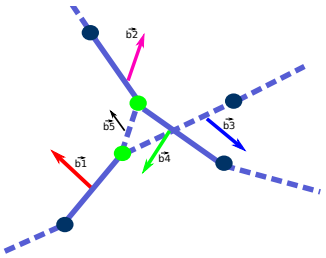
(a) Configuration d'un nœud physique connectant 4 lignes.



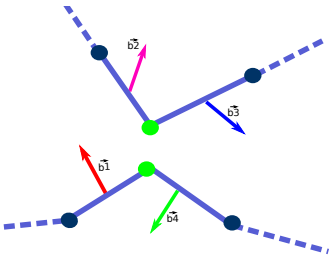
(b) Possibilité 1



(c) Possibilité 2



(d) Possibilité 3



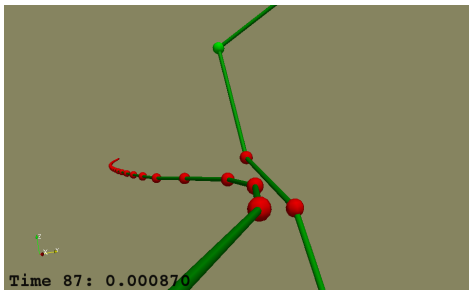
(e) Possibilité 4

FIGURE 15 – Schéma représentant les différentes possibilités de séparation pour un nœud physique.

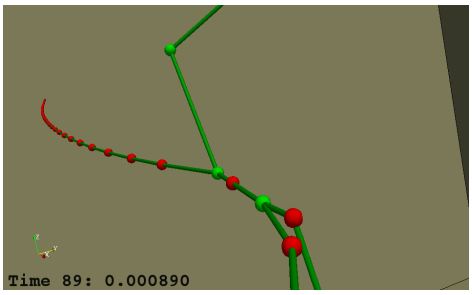
petite pour considérer qu'il y a collision.

La détection des collisions entre segments revient donc à trouver la distance minimale entre des objets 1D se déplaçant de manière rectiligne en 3D sur un pas de temps δt avec une vitesse v constante. Les segments glissant chacun dans un plan bien défini, des optimisations sont possibles selon que ces plans sont identiques (collision 2D) ou non (collision 3D). Chaque collision conduit à une modification de la connectivité et de la topologie des nœuds comme indiqué sur la Figure 16. Ainsi, si deux ou plusieurs segments se superposent, le vecteur de Burgers de la jonction ainsi formée sera calculé comme la somme des vecteurs de Burgers de segments initiaux [5] respectant ainsi la loi de nœuds 0.1.2. Les Figures 16a et 16b présentent le cas où la collision entre deux dislocations de vecteur de Burgers $\vec{b}_1$ et $\vec{b}_2$ conduit à la formation d'une nouvelle jonction de vecteur de Burgers $\vec{b}_3 = \vec{b}_1 + \vec{b}_2$. Les Figures 16c et 16d présentent le cas où $\vec{b}_2 = -\vec{b}_1$, ce qui conduit à annihilation des segments correspondants. Le cas de la collision de dislocations avec la

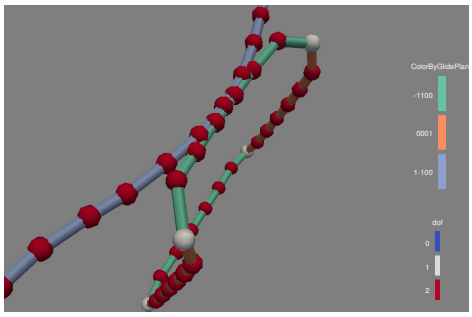
5. La somme se fait en respectant l'orientation de chaque segment



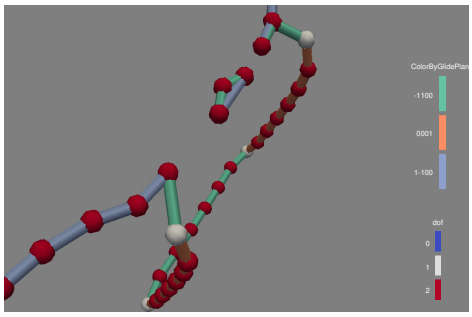
(a) Rapprochement accéléré dû à l'aspect attractif entre deux lignes avant collision.



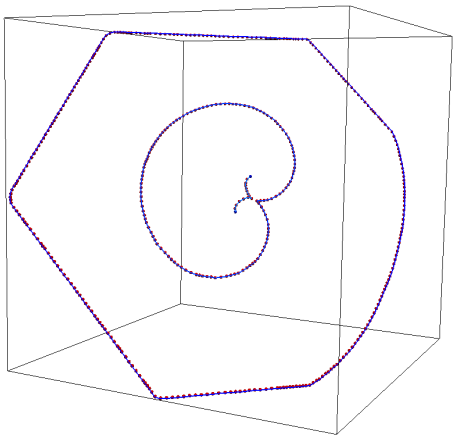
(b) Collision suivie de la formation d'une jonction.



(c) Attraction entre un dislocation coin est une boucle d'irradiation.



(d) Collision avec formation d'une jonction conduisant à l'annihilation.



(e) Collision avec le grain suite à la génération d'une dislocation.

FIGURE 16 – Un sous ensemble de possibilités de collisions au cours d'une simulation.

microstructure est illustré quant à lui par la Figure 16e.

Pour détecter l'ensemble des collisions se produisant au cours du pas de temps, la méthode algorithmique employée dans l'ensemble des codes est basée sur une décomposition en boîtes de l'espace. Au cours du pas de temps δt chaque segment peut *voler* sur une certaine distance. Il convient alors de placer chaque segment dans une boîte et de construire alors une liste d'interactions avec les boîtes voisines. La construction d'une telle liste d'interactions [4] fait passer la complexité de $\mathcal{O}(N^2)$ (pour la version la plus naïve)

à un complexité linéaire $\mathcal{O}(N)$.

2.5.3 Remaillage

Les algorithmes comme la détection de collisions et le calcul du champ de force dépendent de la taille des segments mais aussi de la topologie. Par conséquent avec le caractère très dynamique de ces simulations, l'utilisation d'un maillage adaptatif est essentielle pour décrire la géométrie des lignes de manière optimale. Pour régulariser le maillage on maintient la longueur d'un segment S notée l_s, dans un intervalle $[l_{min}, l_{max}]$. De manière immédiate il existe trois façons d'adapter le maillage pour respecter cette contrainte :

1. supprimer un nœud lorsque celui-ci est trop proche d'un autre nœud ;
2. ajouter un autre nœud lorsque la distance entre deux nœuds est supérieure à la taille de discrétisation maximale ;
3. modifier la position d'un nœud existant pour mieux respecter la répartition le long d'une ligne et ainsi représenter la courbure selon la contrainte.

Puisque l'on calcule la déformation d'un cristal par le déplacement des lignes de dislocation, le remaillage est une source d'erreur. Par conséquent une attention particulière doit être portée pour que le réarrangement ne perturbe pas la géométrie de la ligne au delà d'une certaine tolérance. Enfin, pour pouvoir supprimer un nœud ou autrement dit le fusionner avec un autre nœud, il faut respecter certaines conditions de compatibilité énergétiques. La compatibilité énergétique consiste à vérifier que la fusion des deux nœuds conduit à une relaxation du système, comme pour la séparation d'un nœud physique. Il faut donc vérifier que l'énergie dissipée, E^d par la déplacement des nœuds à la nouvelle position est positive.

$$E^d = F_1.Displacement_1 + F_2.Displacement_2 > 0.$$

2.5.4 Conditions initiales

Le dernier point que nous abordons pour décrire une simulation en DD est la gestion des conditions initiales. Pour avoir des simulations réalistes, il est nécessaire de multiplier les possibilités de paramétrage pour créer tout un ensemble de scénarii. Nous allons décrire les possibilités de pilotage de la simulation, c'est à dire le contrôle de la déformation, puis les conditions aux bords du domaine et enfin la distribution initiale des dislocations.

Pilotage de la déformation

Les deux modes de pilotage implémentés correspondent aux modes de pilotage les plus couramment utilisés du point de vue expérimental. Il s'agit du pilotage à contrainte extérieure imposée, qui correspond aux essais dit de fluage, et du pilotage à vitesse de déformation imposé qui est caractéristique des essais de traction ou de compression.

La première méthode de pilotage est la plus simple à mettre en oeuvre dans une simulation de DD, puisque les dislocations se déplacent dans notre approche sous l'effet

du champ de contrainte qu'elles ressentent. Ce mode de pilotage permet par exemple de déterminer facilement la contrainte d'activation des sources de Frank-Read ou de juger de la stabilité d'un système dans le temps.

La seconde méthode est itérative car elle demande un asservissement ou une adaptation de la contrainte appliquée pour maintenir une vitesse de déformation constante. La vitesse de déformation plastique est, en effet, le résultat du déplacement des dislocations au sein du grain selon la formule

$$\frac{\Delta\epsilon}{\Delta t} = \sum_{s=0}^{N-1} \frac{b_s \otimes n_s + n_s \otimes b_s}{2\Omega} \frac{\Delta A_s}{\Delta t},$$

où ΔA_s est l'aire balayée par chaque segment s au cours d'un pas de temps et Ω le volume du grain. La vitesse de déformation totale s'exprimant comme la somme des vitesses de déformation plastique et élastique

$$\dot{\epsilon}_{elastique} = \dot{\epsilon}_{imposee} - \dot{\epsilon}_{plastique}.$$

Il est alors possible d'ajuster la contrainte appliquée [6] en se basant sur la loi de Hooke, $\sigma = C_{ijkl} \times \epsilon_{elastique}$ avec σ la contrainte, C_{ijkl} le tenseur élastique du matériau et $\epsilon_{elastique}$ la déformation élastique, de façon à imposer la vitesse de déformation souhaitée.

Conditions aux bords

Le choix des conditions aux bords dépend du phénomène physique que l'on cherche à modéliser et a des conséquences à la fois sur les forces ressenties par les dislocations et sur leurs mouvements. Dans sa formulation standard, le formalisme des fonctions de Green utilisés pour le calcul des contraintes 0.2.2 plonge le domaine simulé dans un milieu élastique homogène et infini. Bien que physiquement irréaliste, de nombreuses études physiques sont toutefois possibles.

Deux possibilités supplémentaires sont envisageables :

— Les *conditions aux limites périodiques* sont fréquemment utilisées en DD pour réduire, au moins partiellement, le nombre de segments à simuler tout en s'affranchissant de possibles effets de taille. Dans ce cas, chaque segment ressent à la fois le champ de contrainte interne lié aux autres segments du domaine, mais également celui lié aux copies périodiques de ce dernier. Les lignes de dislocations peuvent alors sortir d'un côté et être réintroduites de l'autre, comme c'est le cas des atomes dans la DM [4]. Des précautions sont toutefois à prendre [18, 67] car ces conditions peuvent engendrer des configurations artificielles lorsqu'une dislocation se rapproche ou se rencontre elle-même comme illustré dans la Figure 17.

— Il est également souhaitable de pouvoir simuler des domaines de taille finie, des milieux élastiquement hétérogènes ou encore des chargements complexes pour se rapprocher au mieux de l'expérience. Si l'impact de ces éléments sur le mouvement des dislocations peut être traité assez aisément via une gestion des collisions, leur

6. L'ajustement peut se faire de façon itérative ou en se basant sur la mesure de la déformation plastique observée au pas précédent.

impact sur le champ de contrainte est beaucoup plus difficile à prendre en compte. Différentes stratégies ont été mises en place pour pallier ce problème via un couplage de la DD avec un code éléments-finis [103, 43, 70, 105]. La mise en place d'une telle stratégie, bien que potentiellement utile, n'a toutefois pas été envisagé dans le cadre de ce travail.

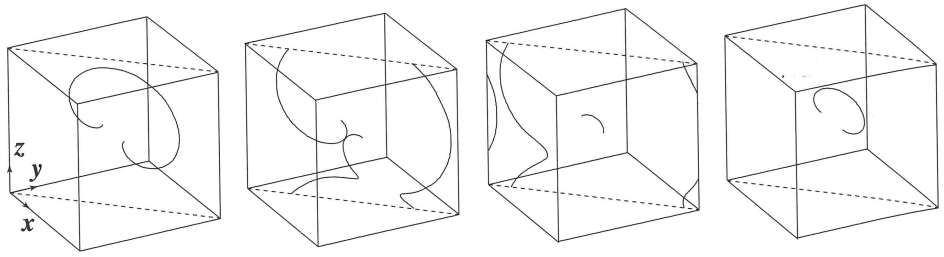


FIGURE 17 – Source de Frank-Read opérant dans un plan de normal (110) avec des conditions aux limites périodiques. La première boucle émise par la source s'auto-annihile totalement au lieu de se propager comme dans le mécanisme présenté à la Figure 10 (image de R. Madec extraite de [67]).

Distribution initiale

La mise en données des dislocations est un élément crucial tant elle conditionne le comportement du grain et la microstructure qui se développera au cours de la simulation (voir par exemple [23, 25]). En effet, si les dislocations peuvent se multiplier via des mécanismes comme celui de la source de Frank-Read 0.1.2, l'apparition *ex nihilo* de dislocations dans un matériau reste discuté du point de vue physique [76]. Sauf cas très particuliers, l'essentiel du comportement plastique d'un grain est donc lié aux dislocations présentes initialement, et qui sont présentent même dans un matériau fortement recuit [60].

La mise en données des dislocations ne peut alors, objectivement, qu'être motivé par des observations microstructurales par MET ou des simulations faites avec des méthodes prenant en compte le mode d'élaboration du matériau. Pour notre étude, les cas applicatifs reposent sur la mesure, par MET, d'une densité et d'une taille de boucles induites par l'irradiation sur un alliage industriel. Respectant ces valeurs, des boucles ont ainsi été introduites dans notre volume de simulation.

3 Introduction aux architectures pour le calcul hautes performances

La simulation numérique nécessite des calculateurs toujours plus puissants. Comme le montre encore la courbe de la Figure 18, issue du classement de la puissance des super calculateurs de juin 2014 du *top500*, l'augmentation linéaire de la puissance de calcul reste vérifiée depuis les années 1990. De plus, il faut moins de 10 ans pour multiplier par 1000 la puissance du meilleur calculateur (*#1* dans la Figure 18). A l'heure actuelle l'ordinateur

le plus puissant est Thianhe-2 situé à l'Université nationale de technologie de la défense en Chine. Cette machine atteint 33.8 PFlops [7] et les perspectives confirment l'arrivée d'une première machine exaflopique ($10^{18} Flops$) avant 2020.

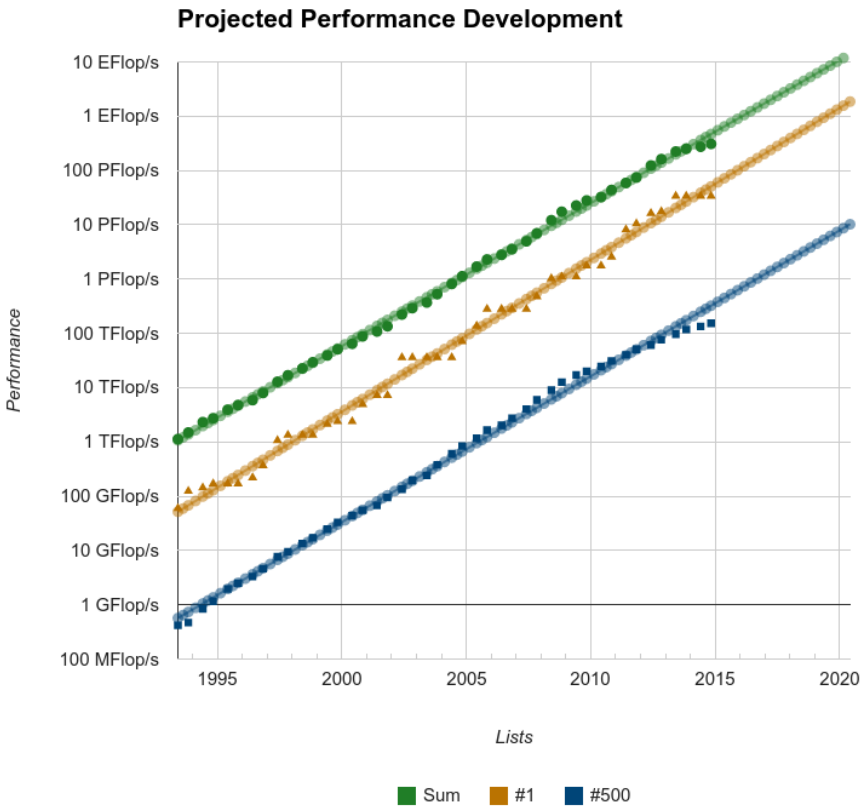


FIGURE 18 – Évolution des super calculateurs. (Source : http ://www.top500.org)

Une machine telle que Tianhe-2 est hétérogène, composée de 3 120 000 cœurs de calculs à savoir 32 000 processeurs Intel Ivy Bridge et de 48 000 Intel Xeon Phi. Cela donne 16 000 nœuds composés chacun de 2 processeurs Ivy Bridge à 12 coeurs cadencés à 2.2Ghz et 3 Intel Xeon Phi. Chaque nœud possède 64 Gigaoctets de mémoire avec 8 Gigaoctets en plus par Xeon Phi. A la vue d'une telle machine deux points vont déterminer si une application sera efficace sur ce type d'architecture :

1. l'équilibrage de charge. Le plafonnement de la puissance des unités de calcul provoquant leur multiplication ($>$ 3 000 000 sur Tianhe-2) rend critique la répartition des calculs. Des stratégies d'équilibrage de charge dynamique combinées aux nouveaux paradigmes de programmation doivent être mises en place pour faire collaborer l'ensemble des unités de calcul.

7. 10^{15} opérations à la virgule flottante par seconde

2. le placement mémoire. Différents types de mémoires sont présentes avec une organisation hérarchique et des temps d'accès très variés. La localité des données devient alors capitale pour prendre en compte la hiérarchie mémoire avec en plus le coût des communications entre les nœuds.

3.1 Architecture des ordinateurs

La puissance des ordinateurs a été guidée par la loi de Moore qui stipule que la densité de transistors sur les processeurs double tous les 18 mois. Mais comme on peut l'observer sur la Figure 19, l'amélioration des temps d'accès aux opérandes en mémoire est nettement plus lent que celui de traitement des opérandes. Par conséquent cela a nécessité de travailler à l'amélioration du traitement des données par les processeurs pour masquer cette différence.

L'une des premières idées est d'utiliser un pipeline en spécialisant des unités indépendantes pour réaliser des instructions en parallèle. Ce parallélisme reste limité car l'exécution finale des instructions reste séquentielle sur les unités de calcul dédiées. Il s'exprime sur le processeur avec des unités spécifiques pour le chargement d'instructions, d'autres pour le décodage et encore d'autres pour l'exécution des opérations.

Ensuite avec l'amélioration de la finesse de gravure, de nombreuses optimisations ont été rendues possibles en complexifiant les processeurs comme par exemple la prédiction de branchement, mais c'est surtout le calcul vectoriel qui va guider les évolutions pour le calcul hautes performances jusqu'à la fin des années 90. Ce premier pas vers le parallélisme au niveau des données, permet d'effectuer une opération basique sur un ensemble de données. Ce n'est que à partir de 2002 qu'une machine scalaire prend la tête du classement *top500*. Depuis, malgré la domination des machines scalaires celles-ci ont hérité d'unités de calcul vectoriel avec les jeux d'instruction SSE, SSE2, puis AVX et AVX2 avec dernièrement des registres jusqu'à 512 bits pour effectuer des opérations sur 8 doubles en parallèle.

Au cours de ces développements, une des évolutions vers la parallélisation consiste à utiliser plusieurs contextes sur une même unité de calcul (multithreading). L'idée de base est qu'en utilisant plusieurs threads, le chargement des données est mieux recouvert en alternant l'exécution des deux contextes. La solution actuelle chez INTEL d'*Hyperthreading* se passe sans changement de contexte, on parle de *Simultaneaous Multithreading*. Cependant, pour les applications scientifiques possédant une forte intensité arithmétique les threads qui partagent le même pipeline peuvent doublement saturer les unités de calcul flottant et donc se faire concurrence. De plus, les applications ayant déjà des optimisations pour l'utilisation mémoire peuvent voir l'utilisation du cache dégradée par ce double contexte. Par conséquent, peu de résultats probants ont été obtenus en utilisant le multithreading dans le secteur du calcul scientifique hautes performances [71].

Finalement, les progrès dans la finesse de gravure des processeurs combinés avec la limitation de fréquence due à la barrière thermique ont poussé les constructeurs à multiplier les cœurs indépendants sur une même puce (Figure 20) pour permettre une exécution parallèle avec un contexte privé à chaque cœur de calcul. Chaque cœur est ainsi indépendant et possède des mémoires privées et d'autres partagées avec une gestion de cohérence de cache comme nous le verrons dans la section 0.3.2.

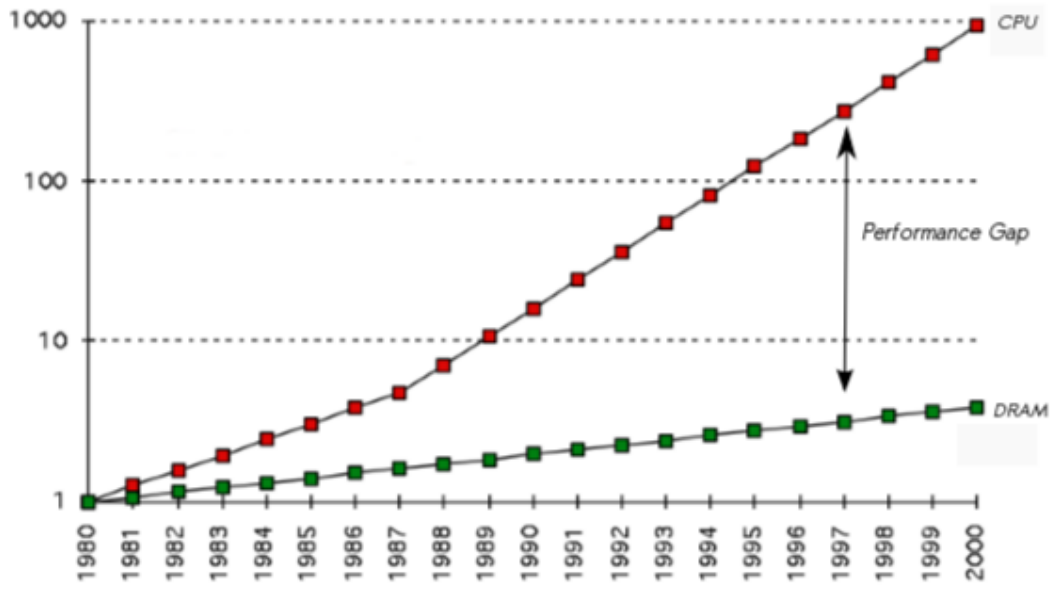


FIGURE 19 – Évolution en terme de performances de la capacité de calcul des processeurs comparée à l'évolution de la rapidité des accès à la mémoire centrale (DRAM). (*Source : extremetech.com*)

Enfin, dernière tendance qui complexifie les architectures (Figure 20), les constructeurs ont développé des accélérateurs spécifiques basés sur des architectures différentes des processeurs généralistes pour augmenter de manière drastique la puissance de calcul d'un nœud. Ces accélérateurs sont vus comme des co-processeurs dans le sens où ils sont toujours couplés à un processeur généraliste. Parmi les différents accélérateurs, deux modèles sont présents dans les calculateurs du *top500*. Les accélérateurs graphiques généralisés pour le calcul (GPGPU) et des co-processeurs Intel Xeon Phi. Ces cartes ont en commun qu'elles utilisent leur propre mémoire. Comme il n'y a pas de cohérence entre la mémoire du processeur et celle de l'accélérateur à l'heure actuelle, la charge incombe au développeur de transférer et synchroniser les données entre les mémoires.

Enfin, parmi toutes ces nouvelles architectures émergentes, un dernier point intéressant à observer est le ratio entre le nombre de calculs effectué par volume de données chargées. En effet, le coût du chargement en mémoire est loin d'être négligeable par conséquent l'utilisation des données c'est à dire le nombre d'opérations qui peut être effectué pour chaque octet transféré depuis la mémoire est crucial. D'après la Figure 21, les accélérateurs sont conçus pour le calcul intensif avec beaucoup plus d'unités de calcul même si les nouveaux processeurs avec de plus en plus d'unités vectorielles améliorent ce ratio. Cependant, ce ratio est important uniquement lorsque l'algorithme est écrit de manière à limiter les transferts mémoire et à intensifier l'arithmétique pour chaque octet transféré. Ainsi, pour assurer cette efficacité mémoire notamment avec l'incorporation de mémoires cache, l'utilisation de techniques algorithmiques spécifiques doivent être considérés comme nous allons le voir dans la section suivante.

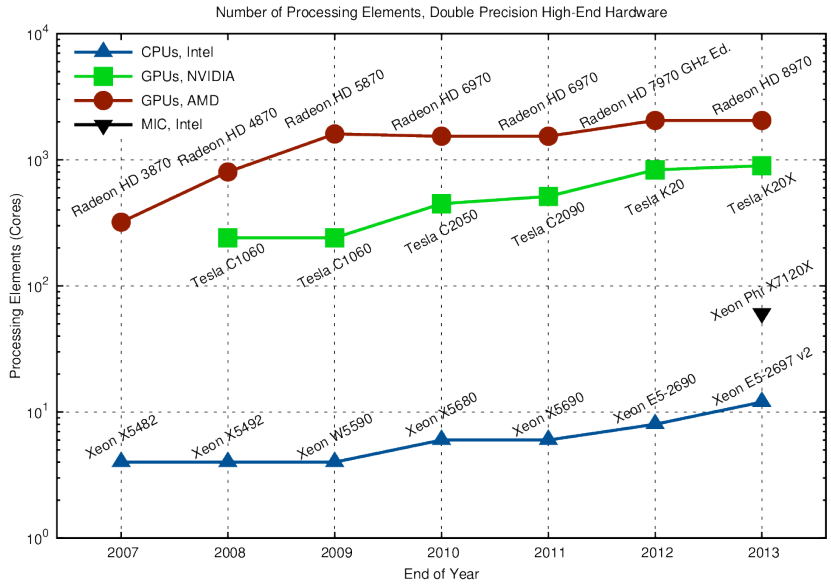


FIGURE 20 – Évolution du nombre d'unités de calcul par puce, avec les processeurs classiques en bleu, puis les accélérateurs avec les GPU en vert et rouge et les Xeon Phi en noir. (*Source : extremetech.com*)

3.2 Les caches et les techniques algorithmiques

Grâce à l'amélioration de la finesse de gravure des processeurs, la vitesse du traitement des opérations a suivi une tendance linéaire pendant plus de 30 ans pour se stabiliser autour de 3.5 GHz en se confrontant au problème de la dissipation thermique. Du côté de la mémoire, l'augmentation de la densité des transistors induit une augmentation des capacités en terme d'espace mais la performance au niveau des temps d'accès n'a jamais pu suivre la même tendance comme on peut le lire sur le graphique de la Figure 19. De plus comme nous venons de le voir la multiplication des cœurs a rendu ces accès mémoires encore plus critiques.

Ces problèmes de performance pour les transferts entre la mémoire centrale et le processeur, combiné à la multiplication des cœurs se traduit donc par l'apparition d'une hiérarchie mémoire. De manière classique, nous trouvons 3 niveaux de caches. On parle de hiérarchie mémoire tant pour les différentes tailles et vitesses d'accès que pour rappeler que les données transitent par ces différents niveaux de cache avant d'être traitées par les registres et unités de calcul. De manière à donner un ordre d'idée du rapport entre la taille de mémoire et la latence d'accès, le Tableau 1 présente ce que l'on peut trouver dans la littérature [100, 26]. Ainsi, si une donnée n'est pas présente dans le niveau de cache L1, il nous faudra 65 cycles pour la rapatrier depuis la mémoire centrale ce qui implique une pénalité de 60 cycles.

Depuis les années 90 et le décrochage entre la vitesse du processeur et la vitesse d'accès mémoire, les recherches dans l'écriture d'algorithmes performants en mémoire ont été florissantes [69, 50, 28]. Deux grands types de possibilités s'offrent aux développeurs pour une utilisation efficace du cache, avec une attention particulière sur la localité temporelle

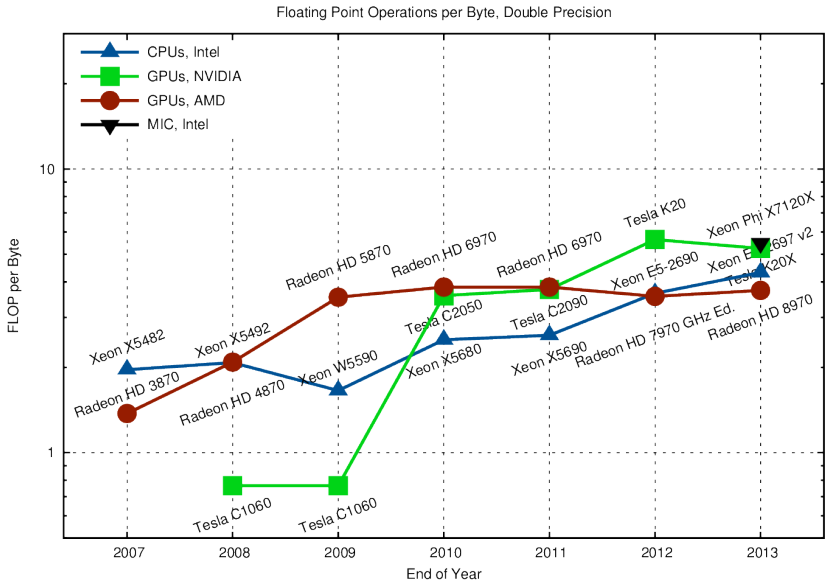


FIGURE 21 – Évolution des capacités d'opération flottante (flop) par octet transféré. (*Source : extremetech.com*)

Niveau mémoire	Taille	Latence
cache L1	32 Ko	4 cycles
cache L2	256 Ko	10 cycles
cache L3	25 Mo	40 cycles
RAM	Infini	65 cycles

TABLE 1 – Caractéristiques d'une hiérarchie mémoire à 3 niveaux sur un nœud moderne avec un processeur Intel Xeon.

et la localité spatiale des données.

— Pour la localité temporelle, le principe découle directement de l'organisation hiérarchique de la mémoire. Nous venons de voir qu'il y a un coût pour charger une donnée de la mémoire centrale donc il faut utiliser cette donnée chargée tant qu'elle se trouve dans le niveau de cache le plus proche des registres.

— Pour la localité spatiale, la sensibilité vient de la façon dont les données sont chargées. Lors du chargement d'une donnée c'est un bloc entier de données qui est transféré, on parle de ligne de cache. Pour éviter des chargements coûteux l'algorithme de gestion du cache, suppose que les accès aux données qui sont contiguës d'une donnée a sont les plus probables. Il revient au développeur d'organiser sa structure de données de telle sorte que l'organisation spatiale des données soit cohérente avec leur utilisation.

Lorsqu'une donnée n'est pas présente dans le cache pour exécuter une opération, celle-ci doit y être chargée. Cette opération est un défaut de cache. Ce chargement, comme indiqué dans le tableau 1, prend un certain temps et donc par conséquent ralentit l'exécution. On peut extraire trois catégories de défauts de cache :

— Défaut initial : lorsque l'on accède à la donnée pour la toute première fois le chargement est inévitable ;
— Problème de taille : une donnée peut être éjectée du cache par l'algorithme de remplacement pour des raisons de taille. Si trop de données sont chargées, selon les algorithmes de remplacement, les données les moins utilisées ou les plus anciennes sont remplacées ;
— Défaut par conflit : La ligne est remplacée suite au chargement d'autre banc mémoire qui entre en conflits du fait de la politique de "mapping" entre les niveaux mémoire.

A cette liste s'ajoute aussi, les défauts de cache liés aux architectures multi-coeur. En travaillant en parallèle, le défaut est provoqué pour garantir la cohérence mémoire. Lorsqu'une donnée est modifiée par un thread, elle est invalidée dans les caches des autres cœurs qui ont chargé la même ligne de cache. Cette mise à jour se fait donc avec le coût d'un chargement d'une ligne de cache depuis la mémoire centrale. Le besoin de localité spatiale est encore plus critique en parallèle.

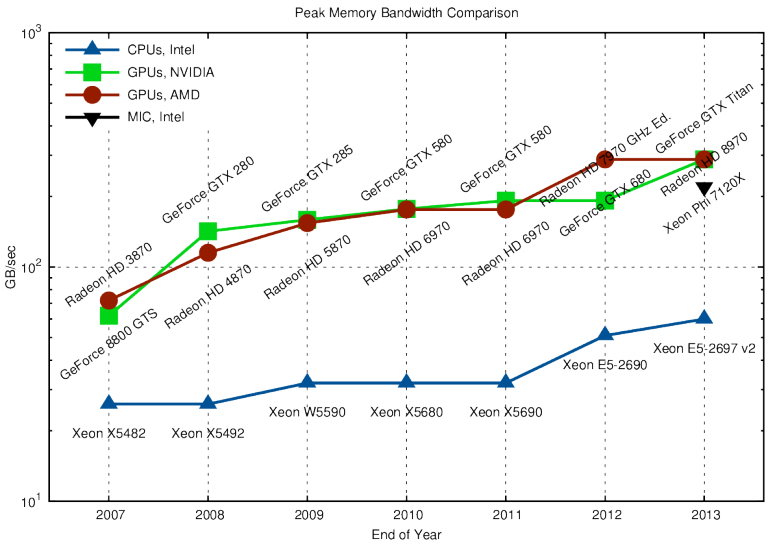


FIGURE 22 – Évolution de la bande passante mémoire.

Un algorithme s'exécutant sur un ensemble de données peut donc être limité par plusieurs facteurs. Pour mettre en relation ce que nous avons vu précédemment, nous savons maintenant que les défauts de cache avec leur différentes origines sont une des explications de l'inefficacité de l'application. De plus, la Figure 22 montre que la bande passante mémoire ne s'améliore que très peu d'une génération de puce à l'autre, nous comprenons qu'une attention particulière doit être portée au niveau algorithmique pour optimiser l'utilisation des données. On peut être limité soit par la non localité des accès mémoire soit par la non prédictibilité de ceux-ci. Ces problèmes se traitent de manière différente comme nous allons le voir mais conduisent au final à prendre en considération le coût des accès mémoire.

3.2.1 Algorithmes efficaces en mémoire

Deux grands types d'approches permettent d'écrire des algorithmes efficaces sur cette hiérarchie mémoire :

— *Cache Aware*. L'idée est de se baser sur la connaissance de l'architecture mémoire pour développer des structures de données et des algorithmes efficaces. Pour cela, il faut connaître à la fois la taille des données que l'on traite, les patterns d'accès mais aussi la taille des différents niveaux mémoire de la machine. L'inconvénient évident est de définir l'algorithme spécifiquement et uniquement pour un type de machine. L'implémentation des *BLAS (Basic Linear Algebra Subprograms)* pour le calcul matriciel bénéficie d'optimisation pour charger en mémoire des blocs de taille adéquate. Ainsi à la compilation on prend en considération les spécificités de l'architecture cible [66].

— *Cache Oblivious*. Ce modèle est plus générique dans le sens où la taille du cache n'est pas directement considérée. Cependant il suppose une décomposition du travail différente et donc souvent plus complexe pour s'adapter à toutes les possibilités de taille de cache. Une vision récursive de décomposition de l'algorithme est souvent nécessaire pour arriver quelle que soit la machine à des tailles de données adaptées qui rentreront dans le cache. Les travaux de Frigo [50] notamment sur la FFT en sont une parfaite illustration.

L'écriture d'un algorithme dédié à une machine (*Cache aware*) sera toujours au moins autant voir plus performant qu'un algorithme *Cache Oblivious*. Mais le coût lié à la complexité du développement pour un unique type de machine cible doit être pris en compte, en sachant qu'un algorithme *Cache Oblivious* peut obtenir des performances très proches en étant bien développé.

Ces modèles peuvent se concrétiser à travers différentes techniques de développement que nous allons aborder. On peut jouer au niveau de la structure de données, pour organiser les données en mémoire afin que le chargement d'une ligne de cache soit utile pour l'algorithme, mais aussi avec le pattern d'accès aux données pour mettre en adéquation la structure de données avec le chargement du cache et l'utilisation des données.

3.2.2 Stockage mémoire des données

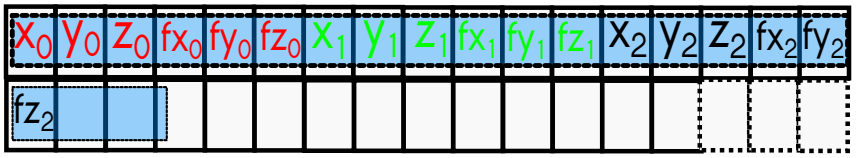
Chaque architecture (CPU, GPU, Xeon Phi) possède sa propre hiérarchie mémoire, sa propre stratégie de chargement et de remplacement de lignes de caches. Cependant, malgré leurs différences chacune nécessite des accès coalescents et cohérents où l'organisation des données joue un rôle capital. Selon cette organisation en mémoire un algorithme peut avoir des performances très variables.

Nous pouvons immédiatement distinguer deux façons de stocker les données en mémoire :

— ***Array of Structures* (AOS)**. Les données sont stockées en mémoire sous forme d'un tableau de structure (Figure 23a). Cette méthode peut être vue comme plus proche de la pensée de l'utilisateur. Les données d'un type, comme par exemple le type *Nœud*, sont groupées de manière contiguë dans la structure.

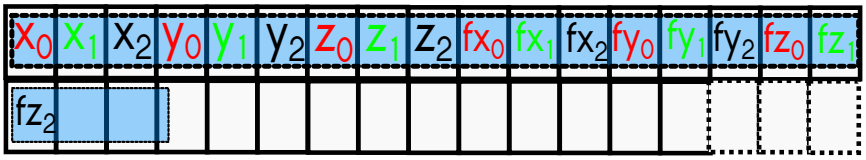
— ***Structures of Arrays* (SOA)**. Cette structure de données (Figure 23b) vise à

grouper de manière contiguë les mêmes données d'un type. Le type *Nœud* n'est plus présent comme un bloc en mémoire mais un entrelacement des membres de la structure. On trouve donc par exemple pour un ensemble de données de type *Nœud*, les coordonnées en x contiguës puis celle en y puis celle en z.




```
struct Noeud{
  double x,y,z;
  double fx,fy,fz;
}
```

(a) Structure de données où les nœuds sont organisés en tableau de structures.




```
struct Noeuds{
  double x[K],y[K], y[K];
  double fx[K],fy[K],fz[K];
}
```

(b) Structure de données où les nœuds sont organisés en structure de tableaux.

FIGURE 23 – Stockage des données en structure de tableaux ou tableaux de structures pour des chargements optimisés du cache.

Pour illustrer les répercutions de ces deux concepts, prenons un exemple avec le type *Nœud* pour l'écriture d'un algorithme afin de déplacer des nœuds dans une certaine direction.

Considérons tout d'abord, une pseudo-interface pour accéder aux données quelque soit la structure. Ainsi, que le stockage soit *SOA* ou *AOS* ; comme présenté dans Figure 24a et Figure 24b, l'accès se fait de manière uniforme à travers l'interface. En utilisant cette interface on masque à l'utilisateur l'implémentation de la structure de données sous-jacente.

Nous considérons alors les Algorithme 1 et Algorithme 2. Pour ces deux algorithmes, le but est le même puisqu'ils consistent à déplacer un nœud comme cela peut être fait lors de la mise à jour des positions. Dans ce cas simple, nous pouvons noter deux choses.

Tout d'abord, les données ne sont utilisées qu'une seule fois donc il n'existe pas d'optimisations pour augmenter l'intensité arithmétique avec le ratio flop par transfert mémoire. Deuxièmement, le type traité ici le *Nœud* est grandement simplifié en étant composé de

```
double getX(i){
  return noeuds[i].x;
}
```

(a) Implémentation pour une structure de type tableau de structures (AOS).

```
double getX(i){
  return noeuds.x[i];
}
```

(b) Implémentation pour une structure de type structure de tableaux (SOA).

FIGURE 24 – Interface standard pour appeler et accéder aux données selon le stockage mémoire : *double getX(i).*

peu de données membres. Cela n'est pas forcément le cas dans une simulation où les données sont de taille bien plus grande et peuvent donc saturer plus vite les lignes de cache.

Dans ce cadre là, nous voyons de suite que dans Algorithme 1, avec un stockage en tableau de structure(AOS), le cache n'est pas bien utilisé. Non seulement, l'ensemble de la structure *Nœud* est mise en cache, suite au défaut de cache initial, mais en plus l'écriture de l'algorithme, décomposé en deux boucles entraîne le double de défauts au final. Le fait de transférer les données par lignes de cache avec une organisation AOS entraîne le transfert de données inutiles depuis la mémoire centrale comme par exemple les données du vecteur force. Ce chargement favorise la saturation de la bande passante mémoire, tout en remplissant le cache avec des données qui ne seront pas utilisées et augmente par conséquent les défauts de cache de part la stratégie de remplacement.

Au contraire, un stockage sous forme de tableau de structure (SOA) minimise les défauts de cache en chargeant uniquement des données utiles. En plus, la décomposition en deux boucles n'a aucun impact puisque les données membres sont contiguës en mémoire. On réduit aussi par la même occasion le volume de données transférées.

Dans Algorithme 2 plus uniforme en terme d'accès, là aussi, le format de stockage en tableau de structure (AOS) implique des transferts inutiles de données dans les niveaux de cache. Au contraire le format structure de tableau (SOA) s'en sort aussi bien que précédemment en terme de défaut de cache et de volume de données chargées, moyennent le nombre de lignes de cache pouvant être chargées par coeur (mémoire partagée) .

Algorithme 1 : Algorithme déplaçant un nœuds en X et en Z.

Données : double $delta_{XYZ}$[3], Noeuds_type noeuds

```
1 pour int i=0;int<NbNodes;++i faire
2 |  noeuds.getX(i)+=delta_XYZ[0];
3 Traitement...;
4 pour int i=0;i<NbNodes;++i faire
5 |  noeuds.getZ(i)+=delta_XYZ[2];
```

L'organisation des données en tableau de structures est plus intuitif pour le programmeur que le modèle structure de tableau, mais cela rend la gestion pour le contrôleur mémoire plus difficile. Nous avons notamment mis en évidence des chargements de lignes

Algorithme 2 : Algorithme déplaçant un nœuds en accédant à la totalité des coordonnées.

Données : double $delta_{XYZ}[3]$, Noeuds_type noeuds

```
1 pour int i=0;i<NbNodes;++i faire
2     noeuds.getX(i)+=delta_XYZ[0];
3     noeuds.getY(i)+=delta_XYZ[1];
4     noeuds.getZ(i)+=delta_XYZ[2];
```

de cache avec des données inutiles ou encore des lectures et écritures sur des données non contiguës.

De plus, sur un maillage de dislocations l'interdépendance des données et surtout les schémas (pattern) d'accès aux données s'avère plus compliqués qu'un simple algorithme transversale tel que Algorithme 2. En effet, en DD les accès peuvent se faire de façon non régulière, comme la recherche de voisins immédiats, ou de l'ensemble des voisins proches. La saturation de la bande passante est alors encore plus critique pour la performance de l'algorithme.

3.2.3 Diviser pour régner

Cette méthode est bien connue et employée à travers de nombreuses idées algorithmiques [38] comme par exemple le tri fusion [91] qui consiste à découper une large structure en sous blocs. L'objectif est de pouvoir travailler sur chaque sous bloc en le faisant tenir au plus proche dans les niveaux de cache. Cette vision d'un algorithme entre dans la catégorie *Cache oblivious* si le découpage est effectué de manière récursive jusqu'à obtenir des blocs suffisamment petits pour entrer dans le cache et travailler efficacement en mémoire. On sera dans la catégorie *Cache aware* si le découpage en bloc est manuel comme pour le calcul matriciel où la taille des blocs est fixée.

De plus, la division d'une structure revêt un autre intérêt avec la possibilité de travailler localement sur les données contenues dans le cache. En effet, en découpant la structure en sous ensembles on peut les gérer et les modifier comme une sous structure indépendante. La cohérence de la structure de données est donc maintenue sans entraîner de modification de la mémoire à l'échelle globale. Cette méthode de calcul par bloc est très utilisée pour profiter de la hiérarchie mémoire mais doit souvent être accompagnée de techniques pour mettre en relation la mémoire et la localité spatiale des données. En effet, il faut notamment chercher à éviter les accès aléatoires en mémoire, qui limitent l'algorithme par la bande passante. Cette cohérence spatiale est d'autant plus importante pour la cohérence de cache dans les algorithmes parallèles avec les défauts liées aux *false sharing* qui vont invalider des lignes de cache du fait du travail concurrent de plusieurs threads sur des données proches en mémoire.

3.2.4 Organisation spatiale hiérarchique

Une structure de données peut être utilisée pour gérer l'organisation spatiale des données et donc pouvoir parcourir et accéder aux informations selon un nouvel ordre lié à

la position dans le domaine de simulation. Ces structures hiérarchiques qui entrent aussi directement dans l'idée présentée précédemment *diviser pour régner* sont souvent de type arborescente. Dans ce cas, le découpage de l'espace peut prendre plusieurs formes comme la grille régulière ou un R-Arbre avec la recherche de la boîte englobante minimum pour chaque objets. Les différentes implémentations doivent prendre en compte le caractère uniforme ou non de la distribution notamment pour des raisons de coût d'espace mémoire.

Finalement, ces découpages, indispensables pour la mise en place des algorithmes, ne sont pas suffisants pour gérer à eux seuls la localité dans la gestion des données. L'utilisation de *space filling curves* [92] permet justement de stocker des données multidimensionnelles comme des données spatiales 3D dans un espace unidimensionnel pour gérer la localité de manière plus aisée. On trouve différentes implémentations de ces courbes comme la Z-curve ou la courbe de Hilbert qui parcourent l'ensemble des cellules de l'espace hiérarchisé en prenant en compte la proximité de ces cellules.

Finalement, une fois ces aspects algorithmiques considérés, il va falloir faire coopérer les différentes unités de calcul en choisissant un ou une combinaison de modèles de programmation.

3.3 Le parallélisme : Modèles de programmation

Avec la complexification des architectures et leur évolution perpétuelle, des paradigmes et bibliothèques ont été développés pour pouvoir bénéficier au maximum de ces nouvelles capacités de calcul à un niveau de programmation relativement haut niveau.

3.3.1 La logique de partage des données

Programmation mémoire partagée

On parle d'architecture à espace d'adressage global lorsque plusieurs cœurs de calcul ont accès à une même zone mémoire. Ces cœurs de calcul sont utilisés de manière naturelle à l'aide de threads qui sont des processus légers. Il s'agit d'un parallélisme à grain fin avec dans la plupart des utilisations un parallélisme au niveau des données.

L'utilisation de ces threads est facilitée par plusieurs bibliothèques. Parmi les différentes possibilités qui existent, nous pouvons noter le standard Posix Thread [108] qui définit un ensemble de fonctions C bas niveau pour un contrôle très fin des threads. L'autre référence est le standard OpenMP [22]. Mis au point par un consortium d'entreprises acteur du calcul parallèle (IBM, AMD...) il permet notamment de masquer au programmeur les détails de gestion des threads avec une approche par directives. Cependant, pour être efficace il est important de considérer toute la hiérarchie de l'architecture tant au niveau des cœurs, organisés par socket, qu'au niveau mémoire. Au niveau mémoire en particulier, il faut bien sûr veiller à protéger les données pour maintenir leur cohérence lorsque plusieurs threads mettent à jour des données partagées et veiller notamment sur les architectures NUMA (*Non Uniform Memory Access*) moderne au placement de threads sur les cœurs.

Programmation mémoire distribuée

Pour les architectures à espace d'adressage disjoint, chaque processus est reconnu comme étant indépendant en s'exécutant de manière concurrente et nécessite donc de communiquer pour échanger des informations entre eux. Le modèle de passage de message permet justement d'effectuer des communications en passant par le réseau. Il existe différentes mises en œuvre du paradigme comme PVM (Parallel Virtual Machine) mais nous citerons MPI [56] (*Message Passing Interface*) comme la norme de référence pour le calcul scientifique. MPI identifie chaque processus avec un numéro unique et permet d'effectuer des communications collectives ou point à point et des synchronisations entre les processus d'un même *communicateur*. Il existe deux principaux challenges pour tirer profit de ces architectures distribuées avec ce paradigme, à savoir, l'équilibrage de charge entre les unités de calcul et la prise en compte des temps de communication entre plusieurs nœuds puisque les données doivent passer par le réseau.

Ce modèle est considéré comme un parallélisme à gros grain, qui peut être combiné avec un parallélisme à grain fin pour avoir un modèle hybride et utiliser au mieux les ressources et spécificités de la machine notamment en réduisant le volume de communications. Différents ratios processus MPI/threads sont possibles selon les architectures mais aussi selon les simulations où la décomposition en domaine peut avoir une influence très variable.

On peut lancer uniquement des processus MPI, où chaque processus est placé sur un cœur du processeur. Cela entraîne un surplus d'échange de messages mais à au moins l'avantage de n'utiliser qu'un seul paradigme de programmation.

A l'inverse nous pouvons lancer un processus MPI par nœud et autant de threads que de cœurs. De cette façon les échanges de message sont réduits au minimum, tout en bénéficiant de la mémoire partagée au niveau des threads. Enfin, selon la complexité de la machine, certains nœuds pouvant être composés d'une dizaine de sockets provoquant de forts effets NUMA, il peut être intéressant de considérer la solution intermédiaire en lançant par exemple un processus MPI par socket avec autant de threads que de cœurs par socket.

Finalement, avec ces différents niveaux de parallélisme, et la spécificité de chaque architecture, nous rencontrons différents modèles algorithmiques pour attaquer ces niveaux de parallélisme.

3.3.2 Les modèles algorithmiques

Du côté de la programmation il existe plusieurs modèles pour exploiter le parallélisme de la machine. Pour chaque modèle de programmation, une approche différente est utilisée et différents challenges doivent être relevés. Ainsi chaque modèle n'est pas adapté à n'importe quel algorithme.

— SIMD pour *Single Instruction Multiple Data* : Cette approche est parfaitement adaptée aux unités vectorielles et aux GPUs qui défendent cette idée de parallélisme au niveau des données. Cette stratégie vise à exécuter sur toutes les unités de calcul les mêmes instructions sur leurs données respectives. Elle implique une synchronisation très forte et une grande régularité des calculs.

— SPMD pour *Single Program Multiple Data* : Ce modèle implique que chaque processus est indépendant et exécute le même programme. Il faut alors décomposer les données entre chaque processus et les échanger explicitement pour maintenir la cohérence globale de la simulation. Ce modèle est parfaitement adapté dans le monde du calcul hautes performances aux architectures distribuées avec une mise en œuvre à travers le standard MPI.
— Enfin, le parallélisme par tâches est en cours d'émergence. Le code décomposé en tâches peut être vu comme un graphe (DAG *Directed acyclic graph*) où les nœuds sont les tâches et les arêtes les dépendances de données entre les tâches. Ensuite il faut le déployer sur l'architecture cible à travers des moteurs d'exécution spécialisés ou avec les dernières évolutions du standard OpenMP. Ce modèle prouve de plus en plus son efficacité notamment face à la complexification des plate-formes. Cette programmation par tâche doit permettre un équilibrage dynamique de la charge entre les différentes unités de calcul [2]. Les moteurs comme Charm++ [65] et StarPU [9], permettent justement d'ordonnancer les tâches sur l'unité de calcul la plus appropriée que ce soit un GPU, un Xeon Phi ou un cœur du CPU.

A la vue de ces modèles de programmation et de leur diversité au niveau de l'approche du parallélisme, nous avons identifié différents challenges qu'un code moderne doit relever pour réaliser des simulations de manière efficace en parallèle.

4 Les codes existants : capacités et limitations

Comme nous l'avons vu, la théorie des dislocations aura bientôt un siècle mais il faut remonter aux années 1960 [45, 44] pour voir les premiers développements d'un code de dynamique des dislocations en deux dimensions d'espace. Celui-ci permettait de modéliser le mouvement d'une seule dislocation dans son unique plan de glissement. Il s'agissait principalement de déterminer par la simulation la contrainte critique d'activation d'une source de Frank-Read. Les codes suivants, toujours en deux dimensions se sont complexifiés pour pouvoir étudier quelques phénomènes d'interactions entre plusieurs dislocations avec par exemple les travaux d'Amodeo et Ghoniem [6, 53].

Les simulations 2d permettent de comprendre certains phénomènes de formation de structures particulières, comme l'agrégation de dislocations (*pile-up*) au bord de grains sur les polycristaux [79]. Cependant, ces simulations sont très limitées dans la compréhension de phénomènes complexes liés aux interactions entre dislocations à courte portée. Une extension existe avec les codes "2.5d". Ce type de simulation prend en compte, à partir des comportements obtenus par des simulations 3d, les phénomènes tels que la multiplication de dislocations, la formation de jonctions, l'annihilation, la montée tout en restant dans un espace à 2 dimensions, pour limiter le coût des calculs. Cette approche est utilisée pour l'étude des joints de grain avec gestion du blocage, de la réflexion et de la transmission des dislocations au bord du grain [54]. Ces simulations apportent un gain d'information par rapport aux simulations 2d mais restent limitées par la simple hypothèse que l'ensemble des dislocations sont considérées rectilignes infinies.

Ce n'est qu'au début des années 1990 qu'un premier code de DD en trois dimensions

d'espace est développé. Il est écrit par Ladislas Kubin, Gilles Canova et Yves Brechet [21, 68], en considérant un modèle de discrétisation *vis-coin*. Depuis, avec l'engouement pour la simulation multi-échelle, la DD placée à l'échelle mésoscopique connaît un fort essor et l'on dénombre aujourd'hui une dizaine de codes 3d à travers le monde. Ce code précurseur, qui prend le nom de Micromegas (mM) a donné lieu à de nombreuses recherches dont notamment la thèse de Benoît Devincre [31] à l'ONERA et celle de Ronan Madec [73] pour une extension du modèle en type *vis-mixte-coin*.

En parallèle, Gilles Canova développe à l'Institut National Polytechnique de Grenoble, une nouvelle branche appelée TRIDIS avec notamment les travaux de Marc Verdier [107] et Marc Fivel [43] et en 2006 le travail de parallélisation de Shin [97]

Enfin, une dernière branche de ce même code originel, MobiDic, est actuellement développée au CEA DAM par Ronan Madec. Le code est spécialisé pour les simulations dans les matériaux BCC.

MicroMégas et Tridis dérivant du même code source, ils emploient à peu près les mêmes algorithmes et stratégies de parallélisation. Le parallélisme est basé uniquement sur une décomposition du domaine de simulation, en utilisant le paradigme MPI pour attribuer un sous domaine par processus. Concernant les structures de données, une différence importante entre MicroMégas et Tridis est l'utilisation pour l'un, de tableaux statiques qui doivent être suffisamment grands pour supporter l'augmentation du nombre de segments tandis que Tridis utilise une liste chaînée pour s'adapter à cette évolution. L'utilisation d'un tableau statique pose notamment des problèmes de fragmentation avec la gestion des trous lors de la suppression des segments, tout comme la liste chaînée avec une fragmentation des accès mémoire. Au niveau algorithmique, le calcul du champ de contrainte est décomposé entre calcul du champ proche et celui du champ lointain. Pour le champ lointain, la contrainte élastique de chaque cellule est approchée en son centre pour calculer l'apport de cette contribution sur le centre des cellules cibles. En parallèle, ce calcul nécessite des communications de type $ALLTOALL$ très coûteuses. Sous l'hypothèse d'une évolution lente du champ lointain cette communication est effectuée seulement tous les n pas de temps [97] tandis que le champ proche est calculé directement avec les 26 boîtes voisines à chaque itération. Pour la gestion des modifications de la topologie, l'ensemble des boîtes voisines d'un sous domaine est échangé pour synchroniser les modifications. Cela engendre par conséquent un fort volume de communication. De plus, les domaines doivent communiquer par vagues pour mettre à jour les informations, du fait de la discrétisation vis-coin qui contraint les modifications entre deux processus partageant une même ligne de dislocation afin de garder la correspondance. Ce modèle de communication nécessite de forts points de synchronisation pour communiquer les modifications et pour créer les correspondance avec les sous domaines voisins. La distribution de la charge entre processus se fait en répartissant les boîtes utilisées pour le calcul du champ proche, les segments étant affectés à ces boîtes. La répartition de ces boîtes est effectuée à l'initialisation et peut être modifiée de manière limitée par la suite. Plusieurs découpages sont possibles à l'initialisation [89]. Premièrement, la répartition peut se faire en précisant le nombre de processus dans chaque direction de l'espace. Ce mode de découpage est notamment limité par le fait que le nombre de processus par direction ne pourra pas évoluer. Ainsi l'équilibrage ne peut pas être calibré finement pour chaque sous domaine selon le déplacement des segments. Un deuxième mode de découpage est basé uniquement sur le

nombre de processus. Dans ce cas là, les boîtes sont réparties de manière équitable entre les processus pour distribuer au mieux l'espace de simulation. Ce mode de répartition est limité par l'aspect hétérogène de la distribution des segments et aussi par la non prise en compte du nombre de voisins pour les phases de communications. Dans MobiDic [89], la scalabilité du code est démontrée jusqu'à 32 cœurs mais par la suite des limitations apparaissent notamment à cause du déséquilibre de la charge par processus combiné au mode de communication par vagues qui place les processus en attente.

Aux Etats-Unis, quelques années après ces premiers développements français, le modèle de discrétisation *nodal* se développe suite aux recherches de Robert Kukta [95]. Par rapport à la discrétisation vis-coin, il offre une meilleure approximation de la courbure de la ligne, avec moins de degrés de liberté, et surtout la formation de tous types de jonctions.

Parmi les diverses initiatives, nous pouvons citer le code propriétaire 3d *PARANOID* (**Para**llel **No**dal **I**BM **D**islocation) développé par Klaus Schwarz [93] au Research Center d'IBM à New York. Le code de Nasr Ghoniem écrit en 1999 est toujours développé au Departement of Mechanical and Aerospace Engineering UCLA. Sa spécificité vient de la discrétisation des lignes de dislocations par des splines cubiques qui permettent une meilleure description de la géométrie avec cependant un coût de calcul plus élevé de par la forme des fonctions de base. Le travail de parallélisation entrepris par Wang en 2006 [109] est basé sur une décomposition du domaine de simulation avec l'utilisation uniquement du paradigme MPI. Pour le calcul du champ de contrainte élastique une méthode avec expansions multipôle de complexité $\mathcal{O}(N)$ est employée [109]. Pour décomposer le domaine, un arbre binaire est construit en séparant de manière récursive en deux partitions égales le nombre de degrés de liberté. La distribution est basée sur un nombre de degré de liberté maximum par processus avec la contrainte supplémentaire de grouper autant que possible les degrés de liberté d'une même ligne dans le même sous domaine. Pour maintenir l'équilibrage de charge, l'arbre est reconstruit toutes les 50 à 300 pas de temps. Pour un cas avec 36 000 degrés de liberté une accélération de 44 est obtenue sur 60 coeurs.

Le code nodal développé à Karlsruhe par l'équipe de D. Weygand est parallélisé en utilisant uniquement le paradigme OpenMP en mémoire partagée [111]. Le maillage des dislocations est stocké dans une liste chaînée avec un mécanisme de *lock* pour permettre l'insertion et la suppression dans les phases de remaillage en parallèle. Le calcul du champ de force considère un calcul direct des interactions avec une complexité quadratique en fonction du nombre de segments. Comme pour *mM*, le calcul complet, trop coûteux n'est effectué que toutes les n itérations et seul le champ proche est mis à jour à chaque évaluation des forces. La vitesse des nœuds est calculée en résolvant le système linéaire par une version parallèle du gradient conjugué. Ce calcul semble être le plus pénalisant dans leur implémentation en ajoutant un assemblage global du système linéaire à chaque pas de temps et des indirections coûteuses. Au final, leur développement est limité par les ressources disponibles sur un seul nœud en terme de mémoire et de puissance de calcul. Le code atteint une accélération de 9.6 sur 16 coeurs d'un processeur XEON à 4 sockets [111].

Enfin le code ParaDiS pour **Para**llel **Di**location **S**imulator du Lawrence Livermore

National Laboratory (LLNL) a été spécialement développé depuis 2002 pour atteindre l'objectif de réaliser de manière quotidienne des simulations à très grande échelle en exploitant les ressources de calcul massives du LLNL. Paradis est le code de DD le plus avancé en terme de calcul à grande échelle et depuis 2004 [16], il a démontré sa capacité de passage à l'échelle sur 132 000 processeurs sur la machine BlueGene/L. A l'heure actuelle, le code tourne de manière journalière sur 100 à 1000 cœurs et est le seul à atteindre 5% de déformation plastique [7]. Un rapport technique [8] ainsi qu'un livre [15], présentent les différents aspects du code tant au niveau de la physique traitée que de la parallélisation et le code est disponible en open source. Les nœuds sont stockés dans une liste chaînée à l'intérieur de chaque domaine. A chaque itération les segments doivent être assemblés à partir des informations contenues dans la structure *Nœud*, et stockés dans le niveau le plus fin de la décomposition régulière de la boîte de simulation. Le parallélisme est basé uniquement sur le paradigme MPI. Le domaine de simulation est découpé en boîte par bissections successives dans chaque direction ce qui permet un équilibrage plus souple pour chaque sous domaine. Ensuite, au cours de la simulation le découpage dans les différentes directions est modifié pour maintenir une charge homogène. Le déséquilibre est mesuré par le temps d'attente aux points de synchronisation lors des échanges de messages. Pour le calcul de la force, la méthode des multipôles rapides appliquée aux dislocations est utilisée [8]. Pour le champ proche, les interactions directes sont calculées analytiquement. Avec le rééquilibrage dynamique le niveau le plus fin de la décomposition évolue mais l'arbre pour l'algorithme FMM est statique avec les cellules réparties à l'initialisation. Ce processus implique une forte phase de communication pour la première étape de la FMM afin de transmettre les contributions des particules à la cellule et la même chose pour la dernière étape en faisant redescendre les contributions du centre des cellules aux segments. En basant le parallélisme sur une décomposition de domaines, une des limitations les plus fortes est de maintenir la charge optimale pour chaque processus. Nous trouvons dans [8], que la meilleure scalabilité est atteinte pour une charge de 200 segments par cœur, ce qui contraint l'utilisation du code pour des calculateurs avec des milliers de cœurs pour atteindre le million de segments. De plus, en partant avec de faibles distributions pour obtenir une forte déformation, il est nécessaire de procéder par *CheckPoint-Restart* fréquents. En effet, afin de permettre de garder la charge optimale ($\simeq$ 200 segments par cœur) dès que la simulation grossit il faut allouer de nouvelles ressources pour pouvoir à nouveau distribuer de manière optimale la charge.

Finalement, un dernier projet Numodis, lancé en 2007 est issu initialement d'une collaboration entre le CNRS et le CEA. Il vise à adopter le modèle de discrétisation nodal et le langage de programmation C++ qui est adapté à la diversité des problèmes traités en DD. Les objectifs du projet sont multiples, avec l'implémentation de différentes cristallographies, des simulations de poly-cristaux, le couplage de code, l'introduction de nouvelles interactions locales, l'utilisation du champs anisotrope, ou encore la gestion des défauts d'irradiation. Au niveau conception, le code a validé la physique simulée à petite échelle avec des comparaisons directes avec les simulations en MD [96], mais il n'est pas destiné aux grandes simulations. Le code utilise une simple liste chaînée et est purement séquentiel. Enfin, le calcul de force est complet et il faut assembler le système linéaire global pour calculer la vitesse des nœuds.

	MicroMégas	**Karlsruhe**	**Paradis**	**Numodis**
Modèle	Vis-Coin-Mixte	Nodal	Nodal	Nodal
Formalisme		Éléments finis	Éléments finis	Éléments finis
Langage	Fortran	Fortran	C	C++
Calcul de force	Méthode des boîtes et cutoff	Calcul complet et cutoff	Méthode multi-pôle rapides	Calcul complet
Calcul de vitesse	Calcul local	Assemblage complet	Assemblage local	Assemblage complet
Structure de données	Tableau avec indexation dans les boîtes	Liste chaînée avec indexation dans les boîtes	Liste chaînée avec indexation dans les boîtes	Liste chaîné avec indexation dans les boîtes
Parallélisme thread	Non	Oui	Non	Non
Parallélisme distribué	MPI	Non	MPI	Non
Équilibrage	Dynamique uniforme	Inexistant	Dynamique non uniforme et Octree statique	Inexistant

TABLE 2 – Récapitulatif des caractéristiques des codes.

La diversité des codes et des implémentations présentées dans le tableau 2 révèle bien le dynamisme de cette thématique de recherche. Comme précisé en introduction, l'inscription de la DD dans le processus multi-échelle est un challenge et pour le moment aucun code à lui seul ne permet d'aborder tous les aspects de ce type de simulation à l'échelle mésoscopique. Ainsi, le domaine est en perpétuelle évolution sur de nombreux fronts. La physique change avec l'introduction de nouvelles lois de mobilité, la prise en compte du champ de contrainte anisotrope afin que l'erreur commise par la simulation soit de plus en plus faible. Sur un plan numérique l'amélioration des méthodes d'approximation du champ de contrainte élastique grâce à de nouvelles méthodes d'interpolation, de nouveaux algorithmes pour détecter les collisions réduit de manière intrinsèque le coût des simulations. Enfin, sur le plan du calcul intensif, la prise en compte des évolutions des architectures de calcul, avec la hiérarchie mémoire, le calcul vectoriel, le calcul parallèle en mémoire partagée et en mémoire distribuée doivent permettre d'entreprendre des études à plus grande échelle.

5 Positionnement

Pour positionner ce travail nous devons prendre en compte l'évolution de la recherche en dynamique des dislocations et la genèse du projet Numodis. La plus grande limitation aux développements de simulation en dynamique des dislocations vient d'une part du coût des calculs associé à une évolution dynamique du système, et d'autre part de la diversité des caractéristiques physiques à intégrer à la simulation pour appréhender tous les phénomènes liés au mouvement des dislocations.

— Pour le challenge lié au coût des calculs, de nombreux travaux ont été entrepris pour diminuer cette pénalité. Dans un premier temps, les évolutions purement matérielles en terme de puissance de processeurs et d'optimisation de compilateurs ont permis d'obtenir de meilleurs modèles 2D puis 3D mais ce sont surtout des améliorations algorithmiques comme le développement de méthodes FMM, qui ont offert la possibilité d'appréhender de nouveaux aspects avec notamment l'étude du comportement à grande échelle des dislocations. Enfin, l'écriture d'algorithmes parallèles a permis d'atteindre une échelle de grain de l'ordre du micromètre avec des densités de dislocations réalistes.

— Pour ce qui est de la diversité des problèmes physiques comme la gestion des joints de grain, des différentes cristallographies, des fautes d'empilement, des phénomènes thermiquement activés, il en résulte une toute aussi grande diversité de codes qui répondent spécifiquement à quelques-uns de ces problèmes.

Dans ce contexte en évolution, vient en 2007 le projet NumoDis suite à deux constats. D'un côté, les limites du modèle de discrétisation vis-coin de TRIDIS qui a les désavantages d'un développement de plus de 15 ans mais surtout de l'autre côté, les nouvelles possibilités des simulations de type nodale ainsi que les promesses des nouveaux paradigmes de programmation pour exploiter les machines hétérogènes massivement parallèles.

L'idée lors du lancement de Numodis n'est pas de créer un n^{ieme} code de dynamique des dislocations mais plutôt de démarrer un projet visant à agréger les dernières avancées dans le domaine, tout en rendant le code open source et suffisamment modulaire pour bénéficier d'un développement collaboratif. Ce projet se devait aussi d'avoir une vision pour l'avenir notamment pour s'adapter aux nouvelles architectures de calcul. C'est donc dans ce cadre là que démarre en 2010 le projet OptiDis soutenu par l'ANR COSINUS. Ce programme qui touche à sa fin avec cette thèse, a produit un nouveau code (baptisé ici **OptiDis**) qui n'est autre que le développement de NumoDis, adapté pour le calcul massivement parallèle. Pour cela trois axes ont été priorisés :

1. l'écriture et l'optimisation d'algorithmes efficaces réduisant intrinsèquement la quantité de calculs par des améliorations de complexité d'algorithmique. On pense notamment à l'utilisation de la méthode de multipôles rapide, des optimisations pour le calcul des intersections, et l'adaptation du schéma de calcul de vitesse purement local tel qu'il est proposé dans Paradis.
2. des développements informatiques pour exploiter de manière performante les architectures de calcul modernes, avec deux priorités. D'un côté, des développements de structures de données ad-hoc, bien adaptées à la dynamiques des dislocations tenant compte des hiérarchies mémoire, et des spécificités algorithmiques. Et d'un

autre côté, des développements pour la parallélisation afin d'exploiter les systèmes de calcul de plus en plus massifs et hétérogènes.

3. Enfin la conclusion du projet est la réalisation de simulations à grandes échelles pour se confronter à l'expérience. Le challenge cible consiste à simuler la formation de bandes claires dans le zirconium irradié. Comme cela à pu être dit, l'application industrielle immédiate permettra de démarrer de nouvelles recherches afin de définir des lois de vieillissement pour les matériaux et ainsi permettre d'allonger éventuellement la durée de vie des assemblages de combustibles dans les centrales nucléaires.

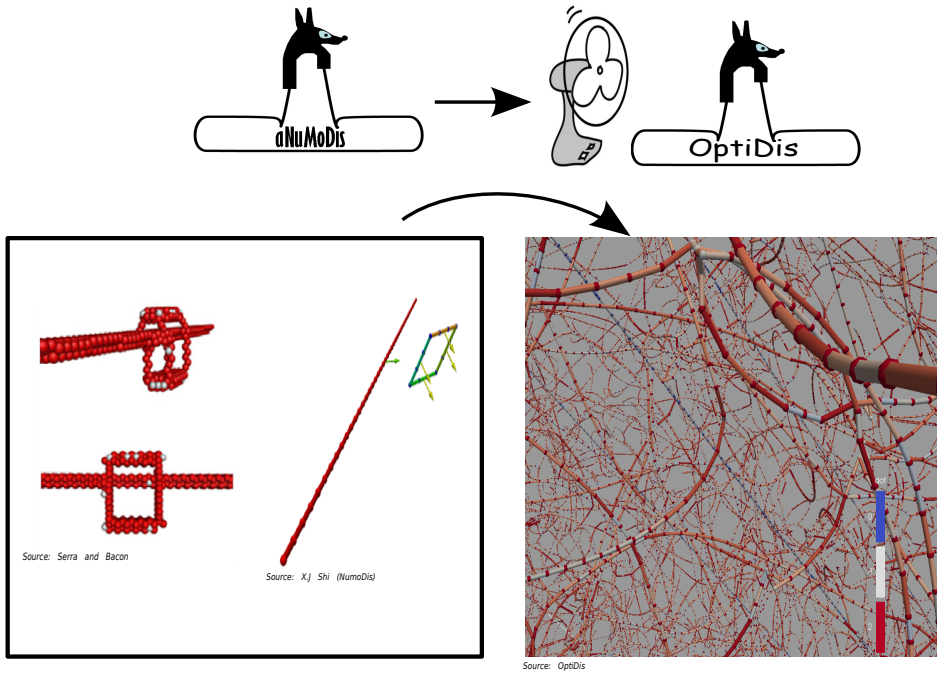


FIGURE 25 – Transition vers les simulations à grande échelle avec le projet OptiDis.

NumoDis est un outil, depuis sa conception, orienté pour la communauté des chercheurs afin de permettre des avancées en travaillant sur un code intégrant de très nombreux développements. Il est écrit avec un design modulaire pour faciliter l'intégration de nouvelles contributions. Il est aussi pensé comme un outil à visée éducative avec un ensemble de tests et autres outils de visualisations pour pouvoir transmettre par l'expérimentation numérique la théorie liée aux dislocations et initier les chercheurs de demain.

Cependant, lorsque qu'un code aussi complet et complexe (plus de 100 000 lignes de codes) est développé sans penser au calcul hautes performances, l'évolution vers ces architectures entraîne d'importantes modifications structurelles. En l'occurrence, avant de démarrer le projet les jalons avaient bien été posés et il était indispensable que ce travail orienté algorithmique parallèle et grands challenges se déroule conjointement avec le travail d'un numéricien. Ainsi, mon travail qui s'adresse directement aux problèmes du passage à l'échelle, a pu se confronter aux grandes simulations suite aux contributions de

Pierre Blanchard avec le développement d'un noyau performant pour le calcul du champ de contrainte élastique par la méthode des multipôles rapides (FMM). La base du projet OptiDis, était de créer une collaboration entre les instigateurs du projet NumoDis, et Inria (équipe HiePACS), pour les compétences en calcul parallèle et l'intégration de la bibliothèque **ScalFMM** avec la méthode des multipôles rapide. Cette bibliothèque est la pièce maîtresse de la simulation puisque environ 85% du temps de calcul est passé dans l'algorithme de calcul du champ de force. Il a donc fallu intégrer ScalFMM puis reformuler l'ensemble des structures de données et des algorithmes autour de cette nouvelle composante pour permettre d'atteindre les grandes simulations.

Dans le suite du document Chapitre 1, nous présentons les contributions apportées au code originel Numodis, avec tout d'abord les développements du code en proposant des solutions algorithmiques pour permettre le passage à l'échelle en partant des algorithmes de Numodis. Après avoir introduit la bibliothèque ScalFMM, nous détaillerons plus précisément les modifications algorithmiques employées pour faire face aux problèmes soulevés, par le calcul du champs de force, la détection des collisions, le schéma de discrétisation en temps et l'adaptation de la topologie au déplacement des dislocations.

Dans le chapitre suivant Chapitre 2, nous présenterons les structures de données utilisées et leur intégration dans les différentes étapes de l'algorithme. Ces structures qui doivent s'imbriquer dans un ensemble hétéroclite d'étapes (calcul de force, calcul de collision et opérations discrètes), et doivent donc notamment associer deux opérations diamétralement opposées. La structure doit permettre de réaliser de manière optimale du calcul intensif avec des accès contigus et cohérents en mémoire, tout en s'adaptant à la modification perpétuelle du réseau de dislocations à chaque itération, le tout en parallèle. Nous présenterons les principes de l'implémentation de ces structures, leur intégration avec la décomposition hiérarchique de ScalFMM et l'ensemble des algorithmes pour manipuler la structure, en nous focalisant sur la correspondance entre la localité spatiale des données et localité en mémoire.

Ensuite Chapitre 3, nous exposerons les problèmes et les solutions pour que le code bénéficie d'un parallélisme hybride performant malgré le dynamisme du système. D'un côté un parallélisme en mémoire partagée, pour bénéficier tant du calcul vectoriel que de l'ensemble des cœurs d'un même nœud, avec une programmation par tâche OpenMP et un mécanisme de liste chaînée sans interblocage adapté aux algorithmes de DD. Un premier niveau auquel nous ajoutons un parallélisme en mémoire distribuée basé sur la décomposition du domaine de simulation selon la courbe de Morton avec l'ensemble des modifications algorithmiques induites et un mécanisme d'équilibrage de charge dynamique.

Enfin, dans la dernière partie Chapitre 4 dédiée aux résultats, nous présenterons les performances du code sur les différents algorithmes dans leur version parallèle et nous confronterons les développements avec la simulation à grande échelle notamment avec le mécanisme de formation de bandes claires sur un matériau irradié.

Deuxième partie

Contributions

Chapitre 1

OptiDis : Les solutions algorithmiques

Sommaire

Une partie conséquente de ce travail de thèse a été consacrée aux développements algorithmiques avec l'idée de faire d'OptiDis non pas un projet de recherche pour le calcul massivement parallèle, mais aussi d'apporter une structure modulaire, évolutive et pérenne pour la recherche en dynamique des dislocations. Pour cela la base de travail de Numodis, avec ses quatre années de développement a été reprise. L'orientation du code développé en C++ a été confirmée et de nombreux choix dans l'architecture du code ont aussi été validés. Cependant avec l'idée de s'orienter vers les grandes simulations, certains choix algorithmiques ont dû être modifiés avec notamment l'intégration de deux nouvelles perspectives.

Tout d'abord, NumoDis est un code séquentiel avec une vision globale des lignes de dislocations ce qui est peu compatible avec l'objectif de réaliser de larges simulations en parallèle. Chaque ligne de dislocation, allant d'un nœud physique à un autre, est stockée dans une liste chaînée et les nœuds de chaque ligne sont eux aussi stockés dans une liste. Dans le cadre d'une parallélisation par décomposition de domaine une telle structure est difficilement maintenable à un coût raisonnable ainsi que la vision par ligne des dislocations. Une première modification fondamentale du code est la suppression de la notion de ligne qui est remplacée par une vision de maillage de type éléments finis avec des nœuds, des segments et des interconnexions entre segments voisins. Cette modification prend notamment toute son importance pour les algorithmes de remaillage avec uniquement un parcours par segment.

Le second point qui dirige tout ce travail concerne l'utilisation de la méthode des multipôles rapide (Fast Multipole Method - FMM) pour calculer le champ de force élastique. Comme introduit précédemment, l'équipe HiePACS s'est positionnée sur ce sujet depuis les travaux de Pierre Fortin [46] pour développer une bibliothèque générique, **ScalFMM**, pour la méthode des multipôles rapide en parallèle [2]. Le calcul du champ de force représentant plus de 85% du temps de la simulation, l'optimisation de ce calcul est donc une étape importante pour la réussite du projet.

Dans ce chapitre nous allons introduire le fonctionnement de la FMM et de la bibliothèque **ScalFMM** pour comprendre son intégration dans le code, tout en introduisant l'ajout inhérent à la bibliothèque, d'un découpage hiérarchique de l'espace par un **octree**, et l'indexation de cette décomposition de l'espace par une courbe de Morton.

1.1 Intégration de ScalFMM dans OptiDis

Comme introduit dans la Section 1.1.1 lors de la description de l'algorithme du calcul du champ élastique, la DD entre typiquement dans le cadre des problèmes à N-corps où chaque interaction met en jeu une paire parmi les N-corps du système. Chercher la solution du système consiste donc à calculer les N^2 interactions à chaque itération. Ces problèmes à N-corps sont fréquemment rencontrés dans la simulation numérique comme par exemple en astrophysique avec les interactions gravitationnelles ou en dynamique moléculaire. Dans ces domaines, il est trop coûteux de réaliser des simulations sur de grands systèmes, c'est pourquoi la recherche s'est penchée sur ces problèmes et deux aspects importants ont été relevés :

— Pour la plupart des systèmes, le principe d'interaction réciproque ($3^{ième}$ loi de Newton) s'applique. Pour 2 particules A et B nous pouvons écrire que la force $F_{A\to B} = -F_{B\to A}$. On évalue donc en une seule fois l'interaction de A vers B ou celle de B vers A, divisant ainsi par deux le nombre d'opérations.
— Sur ces systèmes, le potentiel d'interaction agit à longue portée mais il décroît lorsque la distance entre les particules augmente. On a donc $\lim_{\|\vec{AB}\|\to\infty} F_{A\to B} = 0$.

L'idée de base de la méthode des multipôles rapide est de séparer le champ proche et le champ lointain. Il s'agit d'un algorithme hiérarchique qui repose sur un découpage de l'espace à l'aide d'un octree pour les espaces en trois dimensions. Le découpage se fait en divisant l'espace de manière récursive en deux dans toutes les directions pour former un découpage régulier en boîtes cubiques. Le processus est reconduit sur chaque fils pour obtenir un arbre avec des connexions père/fils, où le dernier niveau correspond aux feuilles, tandis que tout nœud ayant un fils est une cellule. La racine de l'arbre correspond à la totalité de la boîte de simulation qui doit être cubique en 3d (cf. représentation en 2d Figure 1.1). La construction de l'octree nécessite seulement de définir la hauteur H et pour un arbre complet celui-ci contient 8^H feuilles. Dans ScalFMM, l'arbre est construit avec des indirections, [24], pour n'avoir à allouer de l'espace mémoire que pour les sections de l'arbre contenant des segments des dislocations ce qui présente un gain mémoire important pour les distributions hétérogènes comme en DD.

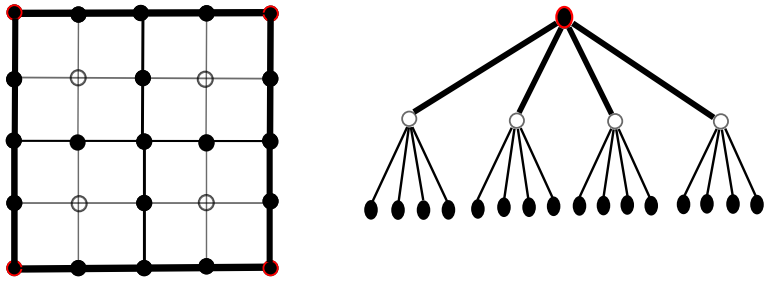


FIGURE 1.1 – Réprésentation d'un *quadtree* dans un espace 2d, avec la correspondance sur le domaine de simulation associé.

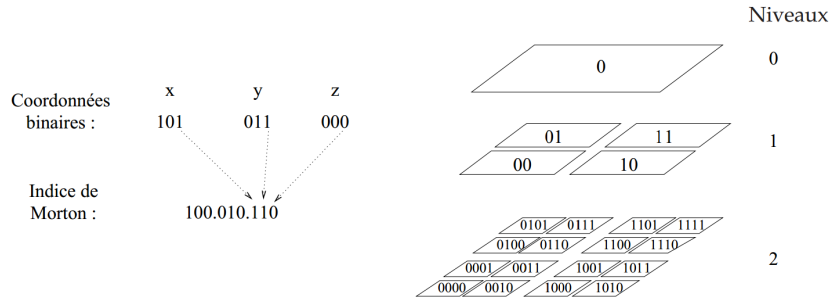


FIGURE 1.2 – Exemple de construction de l'indice de Morton sur un quadtree de hauteur 2. *(Source [46])*

Pour permettre un stockage et un parcours efficace de toutes ces feuilles et cellules ScalFMM utilise la numérotation de Morton. Cette indexage permet de représenter dans une structure de données à une seule dimension, les segments distribués dans un espace en trois dimensions. Les indices de Morton se construisent relativement facilement par décalage de bits à partir du premier niveau comme indiqué dans la Figure 1.2. Cette courbe de Morton permet de prendre en compte la localité des données en construisant des clusters groupés. Cependant, la localité n'est pas parfaite puisque la transition entre deux niveaux dans l'arbre n'est pas toujours continue et entraîne un saut dans le domaine de simulation.

La FMM, pour l'introduire brièvement, repose sur des développements calculés dans les cellules en suivant le découpage en octants (huit sous cubes). L'algorithme pour évaluer le champ lointain se décompose en deux phases avec une phase de montée dans l'arbre et une phase de descente. Deux types de développements sont construits : les développements locaux et les développements multipôles. Les développements multipôles sont construits à partir des feuilles et ils représentent le champ de contrainte dû aux segments d'une cellule dans les cellules "cibles" puis ils sont translatés aux niveaux supérieurs. Enfin, ils sont convertis en développements locaux permettant de déterminer le champ dû aux segments "éloignés" d'une cellule cible. Finalement, ces derniers sont translatés vers les fils jusqu'aux feuilles pour être évalués sur chaque segment.

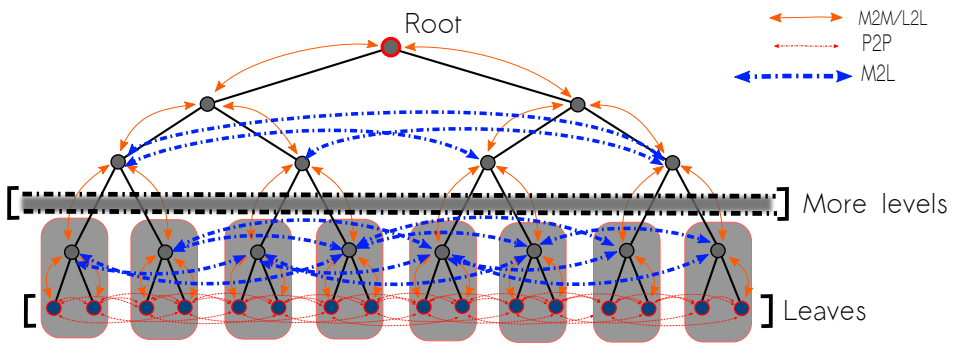


FIGURE 1.3 – Les différentes étapes lors de l'exécution de l'algorithme FMM avec le calcul local au niveau des feuilles ($P2P$), les phases de montée et de descente ($M2M/L2L$) et les phases de transfert ($M2L$).

L'algorithme de la FMM peut être décomposé en plusieurs opérateurs comme présenté sur la Figure 1.3. Nous introduisons ici chacun d'entre eux :

— **P2M** (*Particle to multipole*) calcule le développement multipôlaire à partir des segments contenus dans une feuille ;
— **M2M** (*Multipole to Multipole*) translate le développement multipôlaire des cellules filles vers la cellule père ;
— **M2L** (*Multipole to Local*) convertit un développement multipôlaire en développement local ;
— **L2L** (*Local to Local*) translate le développement local d'une cellule père vers les cellules filles ;

— **L2P** (*Local to Particle*) applique la contribution du champ lointain sur chacun des segments ;
— **P2P** (*Particle to Particle*), cet opérateur effectue le calcul direct entre les segments d'une boîte avec les cellules voisines.

Parmi tous ces opérateurs, le coût de la FMM est principalement dû aux opérateurs *M2L* et *P2P*. Il faut donc choisir la hauteur de l'arbre pour que le calcul du champ proche (*P2P*) soit équilibré avec le calcul du champ lointain. Tant dans la construction de l'arbre que dans le déroulement de l'algorithme, il faut prendre en compte la distribution des particules. Une distribution uniforme permet notamment d'ajuster relativement facilement le coût du calcul entre le champ proche et le champ lointain, tout en sachant que l'arbre sera quasi-complet donc peu d'optimisations sont possibles en terme d'espace mémoire. Si la distribution est non uniforme, il devient nettement plus difficile d'obtenir cet équilibre, aussi d'autres stratégies doivent être mises en place pour sauver de l'espace mémoire et des calculs inutiles sur les zones vides du domaine.

ScalFMM est la brique de base de l'évolution depuis NumoDis vers OptiDis. Nous allons maintenant présenter comment nous avons fait évoluer les différents algorithmes pour d'une part intégrer ScalFMM et d'autre part optimiser les performances des simulations de DD.

1.1.1 Optimisation du calcul de forces

Le calcul du champ de force est la partie coûteuse dans un code de dynamique des dislocations et il existe différentes méthodes pour le réaliser en générant des approximations plus ou moins grandes. Le calcul direct est une solution inenvisageable dès lors que le système comprend quelques milliers de segments. Comme le montre la Figure 1.4 le calcul est quadratique et il domine largement le temps d'une itération passant au delà de la minute à partir de 9 000 segments, soit plus d'un mois de calcul pour 50 000 pas de temps. Pour les petites simulations comme en réalise Numodis lors des comparaisons avec la MD, il est possible d'effectuer le calcul du champ élastique sans employer des méthodes complexes ; voire même, comme cela a été publié par Fitzgerald [42] en utilisant les accélérateurs GPU pour permettre des simulations à quelques milliers de segments.

1.1.1.1 FMM à travers ScalFMM

Pour réaliser de grandes simulations, la méthode hiérarchique que nous utilisons est la méthode des multipôles rapide à travers le moteur ScalFMM. Pour les méthodes FMM, plusieurs formulations ont été écrites dans le noyau élastique des dislocations. La formulation initiale de ParaDis [8] est basée sur des développements en séries de Taylor. Par la suite plusieurs autres formulations sont apparues. Avec le travail de Pierre Blanchard, doctorant dans l'équipe, une approche générique de la FMM par des méthodes d'interpolation basées tout d'abord sur les polynômes de Chebychev puis sur une grille de points régulièrement espacés ont pu être intégrées à notre code. Ces développements sont en dehors du cadre de cette thèse mais nous pouvons tout de même souligner que cette dernière formulation, la plus performante, permet notamment d'interpoler finement la contribution d'un segment à cheval sur deux boîtes.

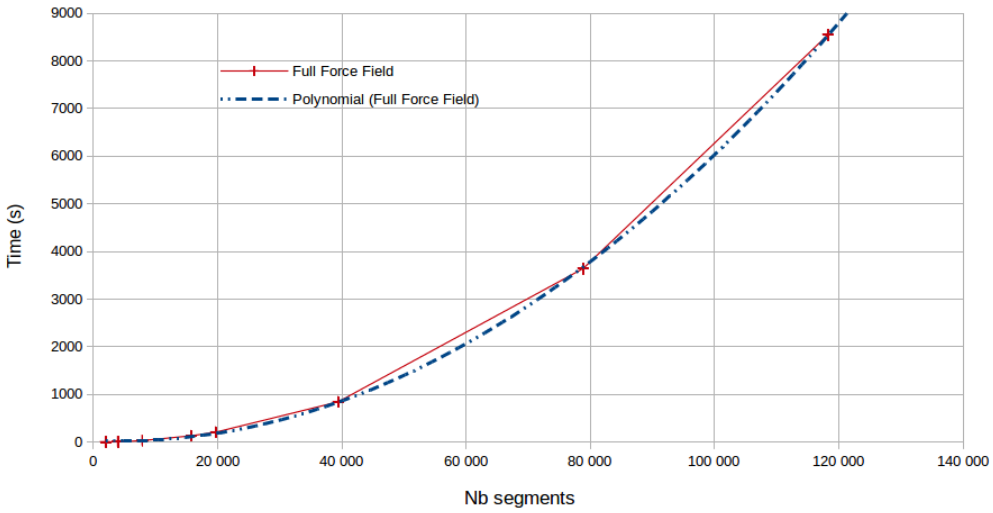


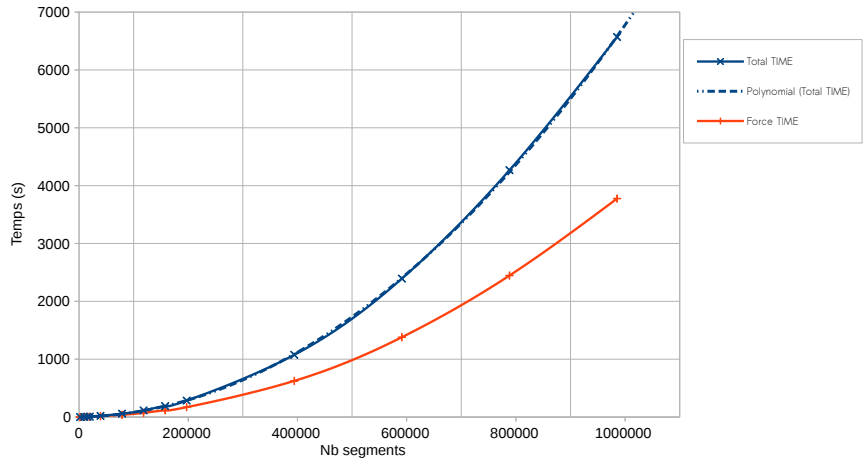
FIGURE 1.4 – Évolution du coût du calcul direct en fonction du nombre de segments dans la boîte. L'approximation polynomiale est associée $y(x) = 6.55\,10^{-7}\,x^2 - 0.0052\,x + 32.87$.

Les segments peuvent appartenir à plusieurs feuilles, la séparation entre le champ proche et le champ lointain n'est plus exacte. Cette formulation par interpolation ajoute la notion de boîtes étendues pour un contrôle plus fin de l'erreur. En interpolant les segments dépassants la feuille, sur une grille uniforme nous sommes capables de contrôler cette erreur à condition que le débordement ne soit pas supérieur à une certaine limite. Un segment est alors affecté à une seule et unique feuille étendue. Pour contrôler le débordement nous affectons le segment par son point milieu et non plus par une extrémité. Ainsi, il existe une relation entre la taille du segment et la taille de la feuille le contenant. Cela implique que nous ne pouvons pas choisir une hauteur maximums d'arbre sans prendre en considération la discrétisation des segments et la taille du domaine de simulation.

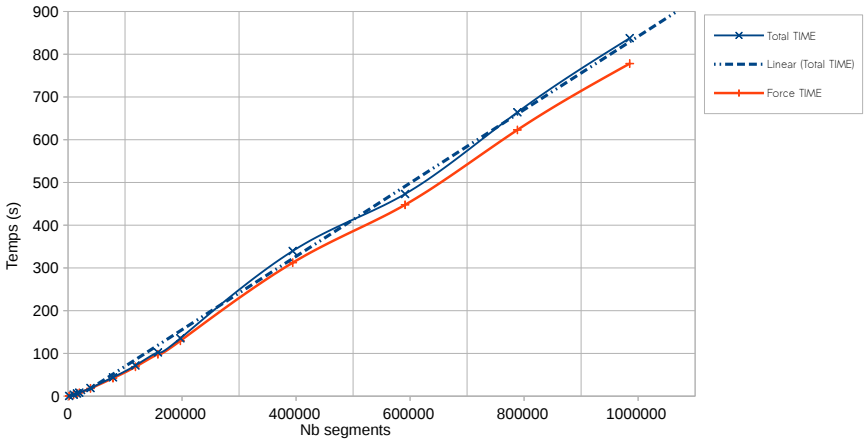
Densité boucles/m^3 \ **Côte (Å)**	10^3	$1,5.10^3$	2.10^3
	Nb boucles/Nb Segments	Nb boucles/Nb Segments	Nb boucles/Nb Segments
1.10^{20}	94/1 128	324/3 888	776/9 312
5.10^{20}	470/5 640	1 620/19 440	3 880/46 560
1.10^{21}	941/11 292	3 240/38 880	7 760/93 120
5.10^{21}	4 703/56 436	16 202/194 424	38 800/465 600
1.10^{22}	9 406/112 872	32 405/388 860	77 600/931 200
5.10^{22}	47 031/564 372	162 024/1 944 288	388 001/4 656 012
1.10^{23}	94 061/1 128 732	324 047/3 888 564	776 003/23 275 000

TABLE 1.1 – Nombre de boucles d'irradiation et nombre de segments obtenus selon la densité($boucles/m^3$) et le volume de la boîte cubique dont la longueur côté est donnée, pour des segments discrétisés à 45Å (12 segments/boucle).

Le coût de la FMM est incompressible mais il faut cependant calibrer l'algorithme pour que celui-ci soit le plus efficace possible. Toute l'analyse de notre implémentation menée ici est purement séquentielle. Nous utilisons une configuration, avec une augmentation de la densité de boucles d'irradiation dans un grain cubique de zirconium comme indiqué dans le Tableau 1.1. Nous choisissons ici, de faire une étude jusqu'à une densité réaliste de 5.10^{22} boucle/m^{-3}, soit 1 000 000 de segments dans un cube de $12\mu m^{-3}$, ce qui correspond à l'état de l'art en terme de simulation de DD. Dans un premier temps, nous pouvons



(a) Temps du calcul de la force (FMM) comparé au temps d'une itération complète. Hauteur de l'octree $H = 4$ (4 096 feuilles maximum).



(b) Temps du calcul de la force (FMM) comparé au temps d'une itération complète. Hauteur de l'octree $H = 6$ (262 144 feuilles maximum).

FIGURE 1.5 – Importance du coût de calcul de la force (FMM) sur le coût d'une itération complète.

observer dans la Figure 1.5a et la Figure 1.5b le comportement de la simulation lors du calcul du champs élastique en fonction du nombre de segments et de la hauteur de l'arbre, respectivement $H = 4$ et $H = 6$. Dans un cas idéal, le temps total de la simulation croît de manière linéaire avec l'augmentation du nombre de segments du fait de la complexité de la FMM en $\mathcal{O}(N)$. Puisque l'itération est gouvernée par le calcul du champ de force, le

comportement linéaire optimal que nous observons pour la Figure 1.5b avec une hauteur 6 jusqu'à 1 000 000 de segments, s'explique par un bon équilibrage du coût calculatoire entre le champ proche et le champ lointain. Quelque soit la hauteur, à partir d'une certaine

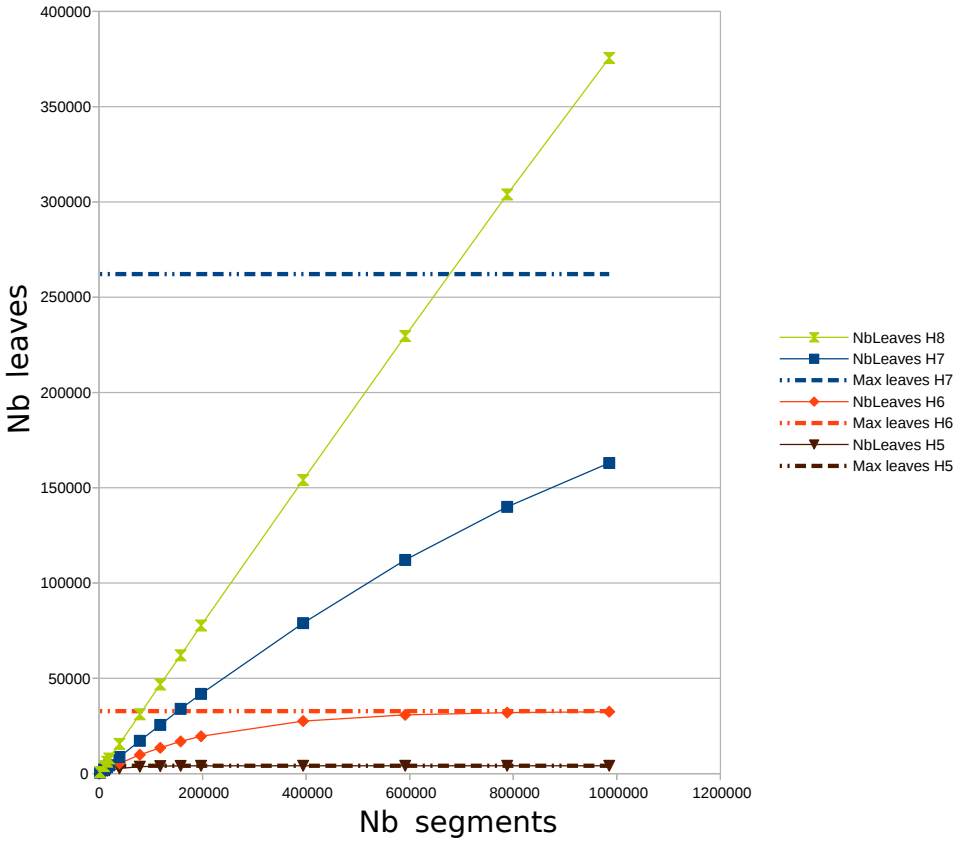


FIGURE 1.6 – Évolution du nombre de feuilles allouées selon la hauteur de l'arbre pour un cas homogène de boucles de dislocations induites par irradiation sur un cube de zirconium.

densité de segments répartis de manière homogène dans le grain, les feuilles vont être saturées. A partir de là, le calcul du champ proche qui reste quadratique avec les feuilles voisines va dominer le calcul comme le montre le temps de calcul pour une hauteur 4 dans la Figure 1.5a. Cela est confirmé en regardant la Figure 1.9 qui présente le remplissage des feuilles selon la densité de défauts. On constate que seul l'octree avec une hauteur de 6 parvient à absorber la charge en allouant de nouvelles feuilles et donc en limitant le coût du champ proche. A l'inverse pour une trop grande hauteur à partir de 7, le calcul de champ lointain domine le coût de l'algorithme et la contribution du champ proche devient minimale.

Au final, une fois le choix de la profondeur de l'arbre effectué, puisque le calcul du champ lointain est incompressible et dépend uniquement de la méthode choisie ainsi que de sa précision, nous pouvons alors paramétrer la fréquence de mise à jour du champ lointain en couplant le calcul FMM avec un calcul uniquement sur les voisins directs (i.e. champ proche) à chaque itération. Cette optimisation additionnelle est détaillée dans la section suivante.

1.1.1.2 Une méthode couplée adaptative

Pour réduire le coût de calcul des forces nous couplons la FMM avec une méthode de rayon de coupure ou *Cutoff method*. Comme le champ élastique est décroissant en $\frac{1}{r}$, la majeure partie du potentiel agissant sur une nœud provient des segments les plus proches. Il faut alors déterminer un rayon de coupure r_c pour que toutes les contributions éloignées, telles $r > r_c$ soient ignorées. Dans la pratique avec le découpage hiérarchique régulier de l'octree, r_c n'a pas besoin d'être déterminé explicitement. Nous considérons les voisins directs de la feuille cible comme montré sur la Figure 1.7 soit les 26 feuilles voisines dans notre domaine 3d. La longueur du rayon de coupure dépend donc de la taille de la boîte de simulation et de la hauteur de l'arbre. Lorsque le rayon de coupure est trop petit et/ou

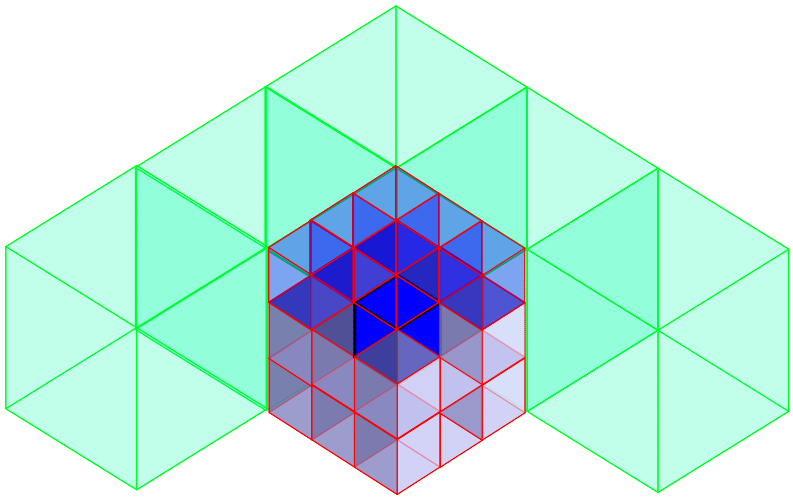


FIGURE 1.7 – Découpage de l'espace, le cube bleu avec les arêtes noires interagit avec les 26 $(3^3 - 1)$ cubes aux arêtes rouges, au delà les contributions dans les cubes verts ne sont pas considérées.

la distribution est très hétérogène, utiliser uniquement cette méthode de rayon de coupure amène à négliger beaucoup de contributions et donc introduit une forte approximation dans le calcul de force. Typiquement, des actions fortes de répulsion ou d'attraction entre deux segments très proches ne se produiront pas de la même façon en prenant en compte la contrainte élastique exercée par la totalité du réseau de dislocations.

La solution que nous adoptons est d'utiliser un processus adaptatif visant à moyenner le coût du calcul FMM global et l'erreur générée par l'approche avec le rayon de coupure. Nous calculons avec la FMM les contributions venant des boîtes lointaines tous les n pas de temps et nous appliquons le même champ lointain sur les n itérations suivantes. Comme la simulation est très dynamique, cette méthode doit donc être bien calibrée en adaptant

dynamiquement la fréquence, n, du calcul FMM pour ne pas engendrer une erreur trop importante. Pour évaluer l'erreur nous mesurons la variation entre le champ lointain que l'on vient de calculer (itération n_c) et celui calculé à l'itération $n_c - n$. Pour cela à chaque nœud on mesure la variation du champ lointain : $V_f = ||F_f^{n_c}|| - ||F_f^{n_c-n}||$. Cette variation est ensuite moyennée sur l'ensemble des nœuds du maillage. Si la variation est supérieure à un seuil de tolérance, la fréquence du calcul FMM est réduite pour améliorer la précision du calcul ($n = n \times 0.8$). Si le champ lointain n'a pas évolué le calcul FMM est effectué moins fréquemment en augmentant n ($n = n \times 1.2 + 1$). Le choix du facteur d'incrémentation est empirique et permet de suivre l'évolution du calcul.

Par exemple, dans les cas avec une contrainte appliquée forte entraînant une variation rapide du champ lointain et avec une distribution très hétérogène des segments, le calcul FMM complet est effectué à chaque itération ($n = 1$). Il faut attendre l'homogénéisation de la distribution sur tout le domaine de simulation avec le phénomène d'écrouissage pour avoir une évolution plus lente du champ lointain et pouvoir augmenter la fréquence.

De plus, pour chaque nœud la contribution du champ lointain C_f par rapport à la contrainte globale est mesurée.

$$C_f = \sum_{i=0}^{N} \frac{F_f^t(i)}{F_{app}^t(i) + F_{el}^t(i) + F_{core}^t(i)}.$$

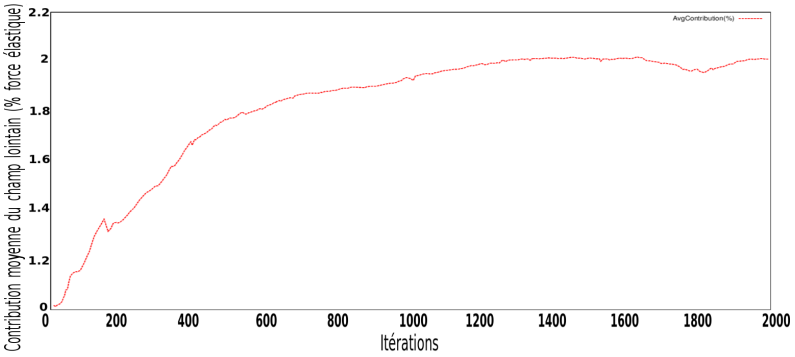
où pour chaque nœud i, $F_f(i)$ est la contribution du champ lointain, $F_{app}(i)$ la force appliquée, $F_{el}(i)$ la force élastique et $F_{core}(i)$ le force de coeur.

Plus cette contribution est importante plus la fréquence n est réduite pour diminuer l'erreur sur le champ global. Les simulations suivantes (Figure 1.8) montrent que, au cas par cas et durant une même simulation, le scénario est très différent. Il faut alors bien régler la sensibilité de l'algorithme pour adapter la fréquence de calcul du champ de force complet.

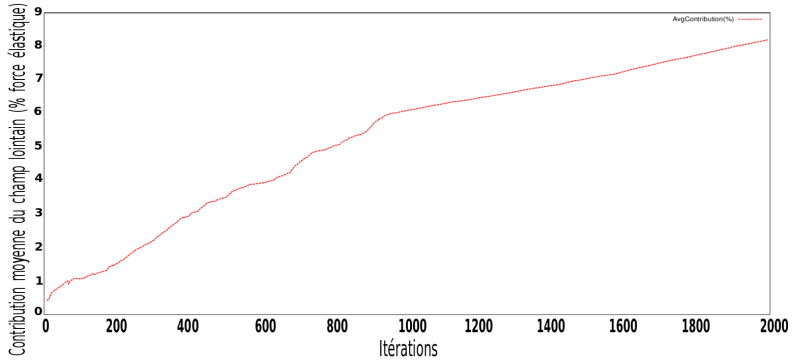
Les courbes de la Figure 1.8 montrent l'évolution de la contribution moyenne du champ lointain sur les nœuds. On constate que selon le type de défauts (boucles d'irradiation ou des sources de Frank-Read) et aussi selon la densité de défauts et le type de contrainte appliquée, pour une même taille de domaine et une même hauteur d'arbre, la contribution des segments en dehors des boîtes voisines ne représente pas la même proportion sur la totalité du champ de force et évolue au cours de la simulation.

Sur la Figure 1.8a représentant un grain contenant des défauts d'irradiation donc très statiques et répartis de manière homogène, la contribution n'est que de 2% au maximum et évolue peu. Pour le même cas, mais en augmentant la contrainte appliquée (Figure 1.8b), la contribution est croissante car la force appliquée est supérieure à la contrainte critique d'activation des boucles, générant ainsi une contrainte élastique interne croissante pour atteindre 10% du total de la force reçue par les segments.

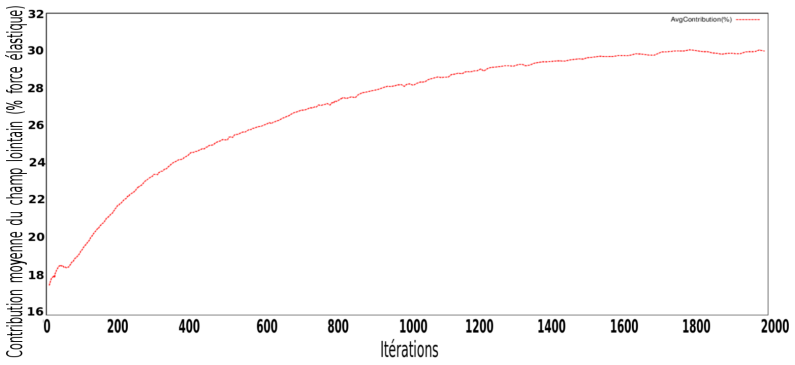
Pour un grain contenant des sources de Frank-Read (Figure 1.8c) et une force appliquée de $\sigma = 90MPa$, l'activation progressive des lignes avec leur déplacement rend la distribution de plus en plus hétérogène et fait donc croître l'importance de la contribution des segments éloignés. Dans les graphiques de la Figure 1.10, nous montrons l'évolution de la fréquence n de calcul du champ lointain. Cette évolution de la fréquence est aussi à corréler avec les courbes de la Figure 1.8 qui sur les mêmes simulations mettent en



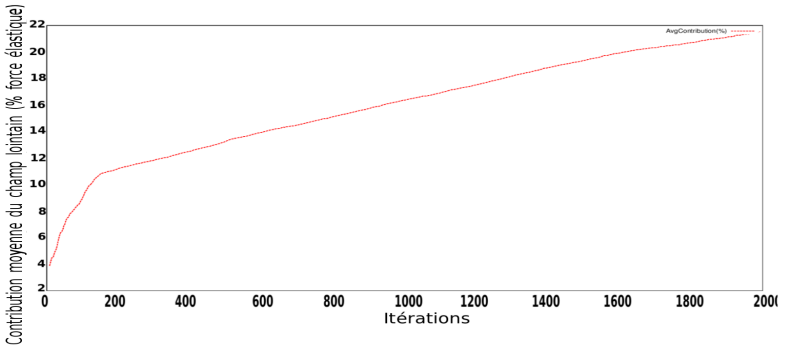
(a) Simulation de boucles d'irradiations (densité de boucle $1.10^{20}m^{-3}$) contrainte constante $\sigma = 25MPa$.



(b) Simulation de boucles d'irradiations (densité de boucle $1.10^{20}m^{-3}$) contrainte constante $\sigma = 270MPa$.



(c) Simulation de sources de Frank-Read avec une contrainte constante $\sigma = 90MPa$.



(d) Simulation de sources de Frank-Read avec une contrainte constante $\sigma = 500MPa$.

FIGURE 1.8 – Évolution de la contribution venant des segments en dehors des premiers voisins pour une hauteur d'arbre H=6, en pourcentage de la contrainte totale selon le mode de chargement et le type de défauts.

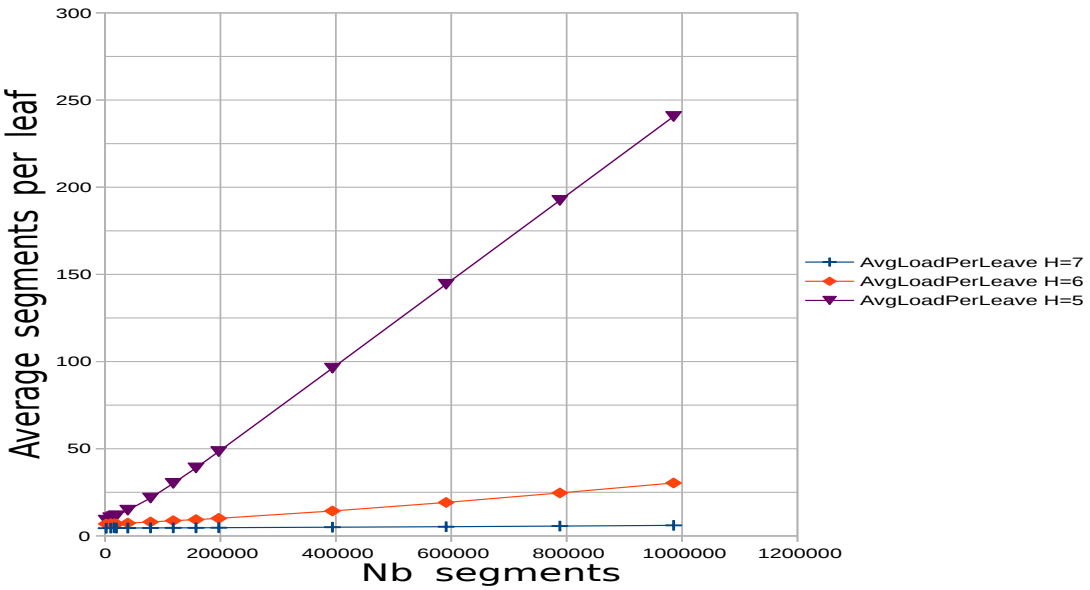
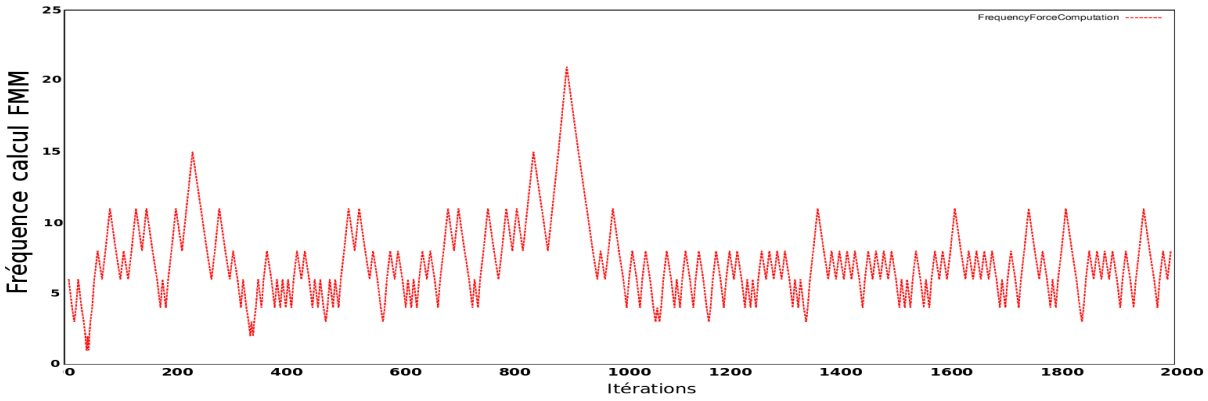


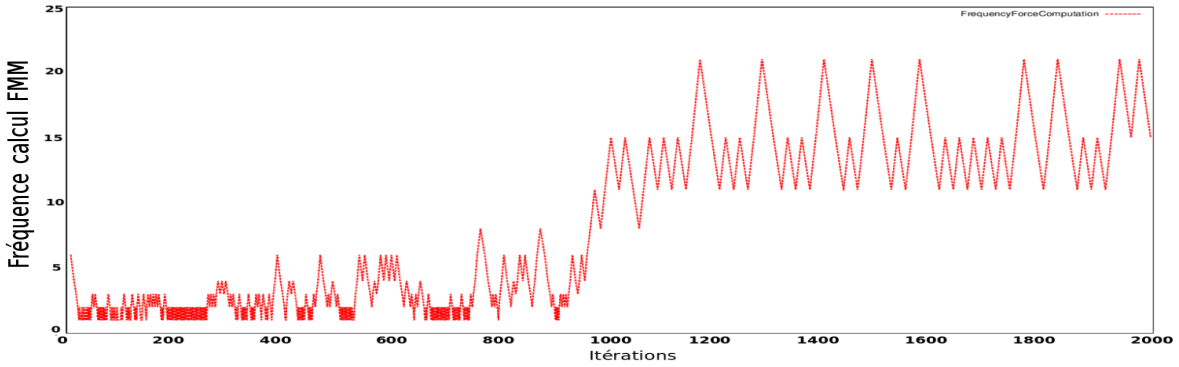
FIGURE 1.9 – Charge moyenne par feuille selon la hauteur et le nombre de feuilles allouées, pour un chargement homogène de boucles de dislocations induite par irradiation d'un cube de zirconium.

évidence la part de la contribution du champ lointain sur le champ total. Cette étude est effectuée en considérant une hauteur d'arbre qui équilibre le coût du champ lointain et du champ proche. Pour tous les cas présentés ici, le critère d'erreur maximal à ne pas dépasser est que la variation moyenne entre deux calculs du champ lointain ne soit pas supérieure à 2% en norme. Cette valeur empirique est acceptable pour la validité des résultats comparés à un calcul exact.

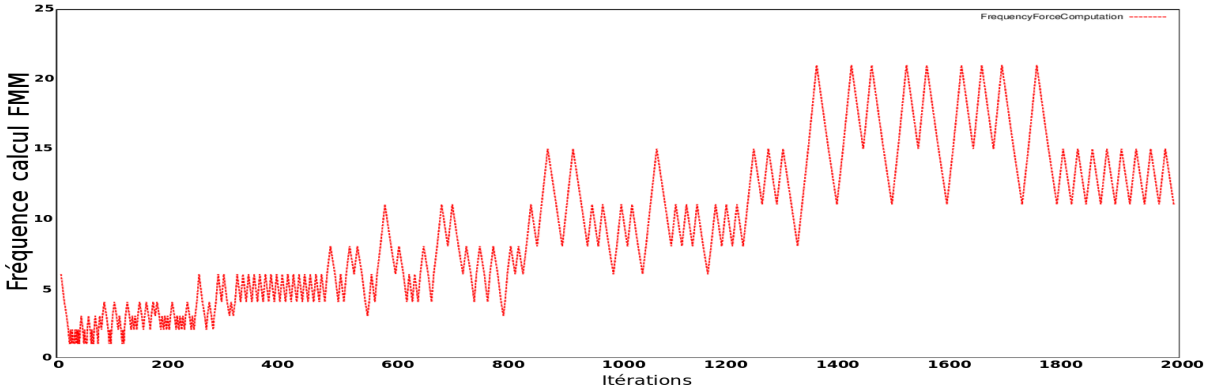
Le premier constat est que l'algorithme adapte la fréquence de calcul selon que la contrainte appliquée est forte ou pas. Par exemple, si les défauts sont de petites boucles d'irradiation en faible densité (Figure 1.10b), sur les premiers pas de temps la contrainte ($\sigma = 270MPa$) provoque l'activation des boucles. Ainsi, la simulation est extrêmement dynamique, et hétérogène au début, puis les boucles vont alors remplir le grain et rapidement rendre celui-ci beaucoup plus dense en défaut et par conséquent plus homogène. Nous avons alors une forte partie du champ de force qui peut être récupérée par le champ proche tandis que le champ lointain plus homogène évolue moins et a besoin d'être moins fréquemment mis à jour avec en moyenne toutes les 15 itérations, ce qui conduit à un gain de temps de calcul important. Pour les source de Frank-Read sous une forte contrainte appliquée ($\sigma = 500MPa$ Figure 1.10d), la même interprétation peut être faite, mais le procédé est plus régulier puisque les sources sont par définition très mobiles. Pour les deux autres cas, avec de faible contrainte ; le cas des boucles ($\sigma = 25MPa$ Figure 1.10a) doit être interprété comme un cas statique et hétérogène du fait de la faible densité. La fréquence de calcul du champ lointain reste donc constante tout au long des 2 000 itérations. Enfin, dans le cas avec les sources de Frank-Read soumises à une contrainte appliquée de $90MPa$ (Figure 1.10c), l'homogénéisation de la répartition des défauts est plus lente que le cas avec une contrainte de $500MPa$, mais les sources étant des objets très mobiles nous



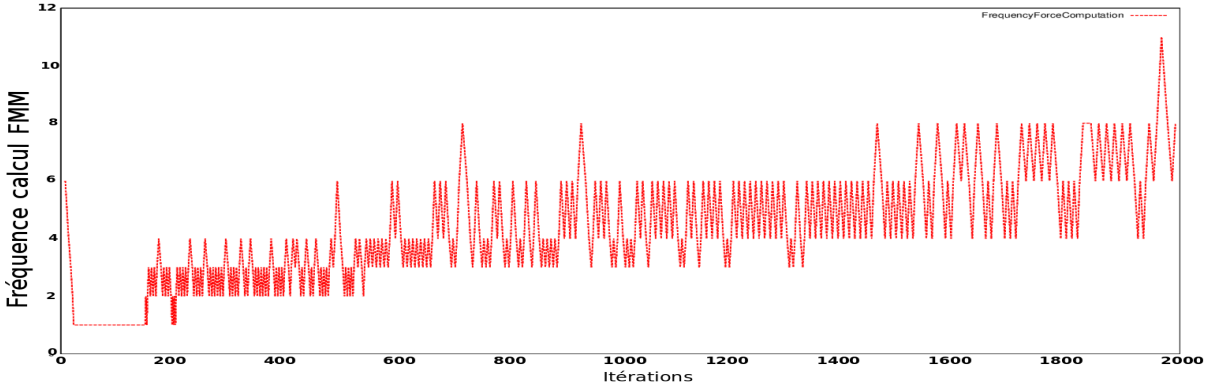
(a) Simulation de boucles d'irradiation (densité de boucle $1.10^{20}m^{-3}$) contrainte constante $\sigma = 25MPa$).



(b) Simulation de boucles d'irradiation (densité de boucle $1.10^{20}m^{-3}$) contrainte constant $\sigma = 270MPa$.



(c) Simulation de sources de Frank-Read avec une contrainte constante $\sigma = 90MPa$.



(d) Simulation de sources de Frank-Read avec une contrainte constante $\sigma = 500MPa$.

FIGURE 1.10 – Évolution de la fréquence de calcul du champ lointain pour maintenir la variation sa moyenne à moins de 2% pour une hauteur H=6.

parvenons tout de même à avoir une variation du champ de force lointain de plus en plus faible d'où une augmentation de n, la fréquence du calcul FMM.

Nous voyons qu'en paramétrant notre hauteur d'arbre nous pouvons équilibrer le coût du calcul du champ proche et du champ lointain, mais aussi régler la part de la contribution entre le champ proche et le champ lointain. Dans les simulations avec une faible densité, il est difficile d'augmenter l'écart entre deux calculs du champ lointain du fait que l'hétérogénéité de la distribution augmente la part du champ lointain dans la contribution totale. De plus, avec moins de défauts, le champ lointain varie plus du fait de la forte mobilité des lignes qui rencontrent peu d'obstacles. Pour les simulations à très fortes densités, la contribution du champ lointain devient minime par rapport à celle du champ proche et évolue de moins en moins. Dans un grain rempli avec une densité de défauts de l'ordre de $10^{23}m^{-3}$, la contribution du champ lointain sur le champ élastique total, est de moins de 1%. De plus, dans ce cas la présence de défauts immobiles comme les boucles d'irradiation réduit la variation de ce champ lointain au cours des itérations. Ces deux aspects permettent de réduire le coût de ce calcul de manière forte en augmentant la période entre deux mises à jours du champ lointain.

1.1.2 Optimisation de la gestion des collisions

1.1.2.1 Détection des collisions

La détection des collisions est le second noyau de calcul possédant une forte intensité arithmétique. De l'algorithme original de NumoDis la stratégie des boîtes englobantes a été conservée mais son intégration dans le code a été modifiée. Principalement, nous utilisons le découpage hiérarchique de l'espace avec l'octree lié à la FMM pour réduire le nombre de tests entre les segments. Les segments sont contenus dans des boîtes qui correspondent aux feuilles de l'arbre et nous testons uniquement les collisions avec les objets contenus dans les boîtes voisines. ScalFMM est une bibliothèque générique pensée pour pouvoir utiliser différents noyaux de calcul. On change alors le noyau de calcul de la contrainte élastique par un noyau de détection de collisions pour intercepter toutes les collisions entre les segments proches et on ne considère que le champ proche en désactivant les opérateurs du champ lointain (*P2L*, *M2L*...). Ainsi l'algorithme de calcul du champ proche (*P2P*) de la FMM décrit précédemment (Figure 1.7), s'applique ici aussi de la même façon en changeant le noyau.

Cependant comme l'algorithme teste uniquement les collisions avec les boîtes voisines pour nous assurer de détecter toutes les collisions, il faut imposer qu'un segment ne puisse pas au cours d'un pas de temps entrer en contact avec un segment venant d'une boîte au delà des premiers voisins. En effet, comme le montre la Figure 1.11 si le déplacement de chaque segment est supérieur à la moitié d'une boîte, des segments considérés comme trop éloignés par l'algorithme peuvent entrer en collision sans pour autant être traités comme tel.

Comme nous affectons un segment à une boîte par son point milieu, la moitié du segment au maximum déborde dans une boîte voisine. Pour les cas pathologiques où le point milieu est sur un coin de la boîte le raisonnement reste identique et chaque moitié de segment qui déborde de sa boîte d'affectation et peut être interpolée dans la boîte

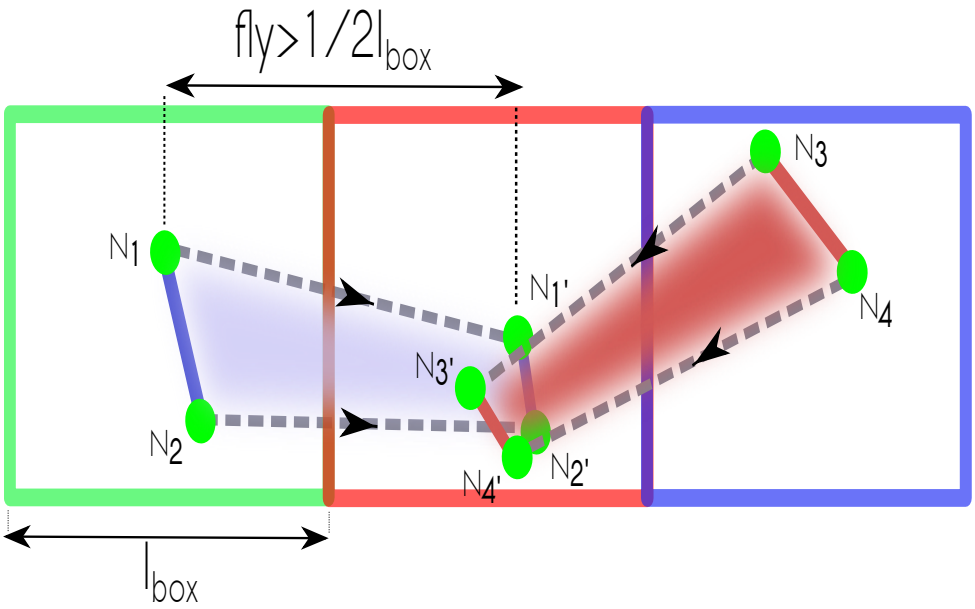


FIGURE 1.11 – Contrôle du vol d'un segment. Si le déplacement ($N_i' = N_i + \Delta t V_i$) est supérieur comme c'est la cas ici, cette collision ne sera pas détectée. $N1$ et $N2$ (Resp. $N3$, $N4$) sont les nœuds du segments 1 (Resp. 2) à t_0 puis $N1'$ et $N2'$ (Resp. $N3'$, $N4'$)la position à t_1.

étendue. Suite au déplacement des nœuds nous avons pour chaque nœud N_i :

$$|N_i N_i'| = \Delta t |V_i| \,,$$

où $|N_i N_i'|$ est la longueur du déplacement du nœud N_i, V_i la vitesse constante sur le pas de temps Δt. En connaissant le déplacement maximal et pour être sûr qu'un segment n'intersecte pas un autre segment au delà du premier voisin, nous contraignons la longueur maximale, L_{max}, de discrétisation des segments. Elle sera toujours deux fois plus petite que la largeur d'une boîte ($l_{box} = L8^{-H}$ avec L la longueur du côté du cube englobant le domaine de simulation). Nous obtenons ainsi une relation liant la discrétisation des segments et la taille des boîtes soit

$$2L_{max} < l_{box}.$$

Le paramétrage de la simulation met en relation la taille de la boîte de simulation, la hauteur de l'arbre et les critères de discrétisation des segments. On peut lire par exemple dans le Tableau 1.2 (ligne 2), que pour un grain de $10\mu m$ de côté avec une subdivision en 64 boîtes par direction pour $H = 6$, il faut prendre un critère de discrétisation maximal vérifiant $L_{max} \leq 78.1$Å. Dans les simulations avec des défauts d'irradiation, la discrétisation habituelle est $L_{max} = 55$ Å pour les boucles d'irradiations. Pour $H = 6$, qui correspond à la hauteur équilibrant le coût du calcul de force entre champ proche et champ lointain, seul un petit grain de 5000 Å de côté nécessite d'augmenter la précision de la discrétisation habituellement utilisée.

Box Size (Å)	H=4	H=5	H=6	L_{max} $(H = 6)$
5000	312,5	156,25	78,125	39,0625
10000	625	312,5	156,25	78,125
12000	750	375	187,5	93,75
15000	937,5	468,75	234,375	117,1875
18000	1125	562,5	281,25	140,625
20000	1250	625	312,5	156,25

TABLE 1.2 – Taille des feuilles en Angström (Å) dans l'arbre en fonction de la taille de la boîte de simulation et la hauteur de l'arbre H. Le dernière colonne donne la longueur L_{max} pour la hauteur d'arbre $H = 6$.

Nous présentons maintenant les deux types de collisions possibles soit avec la microstructure soit entre segments.

1.1.2.1.1 Collisions entre segments et microstructure. A chaque fin d'itération, suite au déplacement des segments nous insérons de nouvelles feuilles et supprimons les feuilles vides dans l'octree. Avant d'insérer une feuille, on teste si des éléments de la microstructure (arêtes, faces) l'intersecte pour savoir si cette feuille contient une partie de la microstructure. A la fin de cette étape, nous récupérons une liste des feuilles de l'octree contenant à la fois des segments et une partie de la microstructure. Cette liste est mise à jour à chaque fin de pas de temps.

Nous considérons deux types d'obstacles : les collisions sur les bords du grain et les collisions sur les précipités. Au final, soit les nœuds entrent en collision avec une surface, soit les segments avec un coin d'un précipité. L'Algorithme 3 détaille les étapes où nous itérons sur la liste de feuilles préalablement sélectionnées lors de la mise à jour de l'octree et nous testons pour chaque segment contenu dans ces feuilles les collisions avec les coins ou avec les surfaces.

Algorithme 3 : Algorithme de détection des collisions entre les segments et la microstructure.

Données : Segment segment1

```
1 pour chaque Box in ListOfBox faire
2     pour chaque Segment in Box faire
3         pour chaque Edge in Box faire
4             TestEdgeCollision(Segment, Edge);
5         pour chaque Face in Box faire
6             TestFaceCollision(Segment, Face);
```

Si un nœud entre en collision avec une face de la microstructure, on stoppe son déplacement et on met à zéro sa vitesse. Cela permet de considérer le nœud de manière différente pour le remaillage et notamment de le supprimer si le segment est aligné et coincé sur le bord. On supprime ainsi des degrés de liberté inutiles à la simulation.

Dans le cas où un segment entre en collision avec le coin d'un précipité, un nœud est inséré au niveau du coin du précipité et ce nœud reçoit un *flag* temporaire (booléen) pour ne pas être supprimé lors du remaillage et éviter ainsi qu'un segment puisse pénétrer dans un précipité.

De manière générale, le test de collision avec la microstructure n'a lieu que sur un faible nombre de feuilles. Il est donc très efficace et facilement parallélisable et représente moins de 1% du temps passé pour calculer les collisions entre segments.

1.1.2.1.2 Collisions entre segments. La détection des collisions entre les segments est plus coûteuse. Notons $C_{p2p}(a)$ la complexité de la détection pour la boîte a et $V(a)$ la liste des premiers voisins de la feuille a. Nous avons alors

$$C_{p2p}(a) = \alpha N_a^2 + \beta \sum_{b \in V(a)} N_a N_b$$

où N_a le nombre de segments dans la feuille a. Le coût total est donc

$$\sum_a C_{p2p}(a) = \alpha \sum_a N_a^2 + \beta \sum_a \sum_{b \in V(a)} N_a N_b.$$

Notons γ le nombre maximum de segments par feuille alors

$$\sum_a C_{p2p}(a) \leq \alpha\gamma \sum_a N_a + \beta\gamma \sum_a \sum_{b \in V(a)} N_a$$

avec au maximum 27 voisins et on a $\sum_a N_a = N$ le nombre total de segments on a

$$\sum_a C_{p2p}(a) \leq \alpha\gamma N + 27\beta\gamma N$$

d'où

$$\sum_a C_{p2p}(a) \leq (\alpha + 27\beta)\gamma N = C_1 \gamma N.$$

De cette complexité on comprend que deux paramètres γ et C_1 vont nous permettre de réduire ce coût. Tout d'abord, le paramètre γ évolue en ajustant la hauteur de l'arbre. En augmentant cette hauteur on réduit le rayon de coupure et donc la zone où chercher les intersections et par conséquent le nombre de paires de segments à tester. Pour une boîte de 15 000 Å de côté, le nombre de segments par feuille pour une distribution uniforme peut être maintenu faible en ajustant la hauteur. Dans la Figure 1.9, pour $H = 6$, et 1 000 000 de segments, nous voyons que le nombre de feuilles allouées est seulement de la moitié de la capacité de l'arbre et nous maintenons une faible charge par feuille. Ce premier point permet de réduire fortement le coût de détection de collision. Pour des cas importants comportant jusqu'à 1 millions de segments, nous maintenons un comportement linéaire pour le nombre de tests d'intersection entre segments à partir d'une hauteur $H = 6$. De plus, à partir d'un grain de $12\mu m^3$ un niveau peut être ajouté car même avec une hauteur supérieure $H = 7$ la longueur L_{max} est suffisamment grande pour ne pas ajouter artificiellement des segments supplémentaires.

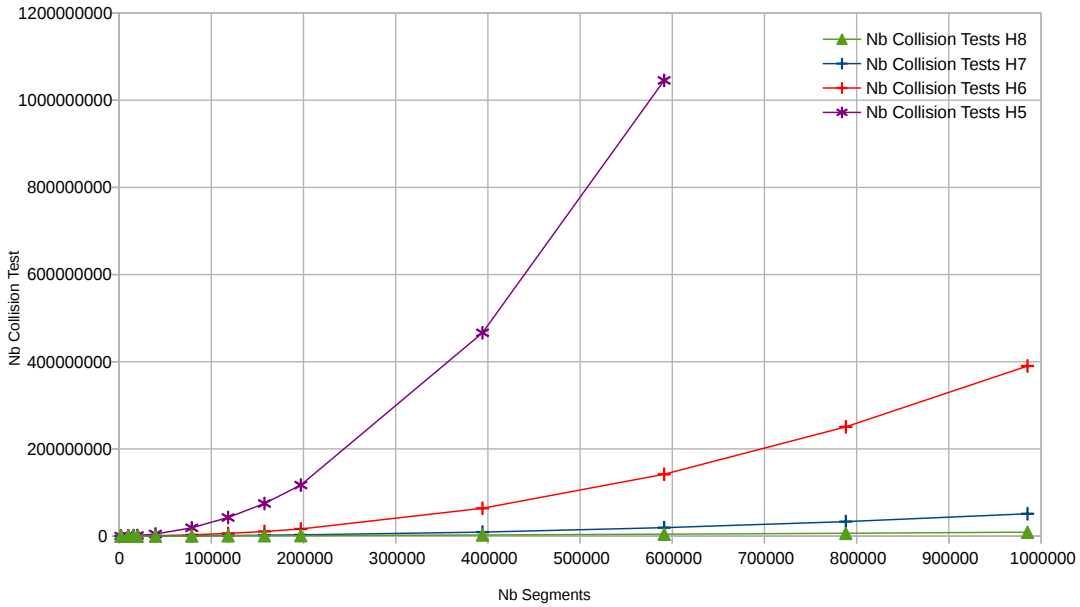


FIGURE 1.12 – Nombre de tests de collisions à effectuer en fonction du nombre de segments et de la hauteur de l'arbre.

Le paramètre C_1 correspond au coût du test de collision entre une paire de segments. Pour l'optimiser on se base sur les propriétés géométriques des déplacements. En effet, les segments se déplacent dans leur plan de glissement et donc le premier test que nous effectuons est de savoir si deux segments appartiennent au même plan comme introduit dans l'Algorithme 4 ligne 1. Si les deux segments appartiennent au même plan nous pouvons considérer l'intersection comme l'intersection de deux objets dans un espace 2d. Si en plus, ces plans sont parallèles nous rejetons facilement la collision (ligne 7 de l'Algorithme 4). Dans le cas où les segments sont inter-connectés, c'est à dire ayant un nœud en commun le test correspond à la projection d'un nœud sur un segment. Enfin si tous ces tests échouent, nous cherchons alors la collision soit comme un cas 2d, soit comme un cas 3d. Le cas 2d revient à projeter les deux nœuds sur le segment opposé pour déterminer la collision, tandis qu'en 3d nous cherchons une collision sur la longueur des segments. Nous ajoutons aussi un rayon de capture pour fusionner les segments colinéaires et proches. Cela évite notamment des phénomènes d'oscillation des segments qui entrent dans ce rayon de capture sans pour autant qu'il y ait une collision. Ces phénomènes font notamment chuter le pas de temps par les vitesses mises en jeu ; leur capture est donc essentielle.

La Figure 1.13 montre plus en détail le comportement de l'algorithme pour une hauteur d'arbre $H = 6$ avec au maximum 262 114 feuilles. Nous faisons évoluer le nombre de segments jusqu'à 1 000 000 et pour plus de la moitié les tests de collisions sont réduits à des cas en 2 dimensions dont la majorité sont écartés car les plans de glissement sont parallèles. Pour 100 000 segments, l'arbre est occupé à 10%, et nous effectuons des tests entre 1 270 000 paires de segments. Pour 1 000 000 de segments l'arbre est alors occupé à 60% et nous effectuons des tests entre 50 000 000 de segments, ce qui donne bien une augmentation du nombre de tests linéaire et non quadratique. La proportion de tests 3d

Algorithme 4 : Détection de collisions entre deux segments

```
Données : Segment segment1,Segment segment2
1  bool commonplane = FindCommonPlane(segment1, segment2);
2  si commonplane alors
3      bool areConnected = AreSegmentConnected(segment1, segment2);
4      si areConnected alors
5          TestConnected2DIntersection(segment1, segment2);
6      sinon
7          si segment1.getPlane() <> (segment2.getPlane() alors
8              return;
9          TestCapture(segment1, segment2);
10         Test2DIntersection(segment1, segment2);
11 sinon
12     bool areConnected = AreSegmentConnected(segment1, segment2);
13     si areConnected alors
14         TestConnected3DIntersection(segment1, segment2);
15     sinon
16         TestCapture(segment1, segment2);
17         Test3DIntersection(segment1, segment2);
```

(ici 50%) est dans la marge haute de ce qui peut arriver en simulation de DD. Dans cet exemple typique d'étude de défauts d'irradiation, nous avons de petites boucles carrées avec un vecteur de Burgers de type <100> [102] contenues dans deux plans de glissement ce qui fait augmenter le nombre de tests 3d. De ce fait, on peut aussi constater que dans 10% des tests il s'agit de collisions 3d entre segments interconnectés qui correspondent aux coins de ces boucles.

Pour ce type de simulations atteignant les densités de défauts recherchée, tout en contenant moins de 1 000 000 de segments, les optimisations algorithmiques présentées ici sont suffisantes et le coût le plus important reste largement lié au calcul du champ de force. Cependant, dans le cas de simulations plus larges, nous pouvons considérer les méthodes de détection à deux niveaux comme proposées dans des algorithmes de détection de collisions temps réel 3d [72]. L'idée reste de conserver le découpage hiérarchique tel que nous l'avons ainsi que l'algorithme de détection de collisions et d'ajouter en plus un niveau de détection grossier pour rejeter facilement une partie des tests effectués sur les boîtes voisines. Pour cela, la méthode classique pour les objets se déplaçant de manière cohérente dans l'espace au cours de la simulation est la technique *sweep and prune* [10]. On génère une unique liste de collisions potentielles à partir de la détection de la superposition des boîtes englobantes du vol (i.e. déplacement) de chaque segment. Pour cela, à partir de la liste triée des boîtes englobantes dans chaque direction, une paire de segments est ajoutée à la liste des contacts si leur boîte englobante se superpose dans les trois directions.

Cet algorithme n'a pas prouvé être plus efficace dans notre cas mais peut devenir

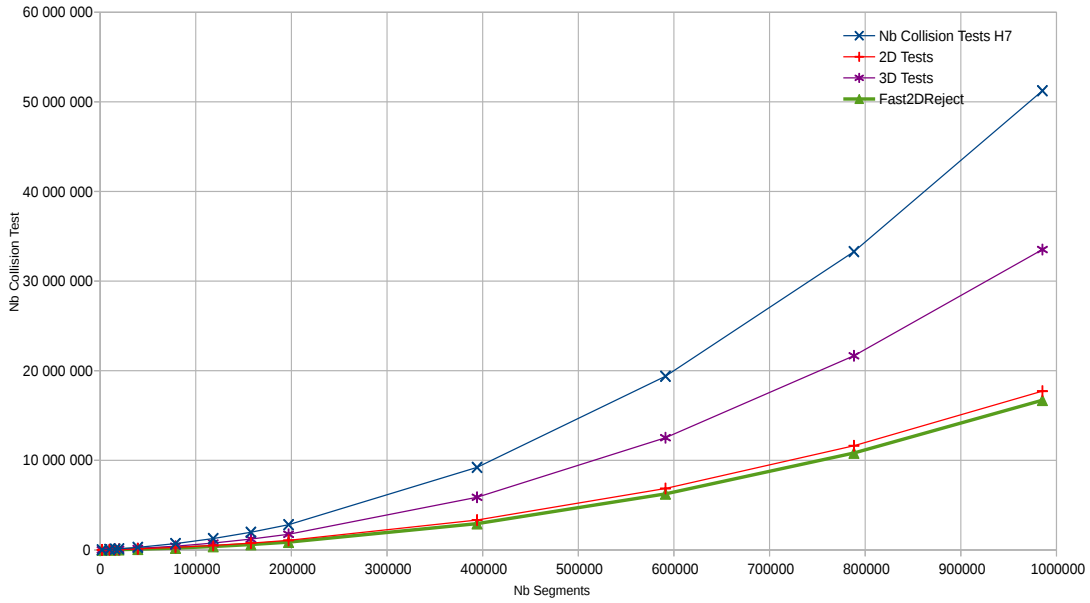


FIGURE 1.13 – Pour une hauteur d'arbre H=7, comportement de l'algorithme sur un cas contenant une répartition homogène de boucles d'irradiations.

utile si l'on considère la situation de segments qui ne sont plus contraints sur un plan de glissement. En perdant cette possibilité de simplifier la collision comme un cas 2d, cet algorithme devient alors intéressant à utiliser.

1.1.2.2 Traitement des collisions

Finalement, comme le montre la Figure 1.14, les collisions se produisant durant le pas de temps modifient la topologie du maillage. Lorsque nous détectons une collision, nous obtenons deux données : le temps auquel elle se produit et la position du point de collision. Nous utilisons le temps pour trier les collisions tandis que le point de collision permet de placer le nouveau nœud créé par la collision. Nous faisons en sorte que les collisions sur un segment soient traitées dans l'ordre chronologique pour éviter les incohérences.

Dans le cas d'une **collision 2d**, nous savons que le nœud nouvellement créé appartient au plan de glissement et que nous n'ajoutons au final pas de degré de liberté puisque il s'agit de la collision d'un nœud avec un segment. Cela se fait en deux étapes :

1. Tout d'abord, nous créons un nouveau nœud pour ajouter un degré de liberté au segment qui entre en collision et ainsi créer un nouveau segment ;
2. Ensuite nous fusionnons le nœud nouvellement créé et le nœud entrant en collision.

Au final nous n'insérons pas un nouveau degré de liberté. Si le nœud physique que nous venons de former est proche d'un nœud de discrétisation nous supprimerons le nœud de discrétisation lors du remaillage.

Dans le cas d'une **collision 3d**, nous devons insérer un nœud à l'intersection des deux plans de glissement pour ne pas créer des segments qui glissent en dehors de leur plan. Ce nœud physique sera donc contraint de glisser à l'intersection des deux plans. L'opération se décompose en 3 étapes :

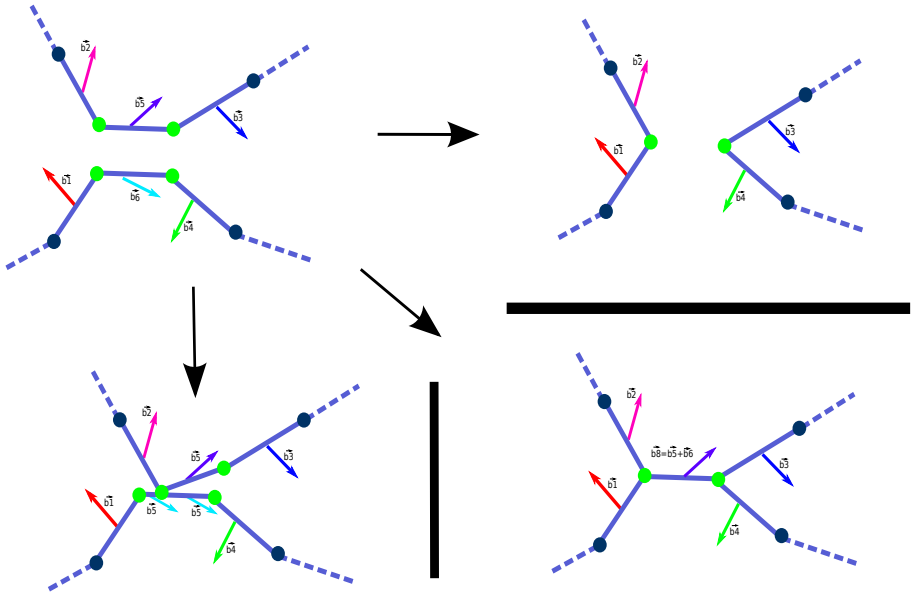


FIGURE 1.14 – Différentes formations possible après une ou plusieurs collisions.

1. Nous insérons un nouveau nœud pour ajouter un degré de liberté sur le premier segment et créer ainsi un nouveau segment ;
2. Nous insérons un nouveau nœud pour ajouter un degré de liberté sur le second segment ce qui insère un nouveau segment ;
3. Enfin, nous fusionnons ces deux nœuds au point de collision, c'est à dire à l'intersection des plans.

A partir de ces mécanismes de base, tout un ensemble de formations topologiques sont possibles comme le montre la Figure 1.14. Suite à une série de collisions, nous obtenons le mécanisme de jonction, comme dans la rencontre entre une dislocation vis et une boucle d'irradiation ou alors celui d'annihilation comme lors du phénomène de nucléation par source de Frank-Read avec la rencontre de segments possédant le même vecteur de Burgers. Ces mécanismes résultants d'un ensemble de collisions peuvent se produire sur un seul pas de temps ou sur plusieurs. Lorsque plusieurs collisions se produisent sur un pas de temps sur un même segment, puisque nous détectons l'ensemble des collisions avant de les traiter on peut veiller à bien conserver la cohérence du maillage. Pour cela la clef de notre algorithme reste de les traiter dans l'ordre chronologique.

1.2 Mise à jour de vitesses/positions

Comme introduit dans la Section 0.2.3, le calcul des vitesses nécessite de résoudre un système linéaire creux. Dans l'optique d'étudier de larges systèmes de dislocations contenant plusieurs centaines de milliers d'inconnues et donc de paralléliser le calcul, l'assemblage et la résolution de ce système global à chaque itération n'est pas envisageable. En suivant les travaux de l'équipe de Livermore [8], on considère que les vitesses sur les nœuds voisins sont approximativement les mêmes et en utilisant une loi de mobilité

linéaire on condense les coefficients de la matrice sur la diagonale. Cette approximation réduit alors le système linéaire global à une relation linéaire dépendant uniquement des nœuds directement connectés. Pour chaque nœud i la vitesse est donnée par :

$$\vec{V_i} = \frac{2}{B \sum_{j=0}^{N} L_{ij}} \vec{f_i},$$

avec f_i est la force au nœud i, B la loi de viscosité et L_{ij} la longueur du segment connectant les nœuds i et j. Cette vitesse doit ensuite être orthogonalisée par rapport aux différents plans de glissement des N segments connectés au nœud i. Cette approximation, contrairement au calcul global initial de NumoDis, peut rendre le système local non inversible sur un nœud. Cela, s'explique soit par un remaillage créant une trop forte discontinuité dans la courbure de la ligne notamment pour des topologies complexes sur les nœuds physiques soit par le rapprochement entre plusieurs lignes de dislocations qui implique l'augmentation brutale de la vitesse. Dans ce cas là le nœud est artificiellement bloqué (vitesse nulle).

Algorithme 5 : Calcul local des vitesses et de séparation des nœuds physiques.

```
1  si Node.GetNbArms() > 3 alors
       /* Déterminer toutes les options possibles pour séparer ce nœud
          physique                                                    */
2      SplitOptionsList ← Node.ComputeSplitOptions();
3      DissipativeEnergy ← −1.;
4      pour tous les SplitOption ∈ SplitOptionsList faire
           /* Pour chaque option nous calculons la puissance dissipée  */
5          SplitOption.ComputeNewLocalForce();
6          SplitOption.ComputeNewVelocity();
7          TmpEnergy ← SplitOption.ComputeEnergy();
8          si TmpEnergy > DissipativeEnergy alors
9              FinalConfiguration ← SplitOption;
10         fin
11     fin
12     Mesh.ChangeTopology(Node,FinalConfiguration);
13 sinon
       /* Calcul local des vitesses                                   */
14     ...
15 fin
```

L'Algorithme 5 présente le calcul des vitesses pour une loi de mobilité linéaire. La complexité de l'algorithme est linéaire en fonction du nombre de nœuds. De plus, pour augmenter l'intensité arithmétique et sans ajouter de dépendance entre les calculs, nous évaluons à la volée sur un nœud physique (ligne 3 de l'Algorithme 5), la configuration qui maximise la puissance dissipée. Pour obtenir la puissance sur chaque nouvelle configuration, on détermine la contribution énergétique des segments locaux par un calcul de

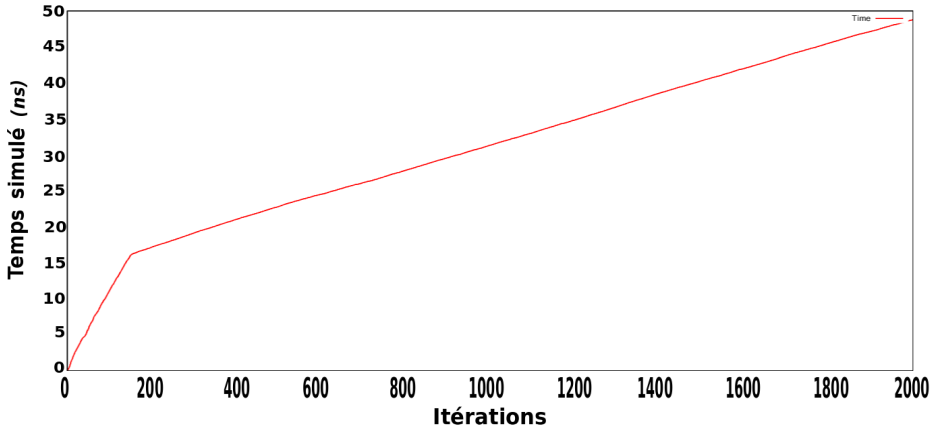
tension de ligne et d'énergie de cœur sur cette nouvelle topologie. Enfin, on ajoute par interpolation la contribution du champ lointain à partir des segments de la configuration initiale. De plus, si elle existe on ajoute la contribution de la jonction en projetant les nœuds à une distance minimum L_{min} l'un de l'autre.

Cet algorithme donne une vitesse sur chaque nœud et nous constatons de très fortes variations dans ce champ de vitesses au cours des itérations. Par conséquent, pour déterminer le pas de temps Δt optimal pour le schéma explicite (Forward Euler), il faut résoudre un problème d'optimisation qui dépend de multiples paramètres. Pour déterminer Δt, on considère les données de vitesse maximale, vitesse moyenne des nœuds sur le réseau de dislocations ainsi que la longueur maximale et minimale des segments et enfin le déplacement maximal et moyen pour les segments. Le calibrage de l'algorithme pour déterminer Δt par rapport à la vitesse de déplacement est important notamment dans les simulations d'écrouissage avec des dislocations initialement très mobiles. Ce déplacement provoque une forte densification du nombre de défauts et une augmentation des interactions (jonctions/séparations) conduisant à des variations de plusieurs ordres de grandeur de la vitesse des nœuds au cours de la simulation. Les tailles de discrétisation minimales L_{min} et maximales L_{max} sont d'autant plus importantes lorsque l'on simule de petits objets comme des boucles d'irradiation. Ces objets qui sont bien plus petits que des lignes de dislocations, doivent être discrétisés plus finement et font donc fortement diminuer le pas de temps.

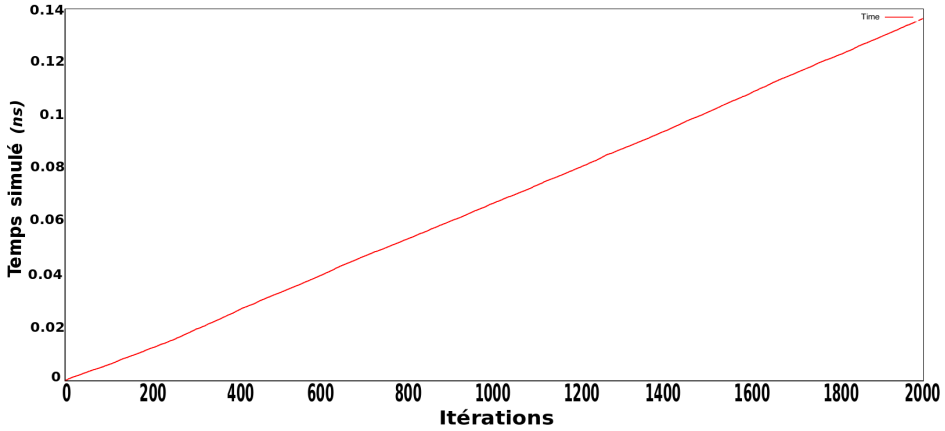
La Figure 1.15 montre pour différentes simulations, le temps simulé au cours de 2000 itérations. Celui-ci est très largement influencé par la taille des défauts et le mode de pilotage. La simulation de la Figure 1.15a, comporte des sources de Frank-Read très mobiles soumises à une forte contrainte constante $\sigma = 500 MPa$. Nous pouvons voir deux étapes dans le calcul. Sur les 200 premières itérations les sources sont droites, peu mobiles, et sans interactions entre elles ce qui signifie que le champ de vitesse est homogène et que le pas de temps peut être augmenté. Ensuite, à partir du moment où les sources commencent à s'activer, la simulation devient extrêmement dynamique avec de nombreuses interactions provoquant de fortes variations des vitesses entre les nœuds quasi immobiles et ceux qui se déplacent le plus vite. Ces deux phénomènes combinés obligent l'algorithme à adapter le pas de temps en diminuant celui-ci pour contrôler les aspects les plus dynamiques de la simulation.

Dans le cas d'une simulation avec des boucles d'irradiation (Figure 1.15b), la taille des objets (quelques dizaines d'atomes) oblige à diminuer le pas de temps à des valeurs de l'ordre des simulations de dynamique moléculaire. Nous ne simulons que $0.15\,ns$ sur 2 000 itérations contre $50\,ns$ avec le cas précédant. Cela s'explique principalement par le fait que l'on utilise un schéma explicite (*Forward Euler*) qui est conditionnellement stable. Ce schéma par expérience et d'après [89] est contraint pour la stabilité à ne pas déplacer les segments de plus de $\frac{L_{min}}{2.}$. Ce qui implique que pour chaque pas de temps Δt_{max} est contrôlé par V_{max} tel que $\Delta t_{max} \leq \frac{L_{min}}{2||V_{max}||}$.

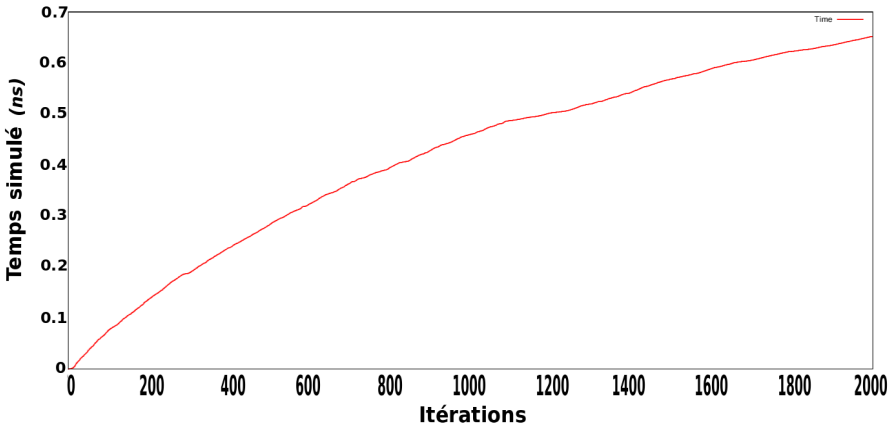
Le dernier cas, Figure 1.15c met en évidence l'adaptation du pas de temps à la taille des objets et au déplacement de ceux-ci en réponse à la contrainte qui est croissante de 0 à $800\,MPa$. La contrainte croissante oblige à réduire en conséquence le pas de temps pour contenir le déplacement des segments de plus en plus important en réponse à la contrainte appliquée. Cependant, ce critère est assoupli en mesurant le déplacement moyen



(a) Simulation de sources de Frank-Read avec une contrainte appliquée constante (Discrétisation 40-90).

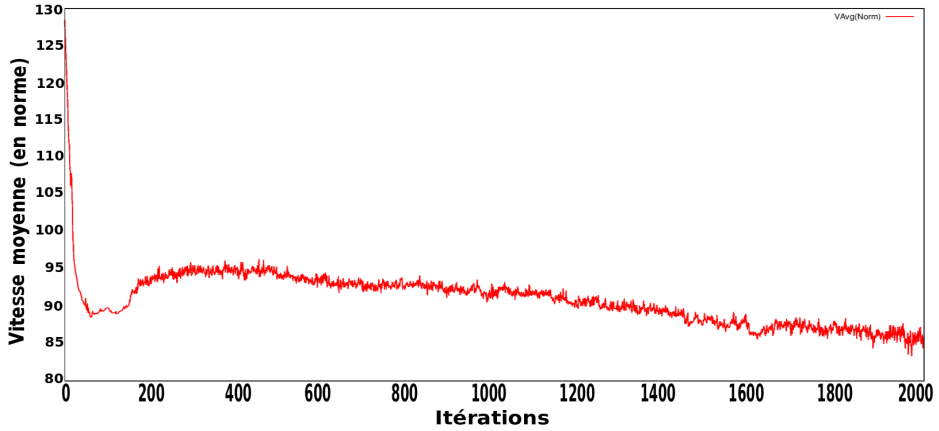


(b) Simulation de défauts d'irradiation avec un pilotage par contrainte appliquée constante (Discrétisation 20-50).

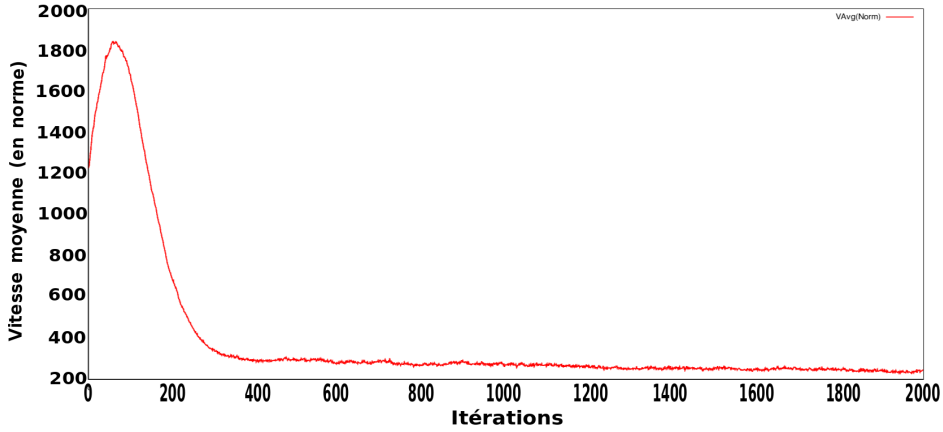


(c) Simulation de défauts d'irradiation avec un pilotage par déformation croissante (Discrétisation 20-50).

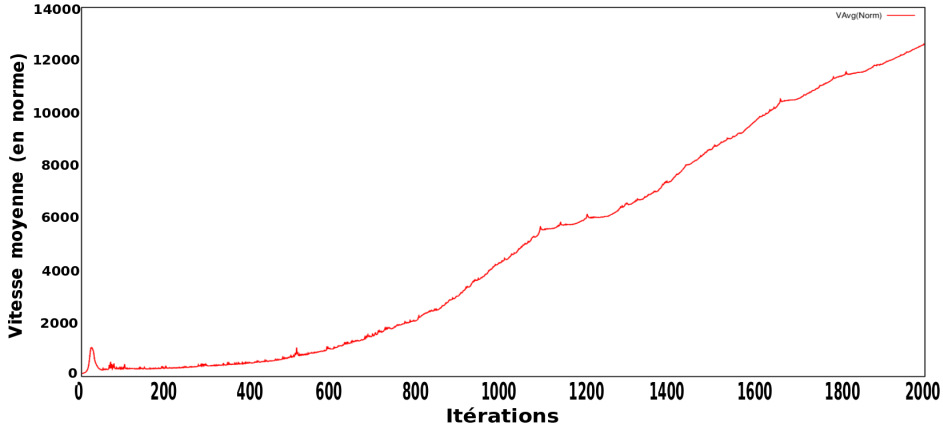
FIGURE 1.15 – Temps simulé sur 2000 itérations selon les objets, leur discrétisation et le mode de pilotage.



(a) Simulation de sources de Frank-Read avec une contrainte appliquée constante (Discrétisation 40-90).



(b) Simulation de défauts d'irradiation avec un pilotage par contrainte constante (Discrétisation 20-50).



(c) Simulation de défauts d'irradiation avec un pilotage par déformation croissante (Discrétisation 20-50).

FIGURE 1.16 – Évolution de la vitesse moyenne des nœuds du système sur 2000 itérations selon les objets, leur discrétisation et le mode de pilotage.

des segments dans le cristal. C'est ce qui permet par exemple de simuler plus de temps dans le cas d'une simulation contenant des défauts fortement statiques (Figure 1.15b). En activant ce critère, nous constatons que les boucles sont statiques, en calculant le déplacement moyen. A partir de là, nous pouvons concentrer le calcul sur les nœuds réellement mobiles qui eux seuls vont contrôler la taille du pas de temps.

Avec les graphiques de la Figure 1.16, nous montrons pour les trois mêmes cas que précédemment l'évolution de la vitesse moyenne des nœuds. Pour les deux premiers cas, en dehors des premières itérations qui sont instables car la contrainte est appliquée sur des dislocations générées aléatoirement qui ne sont pas en position d'équilibre, nous avons une vitesse moyenne stable. La contrainte appliquée constante nous permet d'avoir une régime stable pour bien contrôler le pas de temps. Dans le cas d'une contrainte croissante (Figure 1.16c) la vitesse de déplacement est alors croissante ce qui oblige à ajuster en conséquence le pas de temps. Cependant comme nous venons de le dire grâce au critère de mesure du déplacement moyen, le vol moyen des nœuds reste stable et faible ($\backsimeq 0.1 \times L_{min}$), ce qui permet de ne pas faire trop diminuer Δt par rapport au V_{max}.

1.3 Maillage adaptatif

Après le calcul des vitesses et le déplacement des points, nous devons maintenir une bonne discrétisation et une bonne représentation du réseau de dislocations notamment de la courbure des lignes. Pour cela, le remaillage est la dernière étape d'une itération et nous considérons les deux points suivants :

1. Chaque nœud inséré a un coût pour les étapes de calcul de force et de collisions par conséquent seul les segments strictement nécessaires doivent être introduits.
2. L'adaptation du maillage ne doit pas contraindre le déplacement d'une ligne de dislocation. Cela signifie que selon la déformation appliquée et les obstacles rencontrés on insère des degrés de liberté (c'est à dire des nœuds) pour suivre la déformation des lignes et ne pas fausser la physique du phénomène simulé.

L'ensemble de cet algorithme de remaillage est basé sur les critères de contrôle de l'erreur lors du calcul de la force notamment pour les segments à cheval sur deux boîtes ainsi que ceux de stabilité du schéma d'intégration en temps. Par conséquent pour un segment S, nous fixons le critère de discrétisation suivant :

$$L_{min} < ||length(S)|| < L_{max},$$

et pour éviter des choix contradictoires au cours de deux itérations lors de l'insertion ou de la suppression de nœuds, nous ajoutons le second critère suivant

$$2L_{min} < L_{max}.$$

Ainsi, l'insertion d'un nœud sur un segment de taille supérieure à L_{max} ne conduit pas à la création de deux segments de taille inférieure à L_{min}. Inversement, il faut vérifier que lors de la fusion entre deux nœuds cela ne conduise pas à la formation d'un segment de longueur supérieure à L_{max}.

L'algorithme de remaillage fonctionne en trois passes comme présenté dans Algorithme 6. Les trois phases sont

Algorithme 6 : Remaillage simplifié en 3 étapes.

```
   /* Phase 1. : suppression des segments trop petits.                  */
 1 pour tous les Segment ∈ Mesh faire
 2 |   si Length(Segment) < L_min alors
   |   |   /* On supprime le segment et on fusionne les nœuds a l'aide des
   |   |      Algorithmes 7, 8 et 9.                                    */
 3 |   |   ...
 4 |   fin
 5 fin
   /* Phase 2. : découpage des segments trop grands.                    */
 6 pour tous les Segment ∈ Mesh faire
 7 |   si Length(Segment) > L_max alors
   |   |   /* On découpe le segment en deux en insérant un segment et un
   |   |      nœud.                                                     */
 8 |   |   ...
 9 |   fin
10 fin
   /* Phase 3. : Amélioration de la description de la géométrie.        */
11 pour tous les Segment ∈ Mesh faire
12 |   si Areunbalanced(Segment, Segment.next) alors
   |   |   /* On équilibre selon la courbure et les tailles des deux
   |   |      segments en déplaçant les nœuds.                          */
13 |   |   ...
14 |   fin
15 fin
```

Phase 1. Deux nœuds peuvent être fusionnés pour supprimer un segment ;

Phase 2. Un segment peut être découpé pour insérer un nouveau nœud ;

Phase 3. Un nœud peut être déplacé pour ajuster la distribution selon la géométrie.

Dans l'algorithme d'origine de Numodis, le remaillage revient à parcourir les lignes les unes après les autres et pour chaque ligne, l'ensemble des nœuds ordonnés allant d'un nœud physique à l'autre. Or, cette vision par ligne est trop contraignante pour l'approche parallèle notamment pour maintenir la cohérence dans l'ordre des nœuds et entre les lignes suite à la modification du maillage. Ainsi, le nouvel algorithme développé prend comme unité de base le segment avec seulement l'information à ses extrémités. A savoir, le nombre de connexions et le degré de liberté pour le déplacement des nœuds. Cependant certaines contraintes nous empêchent de simplement supprimer un nœud lorsque deux segments sont trop petits. Il faut alors considérer les degrés de liberté de déplacement des nœuds pour décider s'ils peuvent être fusionnés. A travers les Algorithmes 7, 8 et 9 nous détaillons l'ensemble des cas possibles pour fusionner les deux extrémités d'un segment S tel que $||length(S)|| < L_{min}$. Dans le cas où les deux nœuds peuvent être fusionnés, il faut savoir à quel endroit le placer pour ne pas faire sortir les segments connectés de leurs

plans de glissement.

Algorithme 7 : Algorithme de fusion des nœuds lorsque le nœud 1 est ancré (ddl=0).

Données : Segment seg, nœud Node1, nœud Node2

1 $ddl_2 \leftarrow GetDOFNode2()$;
2 **suivant** ddl_2 **faire**
3 **cas où** $ddl_2 = 0$
`// nœuds immobiles, fusionner si la distance est critique.`
4 $seg.GetLength() < CriticalLength$;
5 **cas où** $ddl_2 = 1$
`// Le nœud ne se déplace que dans une unique direction. Vérifier que le segment est bien inclue dans cette direction.`
6 $seg.GetDirection() = Node2.GetDirection()$;
7 **cas où** $ddl_2 = 2$
`// Le segment appartient au plan de glissement donc fusionner ces deux nœuds à la position du nœud 1.`

L'Algorithme 7 détaille le cas où le nœud 1 du segment S est immobile ($ddl = 0$). Si les extrémités sont deux nœuds physiques et si la distance est inférieure à L_{min}, la fusion de deux nœuds physiques est alors problématique. De plus, si les nœuds sont coincés par plusieurs plans de glissement le nœud est fixé. Il faut introduire une distance critique d'annihilation ($CriticalLength$) et aussi autoriser des segments de taille inférieure à L_{min} à exister au delà d'un pas de temps.

L'Algorithme 8 présente le cas où le nœud 1 du segment S ne peut glisser que dans une seule direction ($ddl = 1$). Nous trouvons par exemple le cas à la ligne 5 de l'algorithme où l'autre nœud est aussi mobile dans une unique direction ($ddl = 1$). Nous devons alors chercher (s'il existe) le point de convergence de ces deux directions pour pouvoir autoriser la fusion. Enfin l'Algorithme 9 détaille le cas classique d'un segment ayant un nœud de discrétisation contraint seulement sur son plan de glissement ($ddl = 2$). Cette possibilité correspond au cas trivial du remaillage d'un segment connectant deux nœuds de discrétisation où la fusion est toujours possible dans le plan de glissement. Pour certains nœuds introduits lors de la collision avec un précipité, on bloque la suppression de ce nœud (via un "*flag*" booléen). Comme montré sur la Figure 1.17, il est indispensable pour décrire la courbure de la dislocation en contact avec un précipité et éviter la pénétration dans le précipité. Pour le raffinement le principal critère est la longueur du segment qui doit être inférieur à L_{max}. Toutefois, lorsqu'une ligne entre en collision avec le grain, la description de la courbure est inutile donc nous faisons en sorte que la longueur du maillage soit la plus proche de L_{max} notamment en repérant le nœud bloqué sur le grain dont la vitesse est nulle comme sur la Figure 1.18a. Pour permettre cette adaptation, le raffinement est combiné avec un algorithme d'équilibrage de la distribution des nœuds. On prend en compte les segments directement voisins pour équilibrer la distribution selon la

Algorithme 8 : Algorithme de fusion des nœuds d'un segment dont le nœud 1 glisse dans une direction (ddl=1).

Données : Segment seg, nœud Node1, nœud Node2

1 $ddl_2 \leftarrow GetDOFNode2()$;
2 **suivant** ddl_2 **faire**
3 **cas où** $ddl_2 = 0$
`// Le nœud 1 ne se déplace que dans une direction donc vérifier que le segment est inclue dans cette direction.`
4 $seg.GetDirection() = Node1.GetDirection()$;
5 **cas où** $ddl_2 = 1$
`// Les 2 nœuds ne se déplacent que dans une direction`
`// Vérifier que ces directions sont identiques.`
6 **si** $Node1.GetDirection() == Node2.GetDirection()$ **alors**
`// Vérifier que le segment est bien inclue dans cette unique direction.`
7 $seg.GetDirection() = Node1.GetDirection()$;
8 **sinon**
`// Fusion des deux nœuds à l'intersection des deux directions.`
9 **cas où** $ddl_2 = 2$
`// Le segment appartient au plan de glissement donc fusionner ces deux nœuds à la position du nœud 1.`

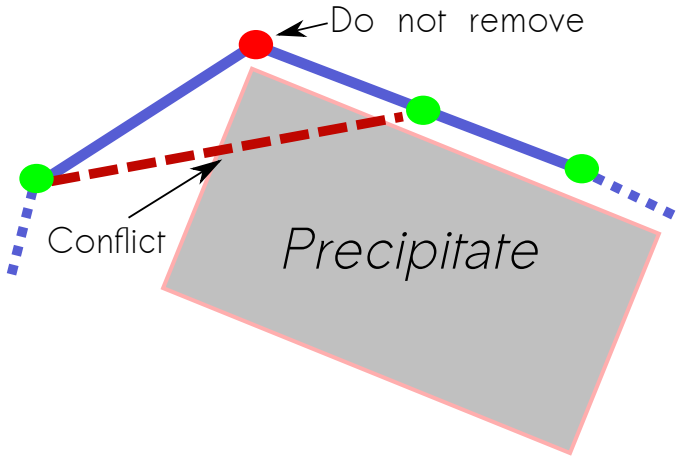


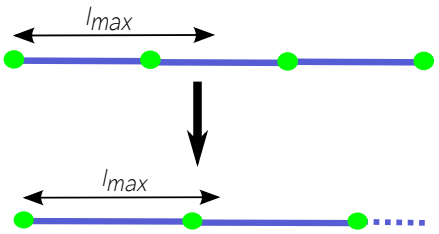
FIGURE 1.17 – Marquage des nœuds ne pouvant pas être déplacés ou supprimés au cours du remaillage.

courbure en calculant l'angle entre les segments. Si les segments sont fortement colinéaires, nous équilibrons le plus possible la distribution pour insérer le moins de nœuds possible

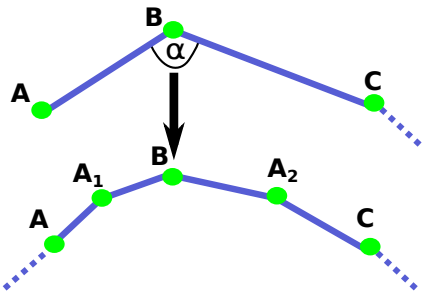
Algorithme 9 : Algorithme de fusion des nœuds d'un segment dont le nœud 1 glisse dans le plan (ddl=2).

Données : Segment seg, nœud Node1, nœud Node2

1 $ddl_2 \leftarrow GetDOFNode2()$;
2 **suivant** ddl_2 **faire**
3 **cas où** $ddl_2 = 0$
// Le nœud 2 ne bouge pas donc fusionner à la position du nœud 2.
4 **cas où** $ddl_2 = 1$
// Le nœud 2 ne bouge que dans une direction donc fusionner à la position du nœud 2.
5 **cas où** $ddl_2 = 2$
// Vérifier que les nœuds appartiennent au même plan.
6 **si** $Node1.GetPlane() == Node2.GetPlane()$ **alors**
// Fusionner les deux nœuds ne pose pas de problèmes.
7 **sinon**
// Trouver l'intersection des deux plans pour fusionner les deux nœuds.



(a) Équilibrage avec espacement de la distribution sur une zone de faible courbure.



(b) Équilibrage et raffinement sur une zone de forte courbure selon l'angle α.

FIGURE 1.18 – Techniques d'équilibrage et de raffinement.

c'est le cas de la Figure 1.18a. Dans le cas où les segments montrent une forte courbure (angle α sur la Figure 1.18a), nous raffinons si nécessaire et équilibrons aussi la charge en rapprochant les nœuds du point de plus forte courbure (nœud B Figure 1.18a), qui reçoit la contrainte la plus importante comme illustré dans la Figure 1.18b. Les nœuds A_1 et A_2 sont placés sur le cercle circonscrit aux nœuds A B et C puis rapprochés ou éloignés de B selon l'angle α. L'adaptation de la discrétisation à la courbure permet notamment d'éviter d'isoler les nœuds recevant une très forte contrainte, qui deviennent instables et obligent à diminuer le pas de temps pour contrôler leur comportement.

Bilan

La modification en profondeur du code original NumoDis, a été la première tâche de ce travail. Afin d'orienter les simulations vers les grandes échelles deux fortes modifications algorithmiques ont été apportées au code. D'une part, une nouvelle vision sur la dépendance des données est apportée au code Numodis par un maillage de type éléments finis classique avec des sommets (nœuds) et des arrêtes (segments) ce qui entraîne la disparition de la notion de ligne de dislocation. D'autre part, avec l'introduction de la bibliothèque ScalFMM nous pouvons calculer le champ de force élastique avec une complexité linéaire en fonction du nombre de segments (FMM) et bénéficier du découpage hiérarchique de l'espace pour les autres phases de calcul. ScalFMM apporte aussi la généricité et la performance pour brancher directement notre noyau de détection de collision. Enfin l'introduction d'algorithmes hiérarchiques offre la base indispensable pour décomposer le domaine afin de paralléliser le code sur des clusters de calcul.

Par la suite nous avons étudié les différents problèmes limitant la performance et apporté des solutions algorithmiques pour que l'ensemble des étapes de la simulation se déroule individuellement mais aussi globalement au mieux. Nous avons notamment mis en évidence, une hauteur de l'octree optimale pour H=6, afin de contrôler le coût lié à la multiplication des segments durant la simulation. Cette hauteur permet à l'algorithme d'équilibrer le calcul du champ de force entre le champ proche et le champ lointain. Elle permet aussi d'adapter la fréquence de calcul du champ lointain en laissant la majeure partie des force reçu par un segment dans le champ proche.

Enfin, nous avons vu que le choix de la hauteur de l'arbre doit être fait en relation avec le paramétrage de la simulation. Des limites apparaissent selon la taille de la boîte de simulation, la taille des objets simulés, ainsi que le schéma d'intégration en temps (schéma explicite) qui oblige à porter une attention particulière aux critères de discrétisation.

Après cette étude au niveau algorithmique, nous avons connaissance des différents mouvements des données au niveau mémoire ainsi que les accès et les parcours associés aux algorithmes. Nous allons pouvoir dans le chapitre suivant définir les structures de données qui permettront au code d'être à la fois, performant dans les phases de calcul intensif, et adaptatif pour la gestion des opérations discrètes.

Chapitre 2

Structure de données adhoc pour la simulation en DD

Sommaire

2.1 Motivation pour des structures appropriées en DD

La dynamique des dislocations présente différentes étapes algorithmiques qui peuvent être regroupées en deux catégories. D'une part, les algorithmes pour calculer les forces et détecter des collisions nécessitent des accès rapides aux données ce qui implique une organisation des données et un pattern d'accès adéquat pour minimiser les défauts de cache. D'autre part, les étapes de séparation des nœuds physiques et de remaillage se traduisent par de nombreuses insertions et suppressions de nœuds. Ce dynamisme conduit à une adaptation permanente du maillage qui perturbe la localité des données. En conséquence, en modifiant l'organisation des données, les accès préalablement cohérents deviennent aléatoire en mémoire, dégradant ainsi au pas de temps suivant, la performance pour les phases de calcul intensif.

Nous allons dans cette partie, présenter nos structures de données et leur fonctionnement pour permettre de concilier au mieux ces deux phases antinomiques et ainsi d'optimiser l'efficacité des calculs tout au long de la simulation.

Dans un premier temps, nous présenterons les structures de données usuelles avec leurs avantages et inconvénients, pour comprendre les éléments qui ont orienté la structure de données d'OptiDis. Puis nous décrirons cette structure de données et nous montrerons comment elle s'adapte aux différentes étapes algorithmiques. Nous introduirons les éléments nécessaires à l'implémentation de cette nouvelle structure de données adaptée à un maillage dynamique : les algorithmes pour absorber le dynamisme lié aux modifications topologiques tout en maintenant la cohérence mémoire et enfin l'intégration avec la décomposition hiérarchique du domaine par l'octree.

2.2 Structures de données génériques pour problèmes dynamiques

Dans la littérature, on constate que les efforts pour développer des structures de données adaptées aux problèmes dynamiques se retrouvent dans les domaines du partitionnement de graphes ou encore les problématiques de maillages adaptatifs. Ces domaines combinent les besoins de concevoir des structures de données dynamiques ayant aussi de bonnes performances en terme de coût des opérations d'accès et de mise à jour des opérandes tout en limitant les ressources mémoire utilisées.

Au cours de notre simulation, les données sont accédées de nombreuses façons :

— de manière transversale simple ;
— par connectivité entre les éléments ;
— par localité spatiale avec uniquement les données proches dans la boîte de simulation.

Pour répondre à tous ces besoins algorithmiques, il est nécessaire d'utiliser plusieurs structures de données qui doivent inter-opérer de manière efficace.

Pour les simulations basées sur des éléments géométriques, on utilise classiquement deux structures de données pour décrire le maillage. La première structure contient les

sommets et la seconde les éléments. En DD, on aura une structure pour les nœuds et une seconde pour les segments avec une inter-dépendance entre ces deux structures, chacune référençant des données de l'autre. De part la dynamique propre à la DD, il est trop pénalisant de maintenir une numérotation globale pour référencer les éléments. Par ailleurs, il est fréquent de combiner ce stockage mémoire avec une structure de données de type hiérarchique pour ajouter un découpage spatial de données.

En dehors des parcours des données, nous avons quatre opérations à exécuter efficacement qui déterminent le choix d'une structure de données. Ces quatre opérations sont :

— Lire : pour accéder à une donnée de manière directe ;
— Écrire : pour pouvoir modifier des données déjà présentes en mémoire ;
— Étendre : pour agrandir la structure afin d'insérer de nouveaux éléments ;
— Réduire : pour réduire la structure afin de diminuer l'empreinte mémoire.

Il s'agit des primitives qui sont soit statiques, dans le sens où elles ne modifient que des espaces en mémoire déjà réservés, soit dynamiques puisqu'elles modifient la mémoire pour permettre l'insertion ou la suppression de données. A partir de ces opérations, nous analysons les structures classiques pour juger de leur performance face à nos problèmes dynamiques.

Dans la section suivante, nous introduisons quelques résultats concernant les propriétés et implantations des structures de données classiques comme les tableaux dynamiques et les listes chaînées.

2.2.1 Tableaux dynamiques

Un tableaux dynamique [1] peut stocker une collection de n éléments de même taille m (en octets), avec des indices allant de 0 à $n-1$. La taille optimale d'un tableau de n éléments est de $n \times m$ octets. Le principal point fort de cette structure est sa performance pour les opérations statiques (Lire/Ecrire) car l'ensemble de données sont stockées de manière contiguë en mémoire. Avec l'indexation directe classique d'un tableau, les opérations d'accès réguliers et de mise à jour sont un temps constant $\mathcal{O}(1)$.

Cependant dans les simulations avec de nombreuses insertions et suppressions d'éléments, il est nécessaire de munir ces tableaux d'opérations dynamiques. Ainsi, on ajoute l'allocation mémoire afin de pouvoir adapter l'espace mémoire aux besoins de la simulation. Nous obtenons donc une structure de données initialement statique avec les opérations supplémentaires pour l'augmenter (*Étendre*) et la diminuer (*Réduire*) [48]. En introduisant ces fonctionnalités nous pouvons faire varier le nombre d'éléments stockés de manière illimitée. La réservation complète d'un nouveau bloc mémoire plus grand est extrêmement coûteuse et entraîne la recopie de larges zones mémoires. Dans un processus très dynamique, cela s'avère trop pénalisant, d'autant que si l'on souhaite réduire le nombre de réallocations on doit réserver une bien plus grande zone mémoire que nécessaire. Ce qui en fin de compte pénalise la simulation avec une mauvaise gestion des ressources.

Enfin, travailler avec une unique zone mémoire pour plusieurs milliers de données nécessite une gestion fine des opérations de suppression et d'insertion. En effet, bien souvent l'insertion va se faire en fin de tableau pour garantir un temps constant et fera perdre la localité spatiale des données. Pour la suppression de manière aléatoire dans la

structure, il faut soit gérer une liste de trous soit effectuer des déplacements mémoire pour maintenir les données contiguës sur l'ensemble du tableau.

2.2.2 Structures avec indirections

La liste chaînée est la structure classique avec indirection. Le principe est de pouvoir allouer n'importe où dans la mémoire, un espace contenant la donnée à stocker ainsi que l'adresse de l'espace mémoire contenant la donnée suivante. Une telle structure dans sa forme classique, permet une insertion et une suppression des données n'importe où dans la liste en un temps constant. Cependant, lors de son parcours les performances sont dégradées du fait de la mauvaise localité des données entraînant des sauts en mémoire. Enfin, réorganiser les données pour gérer la localité n'a pas de sens pour une structure chaînée basique qui fonctionne seulement avec le référencement par la donnée suivante et précédente et non pas sur le placement mémoire.

De nombreux travaux tentent de lever ces limitations en stockant de manière contiguë une partie des éléments [49]. Ce type de structure réduit fortement les désavantages des listes chaînées classiques et présente de nombreux atouts pour réaliser des simulations en DD. Nous adaptons cette structure hybride à certaines spécificités de nos algorithmes. Nous avons développé une structure dont les propriétés bénéficient des avantages des deux structures classiques, les listes et les tableaux, tout en minimisant les désavantages dans nos algorithmes comme nous le présenterons par la suite.

2.3 Structure de données pour la DD

Nous nous concentrons ici sur les développements des structures de données pour le maillage éléments finis 1D avec les segments et les nœuds interconnectés.

2.3.1 L'architecture de la structure

De manière simple, notre structure de données s'apparente à une liste doublement chaînée de tableaux comme introduit dans la Figure 2.1. Chaque élément de la liste est appelé un bloc et il est constitué de 4 données membres. Tout d'abord, deux pointeurs vers les blocs précédents et suivants, un tableau de taille fixe B de type Elt et d'un entier (`idxFree`) donnant le premier élément libre dans le tableau (cf Figure 2.2). Dans le tableau les données sont contiguës de l'indice 0 à `idxFree` -1. On s'assure ainsi d'avoir des chargements de lignes de cache pleins et cohérents. De plus, éviter de tester chaque élément pour savoir s'il est valide ou non, permet d'éviter des branchements coûteux dans les algorithmes. Un paramètre important de la structure est la taille du tableau notée B qui est fixée à la compilation. La valeur de B doit être suffisamment grande pour bénéficier des effets de cache importants lors des différents phases algorithmiques, mais aussi suffisamment petite pour permettre la création d'un grand nombre de blocs pour autoriser suffisamment de concurrence entre les blocs.

Cette structure doit s'adapter aux caractéristiques des phénomènes simulés (dislocations plus ou moins mobiles ou dynamiques) sur des architectures modernes, aussi nous

```
struct List{
  Block *fisrt, *last;
  int    nbBlocks;
}
```

```
struct Bloc{
  Block *previousBlock, *nextBlock;
  int    idxFree ;
  Elt    tab[B] ;
}
```

FIGURE 2.1 – Les éléments de notre structure de données.

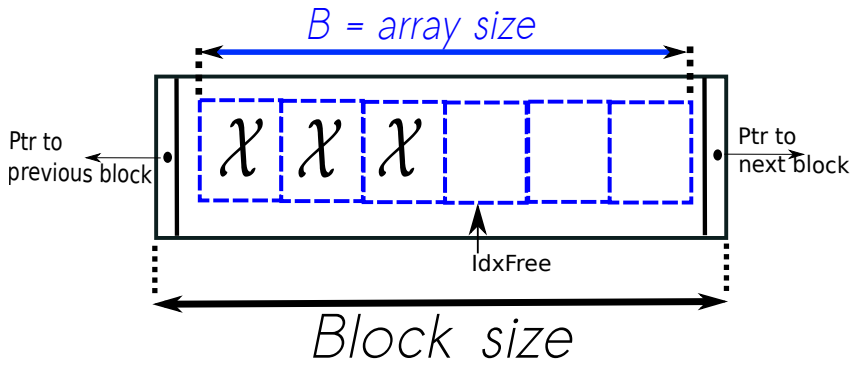


FIGURE 2.2 – Structure de donnée chaînée.

la positionnons dans la catégorie des structures dites *cache oblivious* [100]. En divisant les données blocs, en les agrégeant par la suite et avec la notion de *self tunning* [39] notre structure de données s'adapte automatiquement avec le scénario de la simulation. La structure doit principalement pouvoir absorber le dynamisme à certains endroits du maillage sans remettre en cause l'organisation globale des données. Ce type de structure avec son paramétrage et sa capacité d'adaptation est le parfait intermédiaire entre la liste chaînée et le tableau dynamique sans le coût du chaînage entre chaque donnée.

Avant de détailler comment les quatre primitives s'appliquent sur cette structure, nous présentons d'abord la structure de données maillage basée sur cette structure hybride.

2.3.2 Accès à un élément

Dans une structure chaînée par blocs, un moyen efficace et standard pour accéder aux données est de représenter chaque élément par un couple que nous appelons `ccIndex` comme introduit dans la Figure 2.3. L'indice `ccIndex` d'un élément de la structure est composé de deux données :

— `ptrBlock` qui correspond au pointeur sur le bloc de la liste contenant la donnée ;

```
struct ccIndex{
  Block* ptrBlock;
  int    idxPosition;
}
```

FIGURE 2.3 – Indice d'un élément pour un accès direct.

— `idxPosition` qui donne la position dans le tableau `tab` du bloc.

Ainsi en utilisant cet indexage nous avons un accès rapide aux données que nous soyons avec un seul bloc dans une configuration simple tableau ou avec n blocs chaînés. Pour ce type de structure, avoir un accès au bloc plutôt que directement sur la donnée est essentiel pour gérer finement les réorganisations entre les blocs de la structure sans pénaliser l'accès en simple lecture et écriture.

2.3.3 Structure de données maillage

La Figure 2.4 introduit les principaux attributs pour définir le type *Nœud* et le type *Segment*. Dans la simulation, le type `Nœud` avec tous ces attributs a une taille de 312 *octets*

```
struct NODE {
  double[3]   position;
  double[3]   velocity;
  double[3]   forces;
  int    numberOfArms ;
  ccIndex[numberOfArms] segments ;
}
```

(a) Attributs du type `Nœud` avec les références (`ccIndex`) vers les segments connectés au nœud.

```
struct SEGMENT {
  ccIndex  NODE1 ;
  ccIndex  NODE2 ;
  GlideSystem* gsystem ;
}
```

(b) Attributs du type `Segment` avec les références (`ccIndex`) vers les extrémités du segment.

FIGURE 2.4 – Attributs pour les nœuds et les segments. Les deux structures sont fortement interdépendantes.

et le type `Segment` 168 *octets*. Toutefois dans ces chiffres il manque notamment des attributs supplémentaires pour la gestion du maillage en parallèle. La taille réelle des données comparée à la taille usuelle du cache L1 de $32Ko$ pour la machine cible, révèle l'importance pour nous de manipuler des structures adaptées pour ne pas saturer le cache inutilement. Par exemple, lors du calcul de l'interaction entre deux segments pour évaluer la force, le chargement des données utiles (position des nœuds et vecteurs de Burgers des segments) est de 144 *octets*. Il suffit de 227 paires de segments pour saturer le cache. Pour le calcul de collisions, nous chargeons la position des nœuds et la vitesse de déplacement de chacun pour un total de 240 *octets* soit quasiment moitié moins de paires.

Les deux types de données du maillage sont interdépendantes. Un segment est toujours référencé par deux nœuds tandis qu'un nœud peut appartenir entre 1 et N segments. Cette différence a son importance pour les accès lors du calcul de la vitesse qui parcourt l'ensemble des connexions d'un nœud. Cependant, en général, la grande majorité des nœuds sont des nœuds de discrétisation et sont donc connectés uniquement à deux segments.

Pour accéder à un élément via le `ccIndex`, il faut pouvoir manipuler la structure de données de façon transparente quelle que soit l'implémentation sous-jacente. Ainsi, on doit permettre à l'utilisateur un accès uniforme pour lire ou écrire que l'on soit avec une organisation en structure de tableaux (*SOA*) ou en tableau de structure (*AOS*). Pour cela,

un mécanisme d'interfaçage est proposé (Figure 2.5) pour masquer l'implémentation qui peut être choisie à la compilation. Pour chaque type que l'utilisateur créé, par exemple

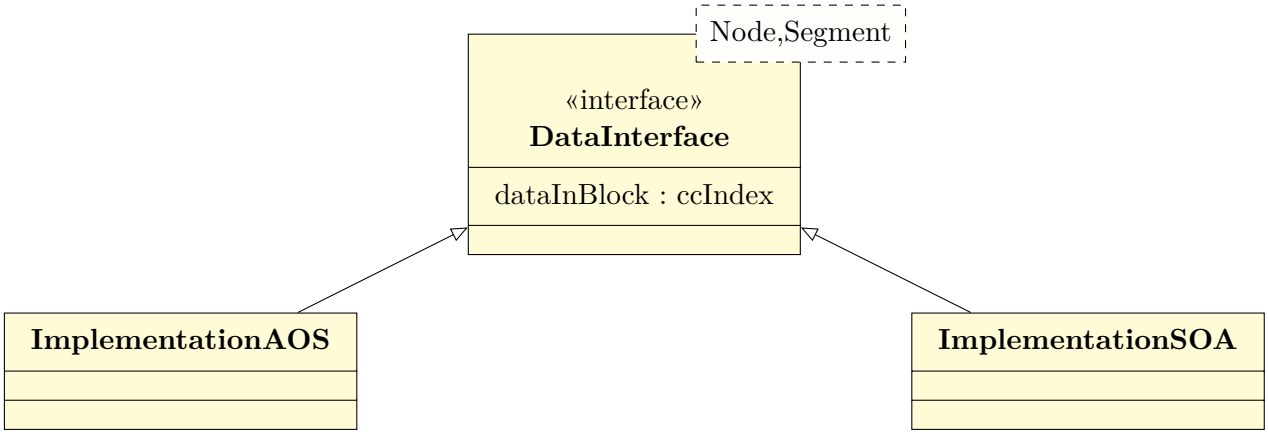


FIGURE 2.5 – Interface générique de la structure

les types *Nœud* et *Segment*, il définit une interface avec l'ensemble des méthodes que le type utilise et un `ccIndex` pour indiquer la position en mémoire de la donnée. Ensuite, une implémentation est faite en héritant de cette interface. Nous avons donc une implémentation pour manipuler les données placées dans un bloc en *SOA* et une autre pour les blocs en *AOS*, qui se modélise comme dans la Figure 2.5. Au final, l'indexation directe entre les segments et les nœuds ainsi que l'utilisation de l'interface rendent les opérations de lecture et d'écriture sur les données uniformes et efficaces.

2.3.4 Primitives d'insertion et de suppression

2.3.4.1 Insertion

Avec l'organisation par blocs, l'insertion n'est plus dynamique car l'espace nécessaire est déjà réservé. Pour chaque bloc, l'espace libre se situe entre l'indice `idxFree` et $B-1$, ainsi aucun appel système n'est nécessaire pour gérer la mémoire. Si le bloc est plein une autre primitive est en charge d'étendre la structure. L'insertion d'une donnée peut se faire de deux façons :

1. **Par clef**. Dans ce cas, il faut parcourir les blocs à l'aide d'une clé par exemple l'indice de Morton pour insérer la donnée dans le bon bloc. Ce type d'insertion n'est pas le plus efficace mais il évite une insertion aléatoire dans la structure notamment lors de l'initialisation de la simulation. Elle permet de grouper les éléments ayant la même clé dans le même bloc. Dans le pire des cas, l'insertion se déroule en $\mathcal{O}(nbBlocs)$ où $nbBlocs$ est le nombre de blocs déjà présents de la structure. Dans notre cas, les données sont organisées selon l'indice de Morton et il suffit de tester simplement le premier élément de chaque bloc pour placer la nouvelle donnée dans un bloc contenant des indices proches. On évite ainsi des comparaisons avec chacun des éléments du bloc.

2. **Par voisinage**. Ce mode d'insertion est privilégié pour son efficacité en terme d'accès et pour maintenir une localité des données. Nous insérons toujours par localité dans le bloc à la position `idxFree` sans perturber l'organisation du bloc. C'est à dire que la donnée à insérer sera toujours placée à la première position libre du bloc sans provoquer un recopie d'une partie des données du bloc. Ce type d'insertion se déroule en $\mathcal{O}(1)$ et favorise des chargements de cache avec des données susceptibles d'être accédées consécutivement.

2.3.4.2 Suppression

Pour des raisons d'efficacité, nous ne supprimons jamais une donnée à la fois mais des ensembles de données. Ainsi, on limite les déplacements de données dans les blocs qui pénalisent les algorithmes. De plus, pour les traitements en parallèle on réduit ainsi l'utilisation de verrous sur les données du maillage. Par conséquent la suppression se fait toujours en deux temps :

Phase 1 si une donnée doit être supprimée, il suffit de l'invalider en faisant disparaître les dépendances des autres données vers celle-ci. Nous stockons son `ccIndex` pour y accéder durant la seconde phase.

Phase 2 On supprime les données invalidées et on déplace les données situées à la fin du bloc pour boucher les trous. En parcourant les indices stockés durant la première phase on cible directement les blocs contenant des données invalidées. Le travail est alors local au bloc et il consiste à réorganiser les données à l'intérieur du bloc pour avoir des données contiguës.

2.3.5 Primitives dynamiques pour étendre ou réduire la structure

Si après la phase de nettoyage un bloc est vide (`idxFree=0`) on doit le supprimer de la liste. Pour cela, un simple déréférencement par le bloc précédent et le bloc suivant est suffisant. Au niveau de la mémoire, un appel à des fonctions systèmes est nécessaire. Pour diminuer le coût de la désallocation et de la réallocation on ajoute à la structure de données (Figure 2.6) une pile pour gérer les blocs vides inutilisés.

```
struct List{
  Block *fisrt, *last;
  int    nbBlocks;
  int freqAlloc;
  Stack freeBlock;
}
```

FIGURE 2.6 – Données de la structure liste avec gestion de la mémoire.

Lors de l'insertion d'une donnée dans un bloc `A`, si celui-ci est plein on insère un nouveau bloc comme dans le cas des listes chaînées. Ce bloc est inséré à la suite du bloc

A. Pour cela, si la pile `freeBlock` n'est pas vide on récupère l'adresse du premier bloc vide disponible sinon on alloue de la mémoire pour un nouveau bloc.

De manière générale, dans nos simulations le réseau de dislocations aura tendance à croître plus ou moins vite au cours d'une simulation. Par exemple, dans un grain contenant une faible densité de boucles avec des sources de Frank-Read très mobiles, on observe une forte croissance puis une stabilisation due à la densification des dislocations. C'est pourquoi, il est important d'optimiser la gestion de la pile et du nombre d'allocations. Pour cela, nous fonctionnons par pool d'allocations de blocs dont la taille `K` évolue selon la fréquence des appels pour demander plus de mémoire. Cette fréquence revient à mesurer le nombre d'itérations entre deux appels à la fonction d'allocation. Si un certain seuil d'allocation est dépassé nous allouons alors un plus grand espace mémoire (`pool`, zone en rouge sur la Figure 2.7) pour réduire le nombre et le coût de ces appels. La taille du `pool`, `poolSize`, est obligatoirement un multiple de la taille d'un bloc : `poolSize` $=$ `K` $\times$ `blockSize` avec `K` qui augmente tel que `K` $=$ `K` $\times 1.5$ et diminue inversement selon la fréquence. Une partie

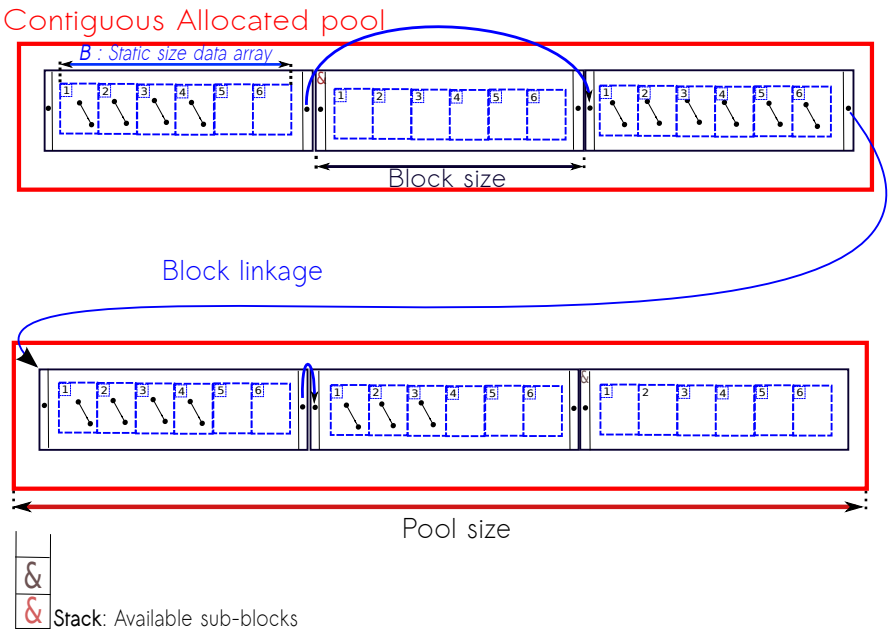


FIGURE 2.7 – Structure de données chaînées par bloc avec allocation par pool pour éviter la fragmentation de la liste, la pile (`Stack`) stocke l'adresse des blocs non encore affectés à la liste.

de cet espace mémoire est directement utilisée pour introduire les nouveaux blocs dans la liste et donc les nouvelles données tandis qu'une autre partie est réservée par anticipation pour plus tard. Les blocs stockés seront disponibles plus rapidement ultérieurement et on évite de trop nombreux et coûteux appels système. Si la simulation est stable le pool retrouve sa taille d'allocation minimale qui correspond à un bloc.

Lorsqu'un bloc de données n'est plus nécessaire à la simulation, il est réinséré dans la pile d'espace mémoire disponible. L'espace supplémentaire stocké dans la pile ne dépasse pas 5 % de l'espace nécessaire à la totalité des blocs contenus dans la liste sinon la mémoire

est libérée.

Le fonctionnement avec un pool d'allocation en plus d'éviter les appels systèmes coûteux lors des phases de croissance de la simulation, réduit aussi la fragmentation de la mémoire inhérent au fonctionnement avec une liste chaînée. Comme on peut le voir dans la Figure 2.7, les blocs sont ainsi contiguës en mémoire pour éviter de larges sauts entre le chaînage des blocs.

2.3.6 Maintien de la cohérence mémoire

Pour maintenir la performance malgré la dynamique, un ensemble de techniques algorithmiques sont mises en place. Pour cela, les techniques abordées dans l'état de l'art Section 0.3.2 ont été développées et adaptées dans le cas du maillage de dislocations à travers les étapes algorithmiques de la simulation. Nous allons revenir sur les caractéristiques de la structure en liant cela directement à la simulation en DD.

2.3.6.1 Organisation hiérarchique de l'espace et réplication de données

Comme présenté dans la section 1.1, la structure spatiale que nous utilisons pour les deux algorithmes de calcul de forces et de collisions est un octree. Il est donc primordial que la connexion entre l'octree et notre structure contenant le maillage soit efficace et cohérente.

Les feuilles de l'octree sont parcourues en suivant la numérotation de Morton qui crée un parcours linéaire de l'espace 3D. Pour attribuer à chaque segment un indice de Morton, nous considérons son barycentre pour évaluer son indice de Morton et son appartenance à une feuille. Cette donnée est inutile au calcul puisque nous utilisons les coordonnées des nœuds dans les algorithmes. Pour minimiser le surcoût de ce stockage, nous enrichissons la structure feuille en ajoutant aussi le vecteur unitaire et la longueur du segment. On peut alors calculer à la volée la position des nœuds du segment sans indirection à travers deux structures de données (vers les segments puis vers les nœuds). Les données stockées dans une feuille de l'octree sont décrites dans la Figure 2.8.

```
struct SegmentInTree{
        double coordBarycentre[3];
        double unitVect[3];
        double length;
        ccIndex segment;
        }
```

FIGURE 2.8 – Représentation des segments dans l'octree.

Avec cette technique de réplication des données, nous augmentons le coût mémoire (4 doubles supplémentaires en plus du *ccIndex* et de la position de la particule *coordBarycentre*) de la simulation mais évitons aussi une double indirection coûteuse. Cette façon de procéder réduit les accès non réguliers en mémoire et permet d'éviter les risques de limitation des algorithmes par la bande passante mémoire.

Enfin, cette technique facilite le calcul avec des conditions périodiques sur l'octree. En effet, lors du calcul des forces ou la détection des collisions en conditions périodiques les feuilles au bord du domaine doivent être copiées et translatées de l'autre côté de la boîte de simulation ainsi que les coordonnées des nœuds. L'algorithme déplace alors simplement l'attribut `coordBarycentre` et ajoute la contribution sur le segment cible de manière transparente par le `ccIndex`. Avec la réplication de données, cette étape se fait à moindre coût, et les indirections vers les attributs statiques (vecteur de Burgers, plan de glissement, vecteur vitesse) se font aussi via le `ccIndex`.

2.3.6.2 Insertion avec maintien de la localité

L'insertion de nœuds ou de segments est une étape fréquente et se produit de manière aléatoire sur tout le maillage. Elle doit être rapide tout en maintenant la localité des données. Cette localité est importante pour être efficace tant sur les parcours de la structure que lors des traitements en parallèle. Nous insérons un nouvel élément dans notre maillage local à différentes étapes de l'itération :

— Collision entre lignes de dislocation ;
— Séparation de nœuds physique ;
— Raffinement de lignes de dislocation ;
— Migration entre les processus.

L'insertion de nouveaux éléments dans la structure conduit naturellement à perturber l'ordre des données. Cet ordre doit être conservé pour maintenir l'efficacité. Au cours des différentes étapes la localité des données est nécessaire. Le calcul des forces (Section 1.1.1) et la détection des collisions (Section 1.1.2.1) nécessitent une séparation entre les segments proches contenus dans les feuilles voisines et les segments lointains. L'organisation des données doit donc toujours prendre en compte cette localité basée sur l'indice de Morton. Ensuite, le calcul de vitesse (Section 0.2.3), la séparation des nœuds physiques et le remaillage font intervenir la notion de localité par interconnexion entre les segments en se basant sur le `ccIndex` et sur la correspondance entre la structure des nœuds et celle des segments.

Pour l'étape de séparation des nœuds physiques (Algorithme 5), nous ne considérons que les nœuds physiques ayant au moins trois connexions. Soit `A` un nœud physique à séparer, il doit disparaître pour former deux nouveaux nœuds qui seront connectés aux mêmes nœuds et segments que `A`. En nous basant simplement sur le `ccIndex` du nœud `A` nous savons où insérer dans la structure les nouvelles données à l'aide des primitives d'insertion. Les nouveaux nœuds sont insérés dans le bloc du nœud physique A et s'il faut insérer un segment de jonction nous accédons au bloc contenant les segments connectés au nœud `A`.

L'étape de raffinement reprend le même principe dans le sens où, suite à un parcours de la structure contenant les segments, l'insertion se fait par voisinage. Nous utilisons le `ccIndex` du segment à raffiner et les `ccIndex` de ses nœuds pour insérer en temps constant tout en conservant la localité pour le nouveau segment et le nouveau nœud.

L'insertion suite à une collision est un peu différente car nous ne travaillons plus à partir de notre structure de données contenant le maillage mais avec la structure hiérar-

chique (octree). Les collisions ne se produisent pas uniquement entre les segments d'une feuilles mais aussi avec des segments dans les feuilles voisines. Donc, les données peuvent avoir des indices de Morton différents et être contenus dans des blocs différents. Dans ce cas, l'insertion se base en plus sur l'indice de Morton attribué à la donnée à insérer par la position de la collision. En comparant cet indice avec ceux des segments impliqués dans la collision on récupère le bloc possédant des segments proches.

Enfin, la migration est la dernière étape provoquant des insertions de données. La migration s'opère tout au long des itérations de la simulation avec le déplacement des segments à l'intérieur du domaine de simulation. La migration à l'intérieur d'un unique domaine n'entraîne pas d'insertion dans les structures mais plutôt une modification de la localité des données dans l'espace et par conséquent de l'ordre de parcours pour les différents algorithmes. Le processus de déplacement est relativement lent ainsi la réorganisation se fait par un processus de tri sur l'indice de Morton (section 2.3.6.3) qui rétablit la relation entre l'octree et les segments dans la structure. Par contre dans le cas parallèle, la migration au sens du déplacement entre les mémoires de deux processus est une source d'insertion dans nos structures de données. Lorsque des segments quittent une mémoire d'un processus pour passer à un autre processus, nous devons procéder à l'insertion dans la structure locale. Pour cela, on cherche dans l'octree la feuille contenant les segments avec le même indice de Morton. Puis on récupère le `ccIndex` d'un segment de cette feuille pour insérer dans le bloc contenant des données proches spatialement. Ensuite, on répète le processus pour placer les nœuds dans le bloc cible à partir des extrémités du segment voisin. Dans le cas où la feuille n'existe pas, nous prenons alors parmi les feuilles voisines, le premier segment avec l'indice le plus proche.

2.3.6.3 Tri et mise en correspondance des structures

Malgré la prise en compte de la localité des données lors des opérations d'insertion et de suppression le déplacement des segments et des nœuds au cours de la simulation empêche d'avoir un contrôle fin pour maintenir la cohérence entre le placement dans les blocs et l'organisation spatiale dans la boîte de simulation. L'organisation des données est faite à trois niveaux avec l'octree, la liste des segments et la liste de nœuds. Or, ces trois structures utilisent des indirections de l'une vers l'autre ce qui rend capital une bonne correspondance pour éviter les accès aléatoires à la mémoire. Puisqu'au cours des différentes étapes, les algorithmes n'interviennent pas sur les mêmes structures de données et que celles-ci sont interdépendantes, il faut rendre l'ensemble cohérent pour que les accès par indirection favorise une bonne utilisation du cache. Comme on peut le voir sur la Figure 2.9, la correspondance entre deux structures comme par exemple entre les segments et les nœuds rend les indirections critiques pour la performance. On constate de manière évidente qu'un parcours au niveau des segments en accédant aux nœuds provoque une forte dégradation de l'utilisation des caches avec des accès aléatoires sur les nœuds. En parallèle, ces mauvais chargements de cache deviennent encore plus critiques lorsque les accès se font entre deux threads concurrents qui invalident les lignes chargées par un autre thread. Par conséquent, nous ajoutons un algorithme de tri sur

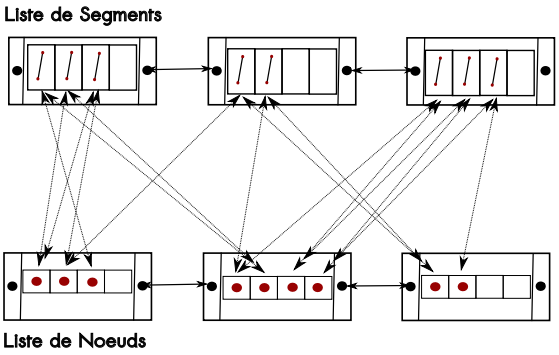


FIGURE 2.9 – Accès aléatoire en mémoire aux nœuds depuis la structure de données des segments.

les structures pour garder la correspondance entre les patterns d'accès aux données des différents algorithmes et l'organisation en mémoire de celles-ci. De manière statique tous

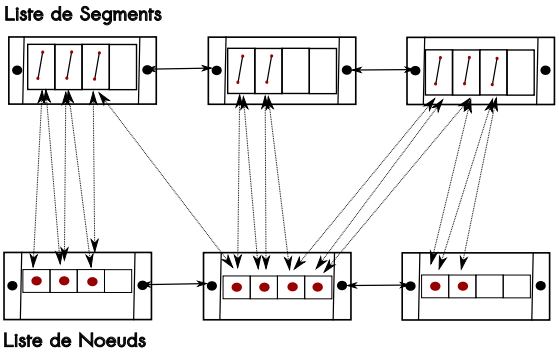


FIGURE 2.10 – Accès réguliers aux nœuds depuis la structure de données des segments.

les X pas de temps une étape de tri crée cette correspondance entre les trois structures pour retrouver un état où les accès sont bien coalescents. La Figure 2.10 montre que chaque segment référence des données proches en mémoire ce qui favorise l'utilisation du cache.

L'algorithme se décompose en 4 étapes principales comme le présente la Figure 2.11 en intervenant sur les différentes structures de données.

Étape 1 : Mise à jour des indices de Morton. La première étape consiste simplement à mettre à jour l'indice de Morton des segments en prenant le barycentre comme

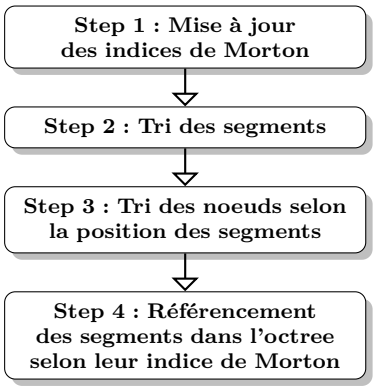


FIGURE 2.11 – Principales étapes pour l'algorithme de tri

position de référence. Cette étape a une complexité linéaire en fonction du nombre de segments. L'indice de Morton sera la clef pour trier les données.

Étape 2 : Tri des segments. Un premier tri est appliqué sur la structure contenant les segments en se basant sur les indices de Morton. L'algorithme de tri est calqué sur l'approche classique du tri dans les méthodes cache oblivious avec une décomposition récursive de la structure. Il s'agit d'un tri de type *quickSort* par bloc en adaptant la récursion à la structure. La récursion s'arrête, soit si un bloc ne contient plus qu'un seul indice de Morton, soit s'il est considéré comme suffisamment trié c'est à dire avec des indices de Morton contiguës. Le tri est initié avec l'algorithme 10 où nous fixons la profondeur maximale en appelant la fonction *"ComputeDepth"*. Cette fonction prend en

Algorithme 10 : Tri par bloc structure des segments.

Données : MeshClass, nbThreads

```
// Premier élément de la liste
1 CCIndex first ← Mesh.getSegments().getHead();
// Dernier élément de la liste
2 CCIndex last ← Mesh.getSegments().getEnd();
// Détermine la profondeur de la récursion
3 int depth ← ComputeDepth(Mesh.getSegments(), nbThreads);
4 QuickSortTask(head,last,depth);
```

compte le nombre de segments et le nombre de blocs pour fixer jusqu'à quel niveau il faut remonter dans l'arbre virtuel du tri. Une fois l'algorithme 11 démarré (appel à la ligne 4 algorithme 10), c'est lors de l'appel à la fonction *"Divide"* que nous pouvons décider de stopper la récursion. Comme le tri est une opération régulière et que le mouvement des segments est permanent, il est inutile de payer le prix d'un tri exact.

Étape 3 : Tri des nœuds. Une deuxième étape de tri est effectuée sur la structure des nœuds. Elle se base sur un parcours de la structure contenant les segments qui viennent d'être triés à l'étape 2. L'algorithme décrit dans Algorithme 12 déplace simplement les

Algorithme 11 : Appel récursif du tri sur la structure et technique d'arrêt.

Données : head,last,depth

```
// Trouve l'élément milieu et utilise le pivot sélectionné (Indice de
   Morton) pour placer les éléments inférieurs au pivot dans la
   partie gauche
```

1 $CCIndex\, half \leftarrow Divide(first, last)$;

```
// Détermine si la profondeur maximale a été atteinte
```

2 **si** *depth>1* **alors**

3 QuickSortTask(head,half,depth-1);

4 QuickSortTask(half,last,depth-1);

nœuds dans l'ordre dans lequel ils sont appelés depuis la structure des segments, pour retrouver exactement la même correspondance que celle montrée dans la Figure 2.10.

Algorithme 12 : Correspondance entre les structures selon les positions (Morton Index) des segments.

Données :

```
// Récupérer la première position de la liste de nœuds
```

1 $CurrentNode \leftarrow MeshClass.getNodes().getFirst()$;

2 **pour chaque** $Segment \in SegmentList$ **faire**

```
   // Vérifie la position du nœud par rapport au ccIndex courant
```

3 **si** *! Segment.getNode1().isInPlace()* **alors**

```
      // Inverse la position si nécessaire
```

4 $Swap(Segment.getNode1(), CurrentNode)$;

5 $CurrentNode++$;

6 **si** *! Segment.getNode2().isInPlace()* **alors**

7 $Swap(Segment.getNode2(), CurrentNode)$;

8 $CurrentNode++$;

Lorsqu'un nœud est accédé par indirection depuis un segment la fonction `isInPlace` (ligne 3 et 6 Algorithme 12) teste la position du nœud dans la structure par rapport au bloc courant pour insérer un nœud. Dans le pire des cas, pour N nœuds nous avons N déplacements.

Étape 4 : Mise à jour de l'octree. Nous terminons en mettant à jour l'octree à partir d'un parcours transversal de la structure des segments triés. Puisque les segments sont groupés par indices de Morton contiguës, l'insertion dans chaque feuille de l'octree est groupée. Ainsi, pour N segments nous n'avons plus à parcourir N fois l'arbre en profondeur pour accéder à une feuille mais seulement à parcourir les feuilles allouées dans l'ordre pour y placer les segments.

L'organisation des données est donc descendante depuis l'octree vers les nœuds, en nous basant sur la localité spatiale des segments dans l'arbre grâce aux indices de Morton. Nous organisons les segments dans leurs blocs, pour créer une correspondance dans l'organisation des nœuds afin d'avoir des accès coalescents aux nœuds en parcourant les segments.

Bilan

Dans ce chapitre nous avons présenté les travaux effectués sur l'élaboration des structures de données qui permettent de gérer de manière efficace le maillage des dislocations. L'introduction d'une liste chaînée par blocs pour gérer le maillage permet d'être performant à la fois dans les phases à forte intensité arithmétique (FMM et détection de collisions) et lors de la modification du maillage (dynamisme des données - insertions/suppressions). Nous avons présenté l'ensemble des primitives adaptées à la DD pour rendre compatible le comportement de la structure avec le déplacement des segments dans un contexte de calcul hautes performances.

Nous avons aussi mis en avant la prise en compte de l'évolution des architectures matériel avec une structure adaptée au différents niveaux de caches (hiérarchie mémoire) et implémenté à travers une interface pour utiliser un stockage en structure de tableau pour réduire la contention mémoire et éviter la surcharge des caches.

Enfin, nous avons présenté les techniques algorithmiques pour gérer la cohérence avec trois structures interdépendantes à savoir l'octree, la liste de segments et la liste de nœuds. Ces techniques visent à réduire le coût des indirections, notamment avec la prise en compte de la localité pour l'insertion, la réplication de données pour le calcul dans l'octree et enfin le tri global pour maintenir la l'organisation en mémoire avec les accès aux données par indirection.

Dans le chapitre suivant, nous aborderons les développement effectué pour introduire un parallélisme hybride Open/MPI à la simulation. Ce parallélisme s'inscrit comme le premier jalon vers les calculs à très grande échelle.

Chapitre 3

Programmation parallèle en DD

Sommaire

Nous présentons ici les méthodes de parallélisation employées pour l'ensemble des algorithmes présentés dans les sections précédentes. Ces travaux sont la première étape pour aller vers un parallélisme massif sur des machines hétérogènes (manycore) et s'orientent vers un parallélisme hybride classique. Nous présentons un premier niveau de parallélisme fin basé sur la programmation OpenMP en utilisant un paradigme par tâches. Puis un niveau grossier avec une décomposition du domaine de simulation en boîtes pour un parallélisme distribué en utilisant un paradigme par échanges de messages basé sur MPI.

3.1 Espace d'adressage global : OpenMP

3.1.1 Parcours de la structure de données

Les algorithmes du calcul des vitesses et de réduction des contributions des segments sur les nœuds s'effectuent en parcourant la structure de données par blocs. On doit donc

paralléliser le parcours de la structure de données. Pour cela, on itère sur les blocs de la structure et on définit une tâche pour effectuer une opération sur un bloc de données. En fonction du coût de calcul de l'opération, on peut augmenter la granularité en donnant plusieurs blocs, `nbBlockPerTask`, à la tâche pour diminuer le nombre de tâches et ainsi masquer le surcoût de la création et de l'ordonnancement des threads qui exécutent tâches. Le principal paramètre pour régler la granularité du calcul est la variable `nbBlockPerTask`

Algorithme 13 : Parallélisation du parcours de la structure chaînée.

```
   Données : MeshClass , nbBlockPerTask
1  DataBlock currentblock ← MeshClass.getFirstDataBlock();
2  #pragma omp parallel
3      #pragma omp single
4          tant que not treated each block of the mesh faire
5              int nbBlockToTreat← 0;
6              DataBlock firstBlock← currentblock;
7              tant que nbBlockToTreat<nbBlockPerTask faire
8                  currentblock← currentblock.goToNextBlock();
9              #pragma omp task firstprivate(firstBlock, nbBlockToTreat)
10                 tant que not iterate over nbBlockToTreat faire
11                     COMPUTE ON EACH DATA OF currentBlock
12                     ⋮
13                     currentBlock← currentBlock.goToNextBlock();
```

(Algorithme 13). Elle correspond au nombre de blocs consécutifs attribués à la tâche. Cette variable `nbBlockPerTask` dépend du nombre de données dans la liste, du nombre de blocs et du nombre de données par bloc qui évoluent au cours de la simulation. L'attribution d'un seul bloc par tâche est trop coûteux pour l'ordonnanceur et surtout trop irrégulier du fait du remplissage non homogène des blocs. En conséquence, nous considérons le problème inverse et nous fixons le nombre de tâches par thread en groupant des ensembles de blocs contiguës. De plus, pour limiter l'impact de l'hétérogénéité du remplissage des blocs nous cherchons à distribuer plus d'une tâche par thread pour mieux répartir les coûts de calcul.

Pour les algorithmes statiques qui n'accèdent pas par indirections aux données référencées dans la structure (des nœuds vers les segments ou inversement) les tâches sont parfaitement indépendantes et sans conflits d'accès mémoire. Le seul risque de concurrence entre les threads est dû à des chargements conflictuels de bancs de cache. Pour cela, en affectant de grandes tâches (groupe de blocs) par thread et en considérant les optimisations pour l'organisation en mémoire de la structure de données (Chapitre 2), ce risque est réduit au minimum.

Cependant, l'indépendance et la cohérence des accès pour les algorithmes statiques sont remises en question pour les algorithmes dynamiques lors de l'insertion et de la suppression de données. Dans la suite nous allons voir comment avec deux mécanismes à base de verrous, un pour l'insertion de nouveaux blocs de données et l'autre pour

l'insertion de données, nous pouvons travailler en parallèle sans blocage entre les threads lors de l'étape de remaillage.

3.1.2 Manipulation threads safe pour des algorithmes dynamiques

Pour remailler nous appliquons un algorithme en deux étapes. Une première étape pour faire disparaître tous les segments trop longs avant d'effectuer une seconde passe pour dé-raffiner en supprimant les segments trop petits comme présenté dans la Section 1.3 Algorithme 6. Pour raffiner (ligne 2 Algorithme 6), en terme de manipulation,

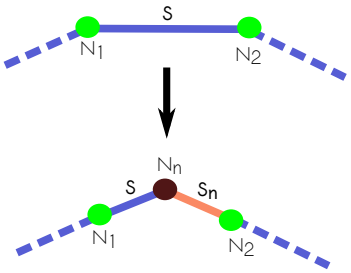


FIGURE 3.1 – Raffinement d'un segment. Création d'un segment S_n et un nœud N_n.

il faut insérer un nouveau segment S_n et un nouveau nœud N_n comme montré dans la Figure 3.1. L'insertion dans la structure segment est évidente puisque un seul et unique thread travaille sur chaque bloc. Pour les nœuds, avec les indirections, deux threads travaillant sur des segments différents peuvent accéder à des nœuds contenus dans un même bloc. Cela peut conduire à des écritures concurrentes dans le bloc pour insérer le nouveau nœud. De même, si deux threads travaillent sur deux segments interconnectés il y a potentiellement un conflit lors de la modification des connections en parallèle.

Pour éviter ces conflits sur les données et pour les modifications de la structure de données en parallèle, nous utilisons deux types de verrous. Un premier verrou (`flagInUse`) sur chaque donnée (segment ou nœud) dès lors que l'on accède à l'une d'elle pour notifier qu'un thread est en lecture sur cette donnée. Ainsi la fusion avec le nœud voisin par un autre thread est protégée tant qu'un thread est en lecture. Un second verrou sur chaque bloc (`flagModif`) si l'on souhaite modifier le maillage local.

Les verrous sont en réalité des booléens encapsulés dans une classe manipulée en sécurité dans le cadre multithreads à l'aide des fonctions OpenMP omp_set/unset_lock(). Pour la manipulation de structures chaînées, nous trouvons de nombreux travaux sur les structures *lock free* dans la littérature [57] dont nous nous sommes inspirés. Nous adaptons ces travaux pour les phases d'insertion et de suppression des données.

Pour gérer ces accès concurrents, chaque bloc de la structure de données contient un flag, `flagModif`, de modification. Il est pris si un thread doit insérer une donnée dans le bloc. Si une position est libre dans le bloc, le thread relâche le `flagModif` dès qu'il a incrémenté l'indice de la première position libre (`idxFree`). Si un autre thread était en attente, il peut alors récupérer le `flagModif` et incrémenter à son tour la première position libre. Pendant ce temps, le premier thread stocke la donnée en établissant les connections (*ccIndex*) avec le reste du maillage. Si un bloc A de la structure est complet,

le thread va piocher un bloc B dans le `pool` de blocs libres et le connecte à la suite tout en gardant le `flagModif` du bloc A jusqu'à avoir réservé la première position du nouveau bloc B. Lors de l'insertion, le bloc B est identifié par un booléen (`newBloc`) comme étant nouveau. Un fois le verrou `flagModif` relâché, un autre thread qui attendait pour mettre une donnée dans le bloc A plein s'en empare et accède par chaînage au nouveau bloc B et réalise l'insertion comme présenté dans la Figure 3.2.

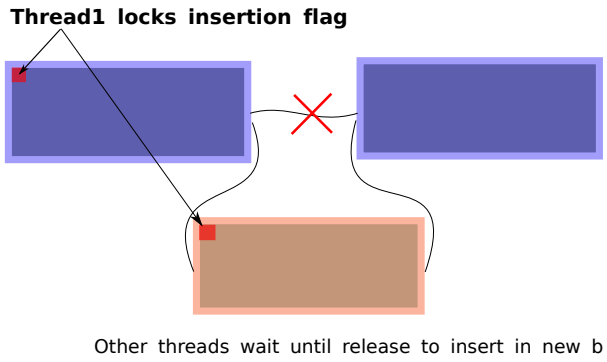


FIGURE 3.2 – Verrouillage du flag de manipulation, insertion d'un nouveau bloc (orange) avec modification des connexions et libération du flag pour autoriser l'insertion par un autre thread.

Concernant les dépendances entre les nœuds et les segments, nous avons la garantie qu'un segment ne sera manipulé que par 1 thread. Il suffit donc de prendre le `flag` de lecture sur les deux nœuds N_1 et N_2 de ce segment pour modifier les connexions. L'opération pour raffiner correspond à introduire un nœud de discrétisation, donc il n'y a pas de modification du nombre de connexions pour N_1 et N_2 ainsi aucune précaution particulière n'est à prendre pour la concurrence entre les threads.

Pour dé-raffiner, il faut supprimer le nœud N_2 et le segment S comme montré dans la Figure 3.3. Nous devons dans chacune des structures supprimer une donnée et ses dépendances. Pour éviter les interblocages entre les threads les données sont dans un

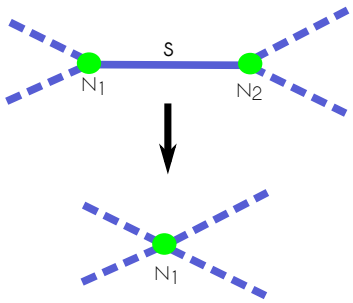


FIGURE 3.3 – Dé-Raffiner du segment S. Suppression de S et fusion N_1 et N_2.

premier temps invalidées (cf. Section 2).

Pour supprimer un segment, on fusionne ses deux nœuds selon la méthode décrite dans la Section 1.3. Une fois la décision de supprimer le segment S est prise, les deux nœuds N_1 et N_2 sont protégés avec la prise du verrou `flagModif` pour qu'aucun autre thread ne

puisse fusionner les nœuds du segment précédent ou du segment suivant pendant cette opération. Au niveau des dépendances, en prenant l'exemple de la fusion de N_2 dans N_1, N_1 reçoit toutes les connexions de N_2 sauf S. Lors de la fusion, un nœud de discrétisation ne reçoit qu'un seul segment tandis que lors de la fusion de deux nœuds physiques N_1 peut recevoir plus d'un segment.

La prise en compte de la courbure de la ligne complexifie les manipulations précédentes car on travaille avec deux segments au lieu d'un. Le mécanisme décrit précédemment s'adapte et supprime les interblocages.

3.1.3 Amélioration de la parallélisation du champ direct

Notre algorithme de calcul du champ de force travaille en différentes étapes pour évaluer la force sur chaque nœud du système. Lors de l'étape de séparation des nœuds physiques nous devons mesurer la dissipation d'énergie pour chaque configuration ce qui nécessite de connaitre les contributions des forces sur chaque segment. Pour réduire au minimum le risque de conflits d'écriture (réduction sur un nœud), nous conservons les contributions des forces sur chaque segment. Le surcoût mémoire de cette approche ajoute deux `double` pour stocker les contributions sur chaque extrémité des segments dans la structure hiérarchique (Figure 2.8). Enfin, la dernière étape du calcul consiste à distribuer sur les nœuds ces contributions stockées sur chaque réplication de segment. Le calcul des

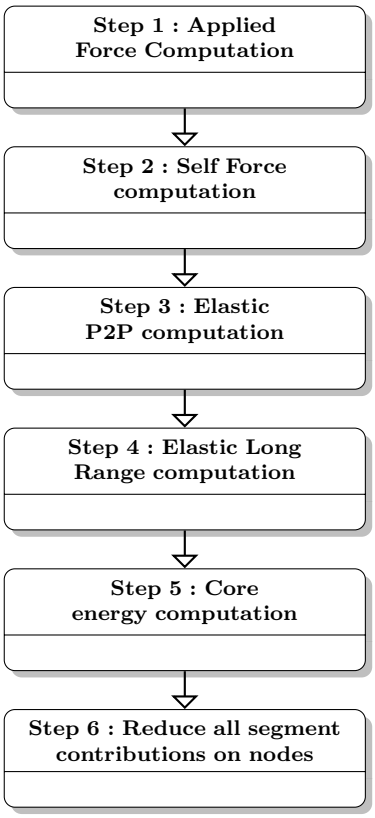


FIGURE 3.4 – Principales étapes du calcul des forces.

forces sur chaque nœud nécessite les six étapes suivantes comme montré dans la Figure 3.4 :

Étape 1 : La force appliquée. Nous calculons sur chaque segment la force en réponse à la contrainte appliquée sur le cristal. Cet algorithme itère sur les segments avec une complexité linéaire $\mathcal{O}(N)$, où N est le nombre de segments. La contribution est stockée au niveau du segment.

Étape 2 : La force élastique du segment sur lui même. Nous calculons la contrainte élastique d'un segment sur lui-même avec une complexité linéaire $\mathcal{O}(N)$. La contribution est stockée au niveau du segment en sommant avec la contribution précédente donc sans risque de conflits d'écriture. Dans l'algorithme cette étape est effectué lors des accès aux segments de l'étape 3 (*P2P*).

Étape 3 : La force élastique due aux interactions proches. Il s'agit de l'opérateur *P2P*. Le calcul pour une paire ne doit être fait qu'une fois pour obtenir la contribution sur les 4 nœuds. A cause de la réciprocité du calcul, les conflits d'écriture sont nombreux puisque deux feuilles même non directement voisines peuvent écrire sur les mêmes segments (Figure 3.5).

Étape 4 : La force élastique due au champ lointain. L'algorithme FMM travaille au niveau des cellules de l'octree pour obtenir sur les segments les contributions du champ lointain. Ces contributions sont stockées de manière séparée sur chaque segment puisque notre algorithme prend en compte l'évolution lente du champ lointain (Chapitre 1). Cette contribution est stockée sans risque de conflits d'écriture et peut être utilisée sur plusieurs itérations.

Étape 5 : La force de cœur. Calculée sur chaque segment avec une complexité linéaire $\mathcal{O}(N)$. De la même façon que pour la force appliquée, en ne considérant que ce calcul nous n'avons pas de risques de conflits d'écriture.

Étape 6 : Réduction. Les contributions sont stockées sur les segments, il faut alors pour chaque nœud, parcourir l'ensemble des segments qui lui sont connectés pour sommer les contributions proches et lointaines pour finalement obtenir la force nodale. Cette algorithme ne soulève pas de conflits d'écriture.

Pour les étapes 1, 5, et 6 le parallélisme est basé sur des tâches OpenMP avec le parcours en parallèle sur la structure de données chaînée par blocs. Nous allons maintenant présenter le parallélisme de ScalFMM pour le champ proche avec les contributions apportées à la DD qui s'appliquent pour le calcul des forces et pour la détection des collisions.

Parallélisme ScalFMM : indépendance des calculs

Pour le calcul du champ élastique et pour celui de la détection des collisions nous utilisons ScalFMM. Il s'agit d'un calcul des interactions proches (opérateur *P2P*) dans ces deux algorithmes. Le parallélisme est basé sur la décomposition hiérarchique de la boîte de simulation en octree. Puisque nous calculons des interactions mutuelles nous écrivons le résultat dans les deux feuilles concernées. Par conséquent, les écritures en mémoire des contributions peuvent être concurrentes et créer des conflits aux interfaces des cellules traitées par des threads différents (Figure 3.5). Dans ScalFMM l'indépendance des données est assurée par un algorithme de coloriage qui permet de séparer le calcul

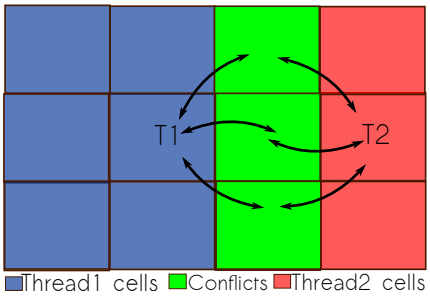
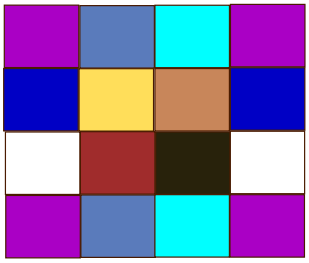
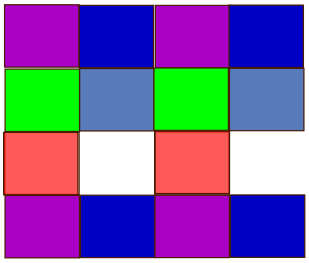


FIGURE 3.5 – Conflits d'écriture entre deux threads pour le calcul du P2P.

pour tous les voisins directs afin d'assurer la non concurrence des écritures. Les noyaux de ScalFMM sont génériques et doivent donc s'adapter à tous les cas de figure de calcul direct. Par conséquent, la séparation pour une même couleur est assurée dans toutes les directions comme on peut le voir dans la Figure 3.6a. Cela signifie que pour une grille



(a) Coloriage avec indépendance complète dans toutes les direction. Ce coloriage utilise 9 couleurs.



(b) Coloriage avec indépendance uniquement dans certaines directions pour réduire le nombre de couleurs utilisées

FIGURE 3.6 – Coloriage sur une grille 2D.

$2D$ nous sommes obligés d'utiliser 9 couleurs pour séparer les feuilles indépendantes et 27 couleurs pour une grille $3D$. Ainsi, entre chaque couleur l'ensemble des threads doivent se synchroniser pour pouvoir passer à la couleur suivante sans créer de conflits.

Dans notre cas, nous ne faisons qu'un seul calcul entre un segment S_1 et un segment S_2 pour obtenir les contributions sur les 4 extrémités. Donc en écrivant notre noyau de telle sorte que seules les feuilles au $Nord - Est$ (Cellules grises Figure 3.7) soient prises en compte par la feuille courante, nous pouvons réduire le nombre de couleurs utilisées en retirant l'indépendance dans une direction.

Cela nous apporte deux contributions avec d'un côté moins de barrières de synchronisation et de l'autre une meilleur intensité arithmétique pour chaque couleur.

Équilibrage de charge dynamique entre les feuilles

En réduisant le nombre de couleurs, nous avons une plus importante charge de calcul pour chaque couleur pouvant ainsi réduire l'hétérogénéité du coût des calculs entre les couleurs. Cependant, la répartition hétérogène des dislocations dans le domaine de simulation fait qu'une couleur peut contenir très peu de feuilles tandis qu'une autre est

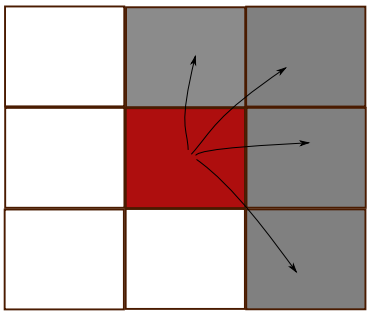


FIGURE 3.7 – Le thread verrouille la feuille rouge et travaille uniquement avec les feuilles au Nord-Est grisée.

beaucoup plus chargée. De plus, le déplacement des segments contribue au déséquilibre entre les couleurs. Nous avons alors des couleurs quasiment vides (peu de feuilles par couleurs) à un moment de la simulation qui se retrouve très chargées par la suite et inversement. Enfin, le coût de calcul par feuille varie selon le nombre de feuilles voisines et la charge de celles-ci (dépendant du nombre de segments contenus). Cela se traduit par de moins bonnes performances puisque dans ScalFMM la distribution des feuilles par couleur et par thread est statique (Algorithme 14) ce qui ne permet pas d'assurer un bon équilibrage de la charge entre les threads. Ceux-ci peuvent rester en attente longtemps du fait d'une mauvaise distribution. En effet, l'ordonnanceur statique assigne un groupe de

Algorithme 14 : Ordonnancement statique.

Données : Octree LocalTree

```
// Distribute leaves among different color
Color ← LocalTree.spreadLeaves();
pour chaque Color ∈ ColorNeigbors faire
    #pragma omp for
        pour chaque Leaf ∈ Color faire
            Kernel.P2P(Leaf);
```

feuille (`chunk`) à chaque thread pour traiter la totalité de la boucle. La taille du `chunk` étant fixe et indépendante du nombre de threads et du nombre de feuilles par couleur, certains threads se retrouvent en attente des autres avant de pouvoir passer à la couleur suivante.

Pour résoudre ce problème, tout en restant avec la même distribution des feuilles par couleur, on paramètre la granularité du travail donné à chaque thread par l'ordonnanceur d'OpenMP. La granularité (`chunk`) dépend du nombre de threads, $nbThreads$, qui vont travailler de manière concurrente sur les feuilles d'une même couleur et du nombre de feuilles, $nbLeaves$, par couleur. Le réglage du `chunk` revient à distribuer des groupes de feuilles sur ce nombre fixe de threads. On calcule alors pour chaque couleur la taille du `chunk` qui permet d'assigner au moins un groupe de feuilles par thread. Notons $nbLeaves(i)$ le nombre de feuilles de la couleur i alors la taille du paquet est évalué pour

chaque couleur par

$$chunkSize = \frac{nbLeaves(i)}{nbThreads} + 1.$$

Cependant, selon la densité de dislocations et la hauteur de l'arbre, nous avons un coût de calcul entre une feuille et ses 13 voisines (Nord-Est) qui peut varier fortement. Pour réduire ces différences, il est préférable de prendre en compte le nombre d'interactions en comptant le nombre de feuilles voisines par couleur. Si pour une couleur i le nombre de feuilles voisines est élevé un seul `chunk` est distribué par thread. Au contraire si ce nombre est faible cela signifie que la distribution des segments est hétérogène donc plusieurs `chunk` sont assignés. Selon ce critère fixé de manière expérimentale, nous obtenons alors un coefficient de granularité ($coefG(i)$) qui va fixer le nombre de `chunk` par threads. Nous avons au final soit 1,2 ou 3 `chunk` par thread et par couleur :

$$chunkSize(i) = \frac{nbLeaves_(i)}{nbThreads \times coefG(i)} + 1.$$

Pour l'ordonnancement dynamique, nous ajoutons donc la phase de pré-calcul (Algo-

Algorithme 15 : Ordonnancement dynamique par couleur.

```
   Données : Octree LocalTree
   // Distribute leaves among different color
1  ColorNeigbors ← LocalTree.spreadLeaves();
2  chunkColor ← ColorNeigbors.computeChunk();
3  pour chaque Color ∈ ColorNeigbors faire
       // Directive pour appliquer le nouveau chunk à la couleur idxColor
4      #pragma omp for schedule(schedule,chunkColor[idxColor])
5      pour chaque Leaf ∈ Color faire
6          Kernel.P2P(Leaf);
```

rithme 15 ligne 2) qui va déterminer pour chaque couleur la taille du chunk selon la méthode décrite précédemment. Ensuite, nous n'avons plus qu'à appliquer cette taille de manière dynamique à chaque changement de couleur pour optimiser la répartition des threads (Algorithme 15 ligne 4).

Ces améliorations introduites dans ScalFMM optimisent le calcul des interactions proches notamment pour un cas hétérogène sur les parties de calcul intensif (force et collision) tout en conservant la même décomposition hiérarchique du domaine.

Dans cette partie, nous avons montré les différentes approches en parallèle pour assurer d'un côté l'indépendance des calculs et de l'autre comment par un mécanisme de verrouillage minimal au niveau des données et au niveau de la structure de données, nous avons permis la modification en parallèle du maillage malgré l'interdépendance des données.

3.2 Mémoire distribuée : Message Passing Interface

Dans un modèle avec un espace d'adressage disjoint, nous décomposons l'espace de simulation en sous domaines et chaque sous domaine est affecté à un processus. Les segments vont alors migrer d'un processus à un autre. En suivant la décomposition hiérarchique du domaine de simulation selon l'octree, on répartit les feuilles entre les différents sous domaines pour distribuer la charge de calcul.

Pour être performante, la décomposition de domaine nécessite :

— Une bonne répartition de la charge. Chaque sous domaine reçoit une partie des feuilles et donc une partie des segments du maillage. Le découpage feuilles en sous domaines doit équitablement distribuer les segments pour que la charge de calcul affectée à chaque processus soit identique pour minimiser les synchronisations entre les processus. De plus, au cours des itérations, la distribution initiale peut devenir déséquilibrée de sorte que l'on doit rééquilibrer la distribution des segments tout au long de la simulation.

— Le deuxième point concerne la prise en compte des interdépendances entre les sous domaines. Cette dépendance s'opère à deux niveaux. Tout d'abord, les algorithmes du calcul des forces et de la détection des collisions d'un sous-domaine nécessitent d'accéder aux segments des sous-domaines voisins. Le second niveau d'interdépendance est lié à la nature des objets simulés qui ne sont pas ponctuels. Une ligne parcourt le domaine de simulation et peut appartenir à plusieurs sous domaines ainsi les segments peuvent être partagés entre deux ou trois sous domaines. Les algorithmes avec prise en compte des connexions directes comme le remaillage ou le calcul de vitesse doivent créer une correspondance au niveau des interfaces notamment en faisant apparaître la notion de données *fantômes*.

Nous allons maintenant aborder la décomposition de domaine choisie pour la simulation en DD et présenter la notion de données fantômes. Enfin nous détaillerons les implications pour les algorithmes et nous discuterons de l'étape de migration.

3.2.1 Décomposition de domaine

L'algorithme de calcul du champ de contrainte élastique représente 85% du temps d'une itération et est particulièrement affecté en distribué par la répartition de la charge ainsi que de l'interdépendance des sous domaines. Nous avons un coût de calcul local basé sur le nombre de segments répartis dans les feuilles et les communications dépendent du nombre d'interfaces directes au niveau des feuilles pour le champ proche mais aussi de la distribution des cellules pour le champ lointain.

Comme cela a été mentionné, la bibliothèque ScalFMM est en charge de cet algorithme et la répartition des calculs est basée sur les intervalles d'indices de Morton (mapping de l'espace 3d sur 1 dimension). Ainsi pour l'ensemble de la décomposition en sous domaines, la distribution est faite en répartissant des intervalles contiguës d'indices de Morton,

$I_i = [a_i, b_i]$ avec $I_i \cap I_j = \varnothing$, entre les processeurs.

$$\Omega = \bigcup_{i=0}^{N-1} D_i\,, \quad \text{avec } D_i = \bigcup_{m \in I_i} F_m,$$

où le domaine de simulation Ω est décomposé en sous domaines D_i. Chaque sous domaine D_i regroupe les feuilles F ayant un indice de Morton compris entre a_i à b_i. Comme

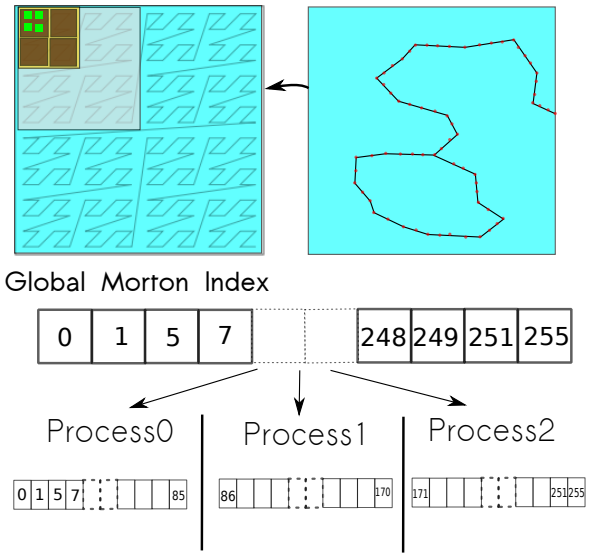


FIGURE 3.8 – Décomposition du domaine en suivant la *space filling curve* de Morton (hauteur d'arbre 4). Seule les feuilles (en vert) contenant des segments sont allouées puis des intervalles contiguës sont distribués entre les processus.

présenté sous forme décomposé dans la Figure 3.8, chaque feuille a un indice de Morton et la décomposition se fait alors en répartissant l'intervalle global des indices de Morton entre les processus. Le processus 0 reçoit le premier intervalle, puis le processus 1 le second, et ainsi de suite jusqu'au processus $N-1$ qui reçoit le dernier intervalle. Chaque sous domaine connaît la totalité de la distribution des intervalles entre les processus. La taille de l'intervalle reçue par chaque processus peut varier, comme nous le verrons avec l'algorithme de rééquilibrage. Il peut s'agir d'équilibrer le nombre de segments attribué par sous domaine ou le nombre maximum de sous domaines voisins.

Après avoir distribué les intervalles de Morton entre les sous domaines, on distribue les éléments du maillage dans chaque intervalle. Comme les segments sont interconnectés, il faut garder la cohérence du maillage aux interfaces des sous domaines pour pouvoir effectuer la totalité des étapes algorithmiques. Pour cela, nous introduisons donc la notion de donnée fantôme.

Soit S un segment d'extrémité $N1$ et $N2$, on définit $M(S) = \frac{1}{2}(N1 + N2)$ le milieu du segment S et l'indice de Morton, $IndexMorton(X)$, associé à la position du point X. L'affectation des données suit les règles suivantes :

Règle 1 Un segment S appartient à D_i si seulement si $IndexMorton(M(S)) \in I_i$;

Règle 2 Un nœud N appartient à D_i si seulement si $IndexMorton(N) \in I_i$;

Règle 3 Un segment S d'extrémité ($N1$, $N2$) est un segment fantôme de D_i si seulement si $IndexMorton(M(S)) \notin I_i$ et $IndexMorton(N1) \in I_i$ ou $IndexMorton(N2) \in I_i$;

Règle 4 N est un nœud fantôme de D_i si et seulement si $IndexMorton(N) \notin I_i$ et s'il est connecté à un segment local S de D_i. On a donc la règle

$$N \text{ nœud fantôme de } D_i \Leftrightarrow IndexMorton(N) \notin I_i \text{ et } IndexMorton(M(S)) \in I_i.$$

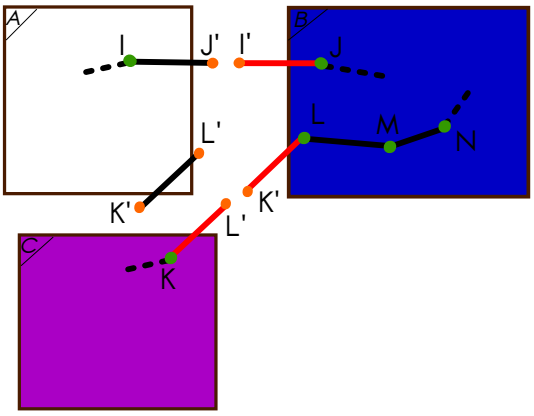


FIGURE 3.9 – Différentes configurations de nœuds et de segments. En orange les nœuds fantômes et en rouge les segments fantômes. Le segment $[IJ]$ est partagé entre deux sous domaines A et B. Le segment $[KL]$ est partagé entre trois sous domaines A, B et C.

Au final, en suivant le processus de distribution des intervalles et les règles d'affectation, nous avons plusieurs types de configurations :

1. le segment est entièrement contenu dans le domaine local et les nœuds à ses extrémités ne sont connectés qu'à des segments locaux. Le segment peut alors être manipulé localement sans implication pour les autres sous domaines qui n'ont pas besoin de connaître son existence (segment [MN] Figure 3.9).
2. le segment est partagé avec un de ses nœuds dans un sous domaine D_i et le second dans un sous domaine D_j. Nous ajoutons la notion de donnée fantôme pour pouvoir conserver l'interconnexion du maillage sur l'interface et effectuer l'ensemble des calculs sur chaque nœud. Sur la Figure 3.9 le segment [IJ] a un nœud dans un sous domaine A, le second nœud dans un sous domaine B. Le segment appartenant à A, il est fantôme dans le sous domaine B.
3. le segment A est entièrement contenu dans le domaine local mais un nœud ou les deux nœuds sont connectés à un segment partagé entre deux sous domaines. Les nœuds du segment A sont locaux par leur position mais partagés par une connexion avec un autre sous domaine. Ils doivent donc être synchronisés entre les sous domaines (segment [LM] de la Figure 3.9).

Gérer la cohérence du maillage entre les sous domaines doit être fait avec précision de par le dynamisme des données qui peuvent changer fréquemment de sous domaines ou

disparaître suite à une collision ou lors du remaillage. Pour cela un tableau ou buffer de correspondance est établi lors de la phase de migration (cf. Section 3.2.2.5). Ainsi, avec les dépendances entre les données et les règles d'attribution de ces données et la connaissance de la distribution globale des intervalles de Morton nous savons avec quels processus un sous domaine partage une interface. De cette façon les communications sont réduites au minimum et ne nécessitent que des échanges point à point avec les processus voisins.

Maintenant que nous avons présenté le partitionnent de l'espace en sous domaine et le principe d'attribution des données, nous allons voir comment les modifications s'opèrent au niveau des différentes étapes algorithmiques de la simulation en DD.

3.2.2 Adaptation des algorithmes de DD distribués

En comparaison avec l'approche séquentielle ou en mémoire partagée, pour réaliser une itération en mémoire distribuée nous devons ajouter des phases de communications entre les principales étapes de l'algorithme pour garder la cohérence des données entre les sous domaines (i.e. entre un nœud et son fantôme). Sur les étapes algorithmiques (Figure 3.10) apparaissent en rouge les nouvelles phases de communication pour synchroniser les données entre les sous domaines.

Avec notre décomposition de domaine, nous n'introduisons pas de zones de recouvrement, mais simplement la présence de données fantômes pour pouvoir connecter les sous domaines. Nous avons dans chaque sous domaine l'ensemble des segments locaux et l'ensemble des segments fantômes directement connectés à un nœud local. Nous allons détailler les modifications des différents algorithmes suite à la distribution des données et des dépendances entre les domaines.

3.2.2.1 Calcul de forces

Pour le calcul de force chaque sous domaine est en charge uniquement de ses segments locaux. C'est à dire qu'il calcule les contributions d'un segment local (pouvant comporter des nœuds fantômes) mais n'évalue pas les contributions d'un segment fantôme.

Pour les étapes de calcul liées à la contrainte appliquée et la force de cœur aucune communication n'est nécessaire puisque chaque sous domaine effectue le calcul sur ses segments locaux.

Avec un domaine distribué, les dépendances entre les sous domaines viennent du calcul du champ de contrainte élastique. Pour la gestion du calcul du champ proche (opérateur *P2P*) entre deux sous domaines l'algorithme de ScalFMM évalue dans chaque sous domaine les interactions avec les sous domaines voisins. On perd la réciprocité au niveau des interfaces mais de cette façon une seule phase de communication est nécessaire. Ainsi les communications sont groupées entre deux voisins en une seule et unique communication asynchrone recouverte par le calcul sur le domaine local. Il est donc important de réduire au minimum la taille des interfaces pour minimiser les calculs redondants sur ces feuilles qui englobent le domaine local.

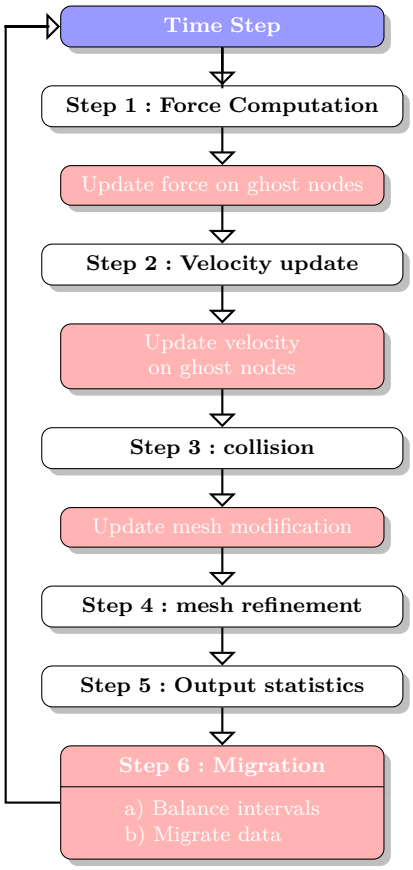


FIGURE 3.10 – Les étapes d'une itération en mémoire distribuée. En rouge les phases de synchronisation entre les sous domaines.

Une fois les étapes de calcul terminées, nous effectuons une nouvelle phase de communication avant d'affecter l'ensemble des contributions des segments sur les nœuds locaux (Étape 6, Figure 3.4). Comme présenté dans l'Algorithme 16 chaque sous domaine repère les segments locaux qui possèdent un nœud fantôme, puis pour chaque segment repéré, le processus construit un buffer contenant la contribution du segment local sur le nœud fantôme vers le ou les sous domaines cibles. Enfin chaque processus envoie et reçoit les contributions de chacun des sous domaines voisins avec qui il partage des segments. Le sous domaine peut alors effectuer la réduction des contributions sur les nœuds locaux de manière classique.

Maintenant, chaque sous domaine peut alors construire la force sur chacun de ses nœuds locaux et il possède aussi séparément la contribution liée à chaque segment. Cependant, si le calcul FMM a été effectué complètement il faut déterminer si la fréquence doit être modifiée. Une dernière communication (Allreduce) est nécessaire ; chaque processus communique à l'ensemble des autres processus la variation du champ lointain pour ajuster la fréquence. A la fin de cette étape les données du maillage sont cohérentes.

Algorithme 16 : Pseudo code d'échange des contributions sur les segments partagés.

```
   Données : Octree LocalTree
   // Pour chaque segment local
1  pour chaque Segment ∈ LocalMesh faire
   |  // Si le segment est partagé
2  |  si isShared(Segment) alors
3  |  |  Vector subDomainsTopackTo ← getsubDomains(Segment);
   |  |  // on prépare les réceptions
4  |  |  pour chaque Neighbor ∈ NeighborsSubDomains faire
5  |  |  |  ReceiveContributions(Neighbor);
   |  |  // on pack les contributions
6  |  |  pour chaque domain ∈ domainsTopackTo faire
7  |  |  |  PackContributions(domain,Segment);
   // Envoi et réception des contributions
8  pour chaque Neighbor ∈ NeighborsSubDomains faire
9  |  SendContributions(Neighbor);
10 |  UnpackContributions(Neighbor);
```

3.2.2.2 Mise à jour des positions

Après le calcul des forces, chaque sous domaine peut calculer la vitesse sur l'ensemble de ses nœuds locaux en appliquant l'algorithme indiqué dans la Section 1.2. Cet algorithme dépend uniquement des connexions directes qui sont connues dans chaque sous domaine avec des segments locaux ou des segments fantômes. En version distribuée, on applique le même algorithme que la version en mémoire partagée, et à la fin de l'algorithme on met à jour les vitesses sur les nœuds fantômes connectés à un segment local. On utilisera cette vitesse pour déplacer les segments fantômes. Comme on peut le voir sur la Figure 3.11, le sous domaine A qui possède le segment $[JK]$ ne connaît pas la vitesse du nœud K qui appartient au sous domaine B. Ainsi B calcule la vitesse de K puisqu'il connaît l'ensemble

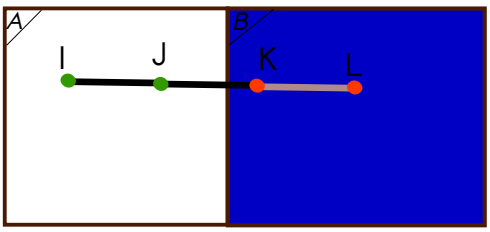


FIGURE 3.11 – Calcul des vitesses entre deux sous domaines. Chaque sous domaine calcule sur les nœuds locaux (Vert pour le sous domaine A et Orange pour le sous domaine B). On échange ensuite les vitesses pour les données fantômes. Ici B communique la vitesse du nœud K au sous domaine A.

de ses connexions ($[JK]$ est fantôme pour B) et communique cette vitesse au sous domaine A qui pourra alors déplacer le segment $[JK]$.

Cette mise à jour sur les nœuds fantômes connectés à un segment local est indispensable pour connaître la vitesse lors de la détection des collisions. Elle se fait aussi simplement que pour la synchronisation des forces avec une communication point à point entre les sous domaines voisins.

Une dernière étape est nécessaire pour déterminer la valeur du pas de temps qui est adaptative comme présenté dans la Section 1.2. Contrairement à la version en mémoire partagée, chaque sous domaine calcule son Δt optimal selon sa topologie. Ensuite, l'ensemble des processus effectue une réduction sur tous les sous domaines pour choisir le pas de temps le plus petit. Cet unique pas de temps Δt_{min} sera utilisé par tous les sous domaines notamment pour déplacer les nœuds et détecter les collisions.

3.2.2.3 Détection des collisions

Suite à cette étape de synchronisation pour déterminer la taille du pas de temps et déplacer tous les nœuds sur Δt tel que $X_i(t + \Delta t) = X_i(t) + V_i \times \Delta t$, nous procédons à la détection des collisions sur l'ensemble des processus.

La détection de collision nécessite d'accéder aux feuilles voisines avec le même schéma de communication que celui pour le calcul de force. Cependant, contrairement au calcul du champ élastique qui évalue une contribution sur les 4 nœuds pour chaque interaction, ici les données sont modifiées uniquement lorsqu'une collision est détectée. Par conséquent, plutôt que d'utiliser l'algorithme de ScalFMM qui duplique les calculs sur la frontière de deux sous domaines, nous choisissons qu'un seul processus effectue le calcul. Comme dans la version partagée, chaque feuille est maître du calcul uniquement avec les feuilles de son enveloppe recouvrante au *Nord-Est* (Figure 3.7). Ainsi, en terme de communications, seule les feuilles qui n'ont pas dans le domaine local la feuille maître du calcul, sont envoyées vers le sous domaine voisin qui calcule cette partie des interactions comme illustré sur la Figure 3.12.

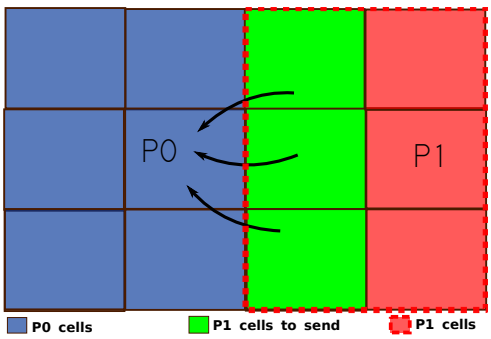


FIGURE 3.12 – Communication dans un sens des feuilles. P_1 communique les feuilles vertes pour que P_0 qui test au *Nord-Est* détecte les collisions avec ses feuilles.

Après cet envoi, dont le coût est recouvert par le calcul sur les feuilles à l'intérieur du sous domaine, chaque feuille peut finir de détecter l'ensemble des collisions sans dupliquer les calculs au niveau des interfaces et en réduisant le volume de communications. Par la suite, lorsqu'une collision impliquant deux sous domaines est détectée, le processus ayant procédé au calcul gère la modification du maillage sur les feuilles de l'interface.

Une fois ces modifications effectuées le processus les communiquent pour que le maillage reste cohérent à l'interface avec le processus qui n'a pas effectué le calcul sur cette interface. Ainsi chaque processus applique les modifications sur les interfaces où il n'est pas maître. La gestion de ces modifications correspond aux mêmes interactions que pour le remaillage que nous allons présenter maintenant.

3.2.2.4 Mise à jour du maillage

Le remaillage prend en considération les segments et leurs voisins. Il est par conséquent une source de conflits entre les sous domaines. Il faut qu'à la fin de cette étape les connexions entre sous domaines restent cohérentes en maintenant les correspondances à jour. Comme nous l'avons vu, cette étape peut produire trois effets sur les données. Un nœud et un segment peuvent être supprimés, un nœud et un segment peuvent être ajoutés, ou un nœud peut être déplacé. En mémoire distribuée, on définit alors la règle suivante :

Règle 5 un sous domaine ne peut pas modifier (supprimer/ajouter) une donnée fantôme.

La majorité des segments à cheval entre deux sous domaines ont un nœud local et un nœud fantôme. Le sous domaine qui possède le segment peut le modifier en prenant en compte le segment précédent et en supprimant/ajoutant un nœud local. Cela permet d'éviter de perdre les correspondances entre les sous domaines aux interfaces. Dans le cas où les deux nœuds d'un segment S (qui appartient au sous domaine D) sont des nœuds fantômes, le remaillage n'est pas effectué par le sous domaine D. Le déplacement des segments rend cette configuration suffisamment peu pérenne pour qu'au pas de temps suivant le segment quitte le sous domaine D pour retomber dans le cas classique présenté précédemment (une extrémité du segment appartient au même sous domaine que le segment S). De plus, les sous domaines qui possèdent les extrémités du segment S peuvent remailler le segment suivant et le précédent pour adapter le maillage.

Finalement, lors de l'étape 3 de l'Algorithme 6 de remaillage n'importe quel nœud local peut être déplacé pour équilibrer ses segments connectés. Par contre, si le nœud est connecté à un segment fantôme il faut alors communiquer sa nouvelle position à tous les sous domaines qui possèdent une copie de ce nœud. En effet, la migration des segments suppose que tous les nœuds sont à la même position pour prendre les mêmes décisions de migration des données dans chaque sous domaine.

Pour communiquer ces différentes opérations de modification du maillage, une unique communication point à point entre deux sous domaines est nécessaire. Chaque opération (ajouter/supprimer/déplacer) est référencée par un identifiant (`tagOperation` comme indiqué dans la Figure 3.13). Cet identifiant prend différentes valeurs pour spécifier s'il s'agit de la suppression d'un nœud (t_DelN), d'un segment (t_DelS), de l'insertion d'un nœud (t_NewN), du changement de connexion ($t\_ConnectN$) ou du déplacement d'un nœud

```
struct Remesh{
  int tagOperation;
  ccIndex iData;
  double x,y,z;
}
```

FIGURE 3.13 – Structure de données communiquées pour déplacer un nœud fantôme dans un sous domaine voisin. L'identifiant d'opération `tagOperation`, l'identifiant de la donnée, et les coordonnées de la nouvelle position.

(t_MoveN). Les modifications du maillage d'un sous domaine D_i qui ont une incidence sur le maillage du sous domaine D_j sont regroupées pour ne procéder qu'à une seule communication. Ainsi à la réception du buffer venant de D_i le sous domaine D_j selon le `tagOperation` sait le type de donnée à lire. À la réception du message, les opérations de remaillage sont lues de manière séquentielle par le sous domaine. Le message est déroulé en effectuant les opérations pour retomber sur un tableau de correspondance cohérent entre les sous domaines.

3.2.2.5 Migration

Suite aux déplacements des points et des segments, il faut les réaffecter dans les bonnes feuilles et donc ils peuvent avoir changé de domaine. L'algorithme de migration se déroule en deux étapes.

Étape 1 : migration des anciens segments locaux. Pour cela, on repère tous les segments locaux dont l'indice de Morton (indiqué par le barycentre) n'appartient plus à l'intervalle du sous domaine. Les données pour reconstruire le segment dans le sous domaine destinataire sont empaquetées à savoir la position des nœuds (extrémités du segments), le vecteur de Burgers, les plans de glissement, le vecteur de la contribution du champ lointain et la vitesse au pas de temps précédent. Le processus destinataire est connu car la distribution des indices de Morton est connue par tous les processus. Donc, la communication a lieu en une seule fois et de même pour la réception des données qui entrent dans le sous domaine.

Localement chaque sous domaine reconstruit l'information dont elle a besoin à partir de cette première phase de communication. Pour chaque message reçu, on lit séquentiellement les informations. Chaque nouveau segment est ajouté aux segments locaux tandis que les nœuds peuvent être des nœuds fantômes. Ils sont donc ajoutés soit aux nœuds locaux ou dans un des tableaux de correspondance avec les sous domaines voisins. Ce tableau de correspondance se créé automatiquement par la lecture séquentielle du message. Ainsi la position du nœud fantôme dans ce tableau donne son équivalent dans le sous domaine propriétaire du nœud réel. Enfin, les connexions avec les données locales sont trouvées par un parcours dans un tableau contenant les nœuds locaux incomplets c'est à dire possédant une connexion fantôme. A la fin de cette étape, toutes les données locales à un sous domaine sont complètes. Cependant les nœuds locaux ne possèdent pas toutes les connexions

auxquelles ils sont rattachés puisque certaines sont fantômes et n'appartiennent donc pas au même sous domaine.

Étape 2 : échange des données fantômes. Chaque sous domaine repère parmi ses segments locaux ceux qui possèdent un ou deux nœuds fantômes. De la même façon que précédemment, ces données sont empaquetées en un seul message destiné au sous domaine propriétaire du nœud fantôme. On procède alors aux envois et réceptions de ces buffers avec les sous domaines voisins. A la réception du message d'un sous domaine voisin D, on lit le buffer séquentiellement et on place ces données fantômes dans l'ordre dans le tableau de correspondance associé au sous domaine D.

3.2.3 Équilibrage de charge dynamique entre sous domaines

Que ce soit pour distribuer initialement le maillage entre les processus où pour équilibrer la charge suite à la migration des segments, il faut développer des algorithmes pour maintenir un bon équilibre de la charge tout au long de la simulation. De façon standard, à l'initialisation de la simulation chaque processus va lire l'ensemble des fichiers de configuration pour paramétrer la simulation. Ensuite, chacun des processus lit la taille de la boîte de simulation et la hauteur de l'arbre pour pouvoir construire la même décomposition de l'espace. Enfin reste à charger la configuration initiale de dislocations. Plusieurs cas sont possibles :

— Démarrage d'une nouvelle simulation. Dans ce cas, un processus maître distribue la configuration initiale puisque les dislocations sont générés dans l'espace de manière aléatoire. Par conséquent, ce processus maître génère cette configuration et distribue des intervalles d'indice de Morton à chaque processus pour que ceux-ci récupère un nombre égal de segments. Chaque sous domaine construit alors localement la sous partie de l'arbre qui correspond à son intervalle de Morton.

— Redémarrage d'une simulation à partir d'une sauvegarde d'une précédente simulation. Deux cas de figures sont possibles. Si la simulation redémarre avec le même nombre de processus, chacun lit le fichier de sauvegarde qui lui est destiné et construit localement la sous partie de l'arbre qui correspond à son intervalle de Morton. Si la simulation redémarre dans le but d'allouer plus de ressources de calcul en passant de N à $N+M$. Le processus maître redistribue sur les M nouveaux processus une partie de l'intervalle de Morton en incluant les nouveaux processus. Puis l'algorithme d'équilibrage dynamique pourra au fur et à mesure améliorer cette distribution.

A l'initialisation, la distribution des données s'effectue par le processus maître comme indiqué dans Algorithme 17. Dans cet algorithme, la charge est basée principalement sur le nombre de segments contenus dans chaque feuille. Nous essayons également d'ajuster le nombre de feuilles par processus qui doit rester proche d'un distribution moyenne. En conservant un nombre de feuille égale entre les processus nous évitons sans faire le calcul exacte de créer de fortes hétérogénéités dans le coût des communications. Dans le cas d'un redémarrage de la simulation le chargement se fait sans communication à partir de la lecture des fichiers de sauvegarde de chaque processus puisque les données sauvées

Algorithme 17 : Équilibrage de charge initial par le processus maître entre un nombre de processus ($nbProcess$).

Données : Octree Tree, int nbProcess

```
// Déterminer la charge moyenne par processus.
// Calcul le nombre moyen de segments par processus.
1  AvgLoad ← ComputeAvgLoad();
// Calcul le nombre moyen de feuilles par processus.
2  AvgLeaves ← ComputeAvgLeaves();
3  currentLoad ← 0;
4  currentNbLeaf ← 0;
// Détermine les indices minimum et maximum des intervalles par
   processus.
5  Morton minInterval[nbProcess];
6  Morton maxInterval[nbProcess];
7  minInterval[0] ← 0;
8  maxInterval[nbProcess − 1] ← MaximumIndex(Tree);
// Distribution des feuilles en nbProcess intervalles
9  pour chaque Feuille ∈ Tree faire
10     currentLoad ← currentLoad + Feuille.Cost();
11     currentNbLeaf ← currentNbLeaf + 1;
       // si la charge max est atteinte
12     si currentLoad > AvgLoad OR currentNbLeaf > 1.5 × AvgLeaves alors
           // on ferme l'intervalle
13         maxInterval[currentProc] ← Feuille.getMortonIndex() − 0x1LL;
14         minInterval[currentProc + 1] ← maxInterval[currentProc] + 1;
15         currentLoad ← 0;
16         currentNbLeaf ← 0;
```

lors d'un checkpoint permettent de retrouver l'état de la simulation. L'initialisation et le redémarrage sont des algorithmes rapides avec moins d'une minute pour générer et distribuer 1 000 000 de segments sur 256 processus.

Une fois le processus itératif démarré, l'équilibrage doit être maintenu malgré le déplacement des segments. Pour cela, nous procédons de manière régulière à un rééquilibrage de la distribution des intervalles d'indices de Morton. L'équilibrage se fait de façon aussi locale que possible donc sans communications et synchronisations globales. L'idée est de faire glisser l'intervalle de Morton à gauche ou à droite pour l'agrandir ou le réduire afin d'ajuster la charge. Par conséquent avec un tel algorithme la contrainte principale est de n'échanger des feuilles qu'avec les processus directement à gauche ou directement à droite sur la distribution des intervalles de Morton. La difficulté est de décider si le processus récupère une partie de l'intervalle des voisins, cède une partie de son intervalle ou les deux à la fois pour décaler les intervalles vers la gauche ou vers la droite. Pour prendre cette décision, l'algorithme exécuté en parallèle sur chaque sous domaine prend en compte uniquement la charge des premiers et des seconds voisins dans l'intervalle comme illustré

sur la Figure 3.14. La prise en compte de la charge des seconds voisins permet d'affiner

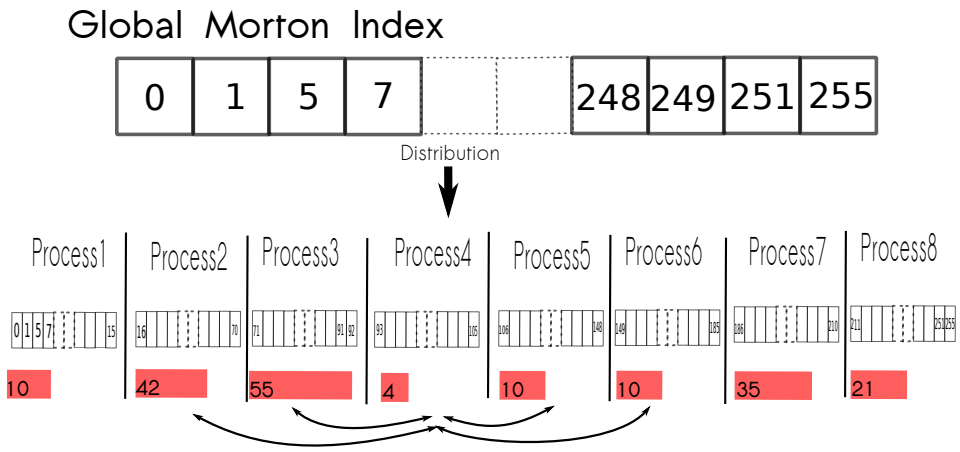


FIGURE 3.14 – Équilibrage de charge dynamique. Le schéma communication est illustrée pour le processus 4 qui envoie et reçoit des informations sur la charge (indiquée en rouge), quantifiée de 1 à 100, à ses deux voisins de gauche(2,3) et ses deux voisins de droite(5,6).

la décision pour équilibrer la distribution des feuilles avec le voisin de gauche ou celui de droite.

La première étape pour un processus P_i consiste à calculer la charge sur son intervalle. Pour cela, P_i compte le nombre de feuilles et le nombre de segments dans son intervalle. Ensuite cette information est communiquée avec les premiers et seconds voisins. Pour P_i, le message est envoyé aux processus P_{i-2}, P_{i-1}, P_{i+1} et P_{i+2} et le processus P_i reçoit en retour la charge de ces quatre sous domaines. À partir de ces informations chaque processus communique à ces voisins ses besoins à savoir se décharger d'une partie de son intervalle ou alors récupérer la charge de ces voisins. Il ne s'agit que de la transmission d'une demande, la décision finale est prise par chaque processus d'accepter ou refuser cette demande.

Un processus P_i a quatre possibilités pour déclarer ces besoins à gauche et à droite :
- P_i n'a pas besoin de feuilles ;
- P_i souhaite récupérer des feuilles uniquement à gauche (P_{i-1}) ;
- P_i souhaite récupérer des feuilles uniquement à droite (P_{i+1}) ;
- P_i souhaite récupérer des feuilles à gauche (P_{i-1}) et à droite (P_{i+1}).

Ces souhaits sont transmis par un message du sous domaine P_i au voisin de gauche P_{i-1} et au voisin de droite P_{i+1}. En plus, les besoins sont pondérés (besoin fort, besoin moyen ou besoin faible) pour affiner la décision finale en cédant une part plus ou moins grande de son intervalle.

La dernière étape consiste à transmettre une partie de son intervalle à gauche et à droite. Pour prendre cette décision finale, chaque processus prend en compte trois informations. Ses propres besoins, les besoins à gauche et les besoins à droite. Puisque chaque processus connaît les besoins des voisins, en appliquant le même algorithme nous assurons la cohérence des décisions pour envoyer ou recevoir une partie de l'intervalle. Pour savoir combien de feuilles de son intervalle le processus doit céder à ses voisins, il déduit de sa

charge actuelle la perte à gauche et la perte à droite. La pondération (besoin fort, besoin moyen ou besoin faible) permet de favoriser un côté de l'intervalle plutôt que l'autre. Enfin une fois le message transmis le processus peut recevoir une partie de l'intervalle des voisins directs. Bien évidemment, si une partie de l'intervalle a été cédée à gauche par le sous domaine D_i, à gauche D_{i-1} n'envoie rien à D_i et de même à droite. Ensuite la nouvelle distribution des intervalles de Morton est diffusée à l'ensemble des processus pour que chaque processus connaisse les nouveaux intervalles de Morton de chacun des sous domaines. Ainsi chacun peut mettre à jour la liste des sous domaines avec lesquels il partage une interface pour cibler les communications pour les autres étapes de l'algorithme.

Nous pouvons prendre l'exemple de la Figure 3.14 pour comprendre la fonctionnement de l'algorithme en considérant le processus 4. Sur l'exemple de la Figure 3.14, le processus 4 constate que la charge à gauche (processus 2 et 3) est très supérieure à la charge à droite (processus 5 et 6). Après la prise d'information de la charge des processus 2, 3, 5 et 6, le processus 4 qui à une charge très faible va informer les processus voisins (processus 3 et 5) de ses besoins. Il constate que la charge des processus à droite (5 et 6) est équilibrée, et que le processus 2 n'a pas de besoins. Ainsi, le message communiqué par le processus 4 à gauche (processus 3) et à droite (processus 5) sera un besoin *fort uniquement à gauche*. Enfin le processus 3 en recevant le message du processus 4 va prendre en compte le fait que le processus 2 n'a pas de besoin pour transmettre le maximum de son intervalle au processus 4.

Algorithme 18 : Déterminer les numéro des processus voisins.

```
   Données : Octree Tree,int myRank, int nbProcess, Morton
             minInterval[nbProcess], Morton maxInterval[nbProcess]
1  int neighborsId[];
   // pour chaque feuille locale
2  pour chaque Feuille ∈ Tree faire
3      LeavesNeighbours[27] ← GetNeighbors(Feuille);
       // pour chaque feuille voisine
4      pour chaque neighbourLeaf ∈ Neighbours faire
5          MortonneigLeafIndex ← MortonIndex(neighbourLeaf);
           // on récupère le numéro du processus qui possède cet indice
6          int processOwner ←
           getOwner(minInterval, maxInterval, neigLeafIndex);
           // si cette feuille n'est pas locale
7          si processOwner! = myRank alors
8              si processOwner ∉ neighborsId alors
                   // ajout à la liste des voisins
9                  Push(processOwner,neighborsId)
```

La dernière étape de l'algorithme de rééquilibrage met à jour la distribution des intervalles de Morton via une communication globale. Une fois cette distribution des intervalles connue par tous, chacun peut déterminer le numéro de ceux avec qui il partage une inter-

face en se basant simplement sur les indices de Morton (Algorithme 18). En identifiant précisément les processus partageant une interface, le volume de communication est réduit pour les algorithmes de mise à jour pour les données fantômes et on peut surtout effectuer la migration des données. En sortie de l'algorithme seul les intervalles des indices de Morton sont modifiés, par conséquent la dernière étape consiste à appliquer l'algorithme de migration décrit précédemment afin de distribuer le maillage de dislocations selon les nouveaux intervalles.

Cet algorithme d'équilibrage entraîne peu de communications avec seulement des échanges point à point entre les processus voisins sur l'intervalle et une communication globale pour connaître la distribution générale en fin d'algorithme. Il a donc un faible coût mais aussi le désavantage de n'équilibrer que localement, il faut donc qu'il soit exécuté régulièrement pour ne pas créer de déséquilibre sur la globalité des processus.

Bilan

Nous avons décrit l'ensemble des techniques et algorithmes mis en place pour exploiter un parallélisme hybride sur toutes les étapes de la simulation en DD. En mémoire partagée, nous avons cherché à optimiser les calculs en améliorant l'intensité arithmétique notamment en réduisant le nombre de couleurs pour les dépendances entre feuilles voisines. De là, nous avons pu proposer une amélioration de l'ordonnancement statique du travail des threads sur chaque couleur. Au niveau du maillage et de la structure de données, nous présentons un parallélisme à base de tâches OpenMP. Les algorithmes statiques c'est-à-dire sans insertion de données sont optimisés pour adapter la granularité des tâches de manière dynamique. Pour les opérations dynamiques sur la structure de données (insertion, suppression) notre implémentation garantie que les algorithmes de remaillage et de séparation soient sûrs en parallèle.

En mémoire distribuée, nous introduisons la distribution basée sur les intervalles de Morton. Nous analysons les différentes modifications apportées aux algorithmes pour fonctionner en se basant sur la décomposition du domaine selon ces intervalles de Morton. Nous présentons l'introduction de données fantômes pour réduire les zones de recouvrement. La perte de la réciprocité des calculs du champ élastique proche est justifiée pour éviter des communications coûteuses, tandis que pour la détection des collisions, nous conservons cette réciprocité en ajoutant une phase de communication à la fin de l'algorithme pour mettre à jour le maillage.

Enfin, nous présentons à travers notre algorithme d'équilibrage les efforts faits pour gérer la distribution des sous domaines et l'équilibrage de charge entre les processus malgré le déplacement des segments. Cet algorithme à faible coût prend en compte l'évolution permanente de la distribution pour pouvoir être exécuté de façon régulière au cours de la simulation.

Chapitre 4

Validation sur des grands challenges

Sommaire

Dans cette dernière partie, nous validons les évolutions apportées au code dont l'objectif est de permettre le passage à l'échelle pour les grandes simulations. Nous avons présenté dans les chapitres précédents, différentes contributions pour passer du code séquentiel original Numodis, à la version parallèle OptiDis.

Dans le chapitre 1, nous présentons les évolutions algorithmiques. L'introduction de la bibliothèque ScalFMM, l'algorithme adaptatif pour le calcul du champ de force élastique, l'introduction d'un noyau de détection de collisions sont autant de développements pour améliorer les performances. Dans une seconde partie 2, une structure de données adaptée au dynamisme des simulations de DD, ainsi que l'ensemble des techniques algorithmiques pour le maintien de l'organisation et de l'efficacité des accès en mémoire ont été introduits. Enfin dans le Chapitre 3, un parallélisme hybride OpenMP/MPI est présenté. En plus des améliorations du parallélisme natif de la bibliothèque ScalFMM, un parallélisme par tâches sur la nouvelle structure de données est apporté ainsi que l'ensemble des modifications liées à la décomposition de domaines avec une technique de recouvrement minimum.

Nous allons illustrer ces trois principaux axes d'évolution à la simulation sur une architecture de calcul moderne pour mettre en évidence les impacts sur le calcul.

4.1 Présentation des architectures et des tests

4.1.1 Description du cluster de calcul

Les calculs ont été effectués sur le cluster Mistral de la plate-forme de calcul Plafrim [64]. Les nœuds possèdent $128Go$ de RAM chacun et sont interconnectés par un réseau Infiniband QDR à 40Gb/s. Chaque nœud est composé de 2 sockets à 10 cœurs de type Ivy-Bridge Intel® Xeon® E5-2670 v2, ce qui donne 20 cœurs cadencés à $2.55GHz$ par nœud du cluster.

Pour chaque cœur il existe deux niveaux de cache privé L1 et L2 de taille respective $32Ko$ et $256Ko$, et un troisième niveau de cache L3 de taille $25Mo$ partagé entre les 10 cœurs d'une socket. La Figure 4.1 présente cette architecture.

Au total nous avons 8 nœuds disponibles pour 160 (8×20) unités de calcul.

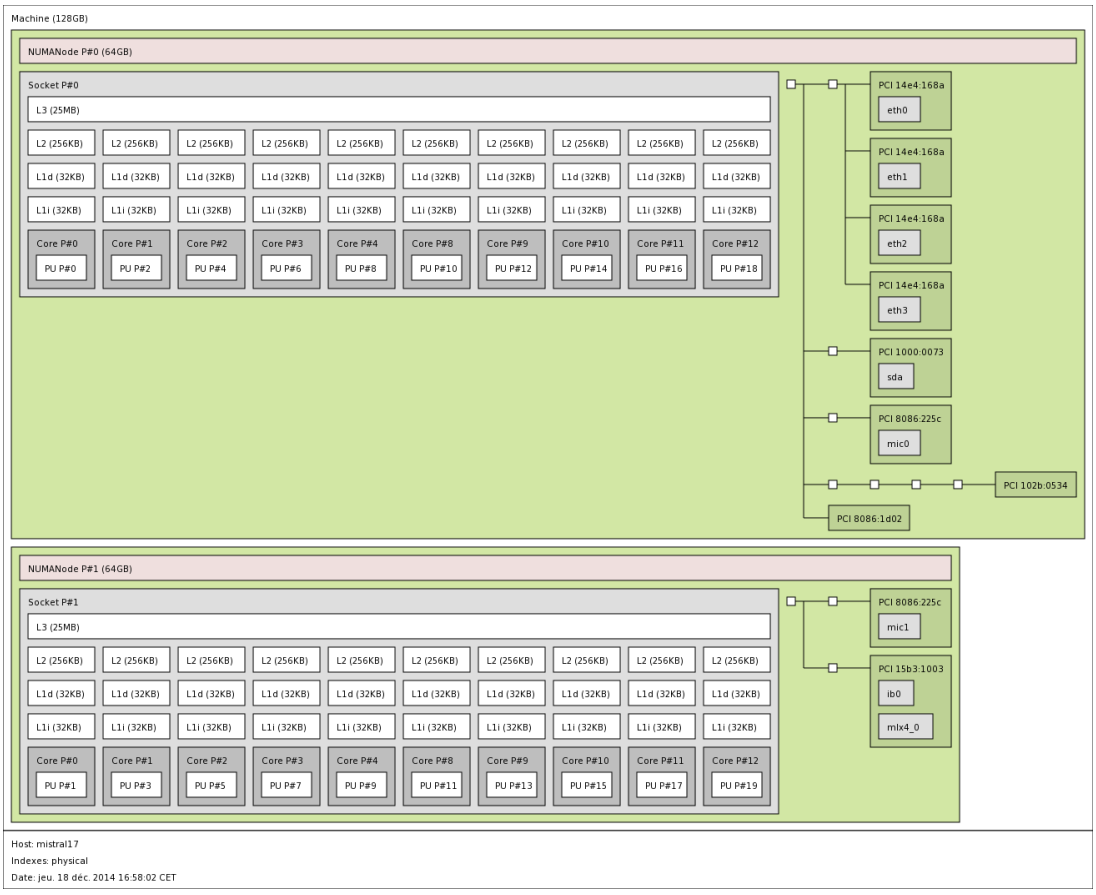


FIGURE 4.1 – Représentation avec Hwloc (Portable Hardware Locality) de l'architecture d'un nœud Mistral sur la plate-forme Plafrim.

4.1.2 Indicateurs de performance

L'accélération permet d'évaluer la capacité du code à utiliser l'ensemble des ressources de calcul. On mesure l'accélération entre un calcul séquentiel sur un cœur $T_{séq}$ et un calcul

utilisant plusieurs cœurs T_{para}. Plus le résultat est proche du nombre de cœurs utilisés meilleure est l'implémentation. L'accélération est donné par :

$$Acceleration = \frac{T_{séq}}{T_{para}}$$

On obtient aussi l'efficacité parallèle Eff, qui est calculée en considérant le temps séquentiel $T_{séq}$, le temps en parallèle T_{para} ainsi que le nombre d'unités de calcul $NbCores$ avec la formule suivante :

$$Eff = \frac{Acceleration}{NbCores}$$

L'efficacité de 100% signifie que nous utilisons pleinement l'ensemble des ressources de calcul.

4.1.3 Description des cas tests et validation physique

Deux classes de simulations sont présentées ici, en fonction du type d'objets modélisés. Les simulations comportent des lignes de dislocations et/ou de petites boucles d'irradiation. Nos tests démontrent l'apport des contributions pour l'efficacité du code mais la validité physique des résultats reste bien sûr primordiale. Toutefois, il n'existe pas un indicateur ou un critère particulier de convergence qui validerait où invaliderait les résultats d'une simulation. Une des méthodes utilisées revient à comparer des résultats entre les simulations en dynamique moléculaire et les simulations en DD. On valide ainsi des réactions topologiques complexes comme la jonction entre un défaut d'irradiation et une source. Le code Numodis ayant déjà été validé avec ce type de comparaisons, nous savons que la physique utilisée est correcte [96]. Cependant à grande échelle des milliers de réactions se produisent à chaque pas de temps. Ainsi, l'effet d'une large population de dislocations pourrait modifier le comportement de ces réactions sans qu'une comparaison ne soit possible. En parallélisant le code les traitements ne sont pas identiques, ni effectués dans le même ordre qu'en séquentiel, entraînant une variation notable de l'énergie du système. Enfin, l'observation au microscope électronique informe des patterns qui se forment dans les cristaux ou des densités de dislocations suite à un certain pourcentage de déformation. Cependant, à l'heure actuelle, à grande échelle la présentation précise du protocole de simulation reste essentielle pour obtenir la validation de résultats par la communauté.

Cas test 1

Dans ce premier cas, nous construisons une simulation visant à modéliser le comportement mécanique du matériau non irradié. Elle doit permettre, à terme, de reproduire et comprendre la courbe de traction représentée par la courbe noire à la Figure2. Dans cette simulation, des lignes de dislocations infinies (sans ancrage des nœuds) sont générées aléatoirement sur les plans d'un cristal de type HCP. Le grain a une forme cubique avec des conditions aux bords périodiques. Pour ce premier cas, le grain cubique à une taille

de $10\mu m$ de côté et est paramétré comme un grain composé de zirconium. L'état initial de la simulation est illustré par la Figure 4.2. Enfin pour la contrainte appliquée, il s'agit d'une contrainte imposée de $500MPa$.

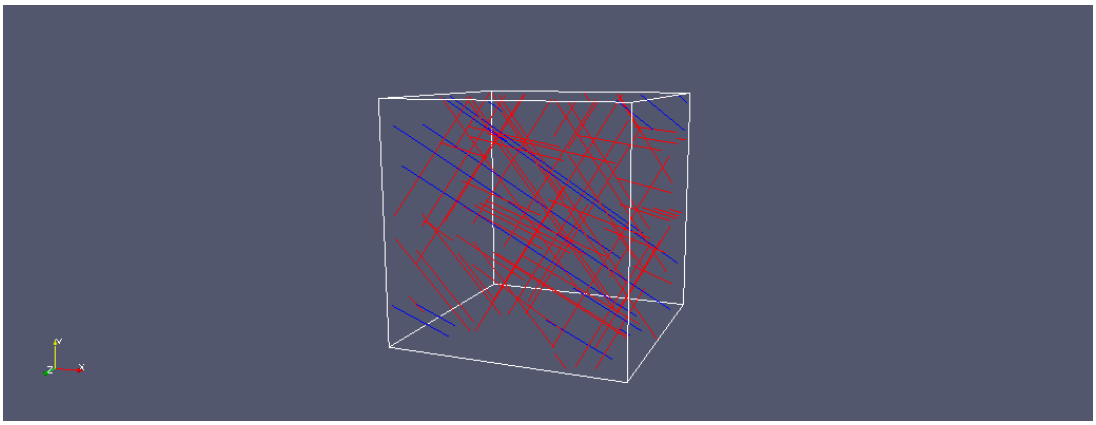


FIGURE 4.2 – Conditions initiales d'une simulation comportant des dislocations rectilignes infinies avec conditions périodiques dans un grain cubique de type HCP.

Cas test 2

La seconde configuration se rapproche du challenge industriel initial en simulant le comportement mécanique d'un matériau irradié. L'enjeu est à la fois de comprendre le durcissement du matériau mesuré à la Figure 2, mais également le phénomène de canalisation de la déformation. Pour cela, une forte densité de petites boucles induites par l'irradiation des matériaux est introduite. Nous utilisons un grain de type HCP paramétré comme un alliage de zirconium. Il se présente sous la forme d'un parallélépipède de $10\mu m$ de côté et $5\mu m$ de hauteur. Les conditions aux bords sont périodiques pour permettre l'expansion et la multiplication des sources de Frank-Read (FR). Nous introduisons les boucles de manière aléatoire dans le grain pour atteindre une densité de l'ordre de 10^{22} dislocations par m^3. Pour rendre le cas dynamique, nous introduisons aussi des sources de Frank-Read (FR) artificiellement ancrées aux extrémités. Les conditions initiales de cette configuration sont illustrées par la Figure 4.3. La contrainte appliquée est une contrainte imposée de $180MPa$.

4.2 Étude générale de la simulation

4.2.1 Profilage du code

Dans un premier temps nous profilons le code dans la configuration du cas test 1 sur un pas de temps. Le profilage est effectué en séquentiel dans un cas usuel d'utilisation. Les mesures sont effectuées pour différentes densités de segments allant de 22 000 à 900 000. Nous faisons varier la densité de dislocations en générant plus de lignes aléatoirement de dislocations dans le domaine.

La Figure 4.4 donne pour un octree de hauteur $H = 6$, le pourcentage du temps sur une itération, nécessaire pour calculer le champ de force complet (FMM) (bleu), détecter et traiter les collisions (jaune), calculer les vitesses (orange), modifier la topologie (vert) et procéder aux sorties pour les résultats (marron).

De manière générale pour les différentes densités de segments nous constatons que le calcul de force, malgré l'utilisation de la FMM (complexité linéaire), représente la grande majorité de la consommation des ressources CPU et que le calcul de vitesse, le remaillage et les sorties représentent moins de 5%. Pour des simulations comportant jusqu'à 200 000 segments le calcul du champ de force représente entre 85% et 90% du temps de l'itération. Ce calcul des forces est complet et maintient un équilibre entre le coût de calcul du champ proche et celui du champ lointain. Cependant, la Figure 4.4 montre aussi, en observant le coût de détection des collisions, que plus la densité des segments augmente plus l'opérateur *P2P* (interactions directes) est coûteux. Aussi utilisé pour noyau pour la détection des collisions, cet opérateur prend une part de plus en plus considérable des ressources. Le calcul des collisions représente seulement 10% pour 100 000 segments contre 35% pour 900 000 segments. Comme la hauteur de l'arbre est fixe, cette variation dans la consommation des ressources correspond à l'augmentation de la charge dans les feuilles de l'arbre et donc à l'augmentation des interactions entre les segments voisins. Nous avons mesuré que pour 45 000 segments, nous avons en moyenne 132 interactions directes entre segments pour chaque feuilles avec 13 000 feuilles alors qu'avec 900 000 segments il y a plus de 4 100 interactions directes et 32 000 feuilles. Par contre, comme on peut le voir sur la Figure 4.5 si on augmente la hauteur de l'arbre en ajoutant un niveau, on peut alors allouer plus de feuilles et avoir même pour les fortes densités un nombre d'interactions entre segments voisins qui reste faible. Ainsi, on maintient le coût de l'opérateur *P2P* stable relativement aux autres opérateurs comme le montre la Figure 4.6.

Par conséquent pour remédier à cette augmentation du coût calculatoire du champ proche lors des simulations à grande échelle, l'utilisation d'un octree à taille variable

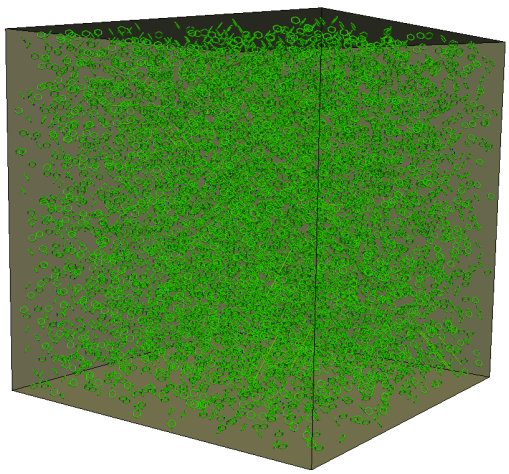


FIGURE 4.3 – Conditions initiales avec des défauts d'irradiation et des sources de FR aux extrémités ancrées dans un grain avec conditions périodiques.

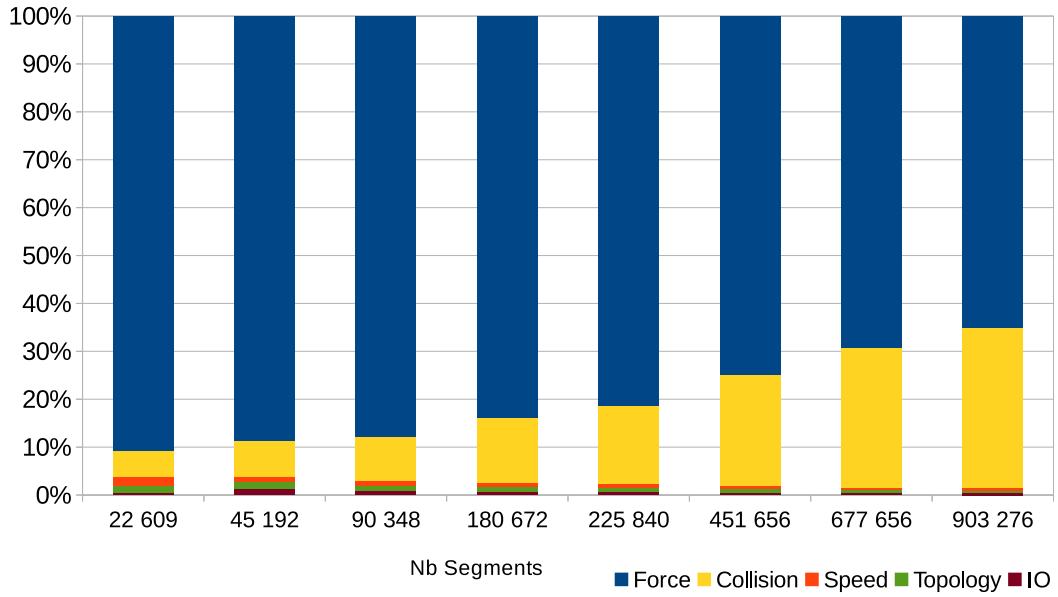


FIGURE 4.4 – Occupation des ressources CPU pour différentes densités de dislocations avec une hauteur d'arbre $H = 6$ sur un pas de temps.

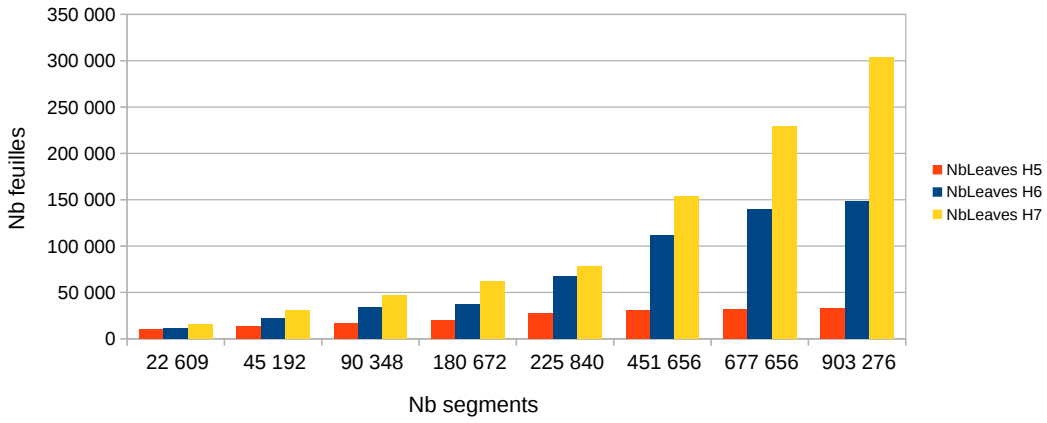


FIGURE 4.5 – Comparaison du nombre de feuilles allouées selon la hauteur de l'arbre.

peut être nécessaire avec une hauteur supérieure pour la détection des collisions. Pour les grandes simulations avec de fortes densités nous utilisons une hauteur $H = 6$ pour calculer le champ de force et une hauteur supérieure $H = 7$ pour la détection des collisions. Avec $H = 6$, pour le calcul du champ élastique on déséquilibre le coût de calcul du champ proche par rapport à celui du champ lointain tout en conservant une forte contribution de la force pour le champ proche. Ainsi, notre algorithme adaptatif en tire avantage en effectuant moins souvent le calcul FMM complet. Pour la détection des collisions avec une hauteur de 7, le coût est réduit sans pour autant contraindre le déplacement des segments. En effet, d'après l'étude menée dans le Chapitre 1.1.2.1 pour les plus grandes simulations

(volume $> 12\mu m^3$) et notamment avec la discrétisation des défauts d'irradiation une hauteur d'octree de 7 respecte tous les critères de stabilité des schémas. A l'heure actuel le choix de la hauteur de l'arbre en fonction de la densité de segment initiale est à la charge de l'utilisateur. Une évaluation automatique du code sera à intégrer pour déterminer la hauteur fonction de la densité courante de segments par cellules et donc du nombre d'interactions directes mais aussi des paramètres de la simulation comme la durée du pas de temps, la contrainte appliquée ainsi que les critères de discrétisation. De cette façon,

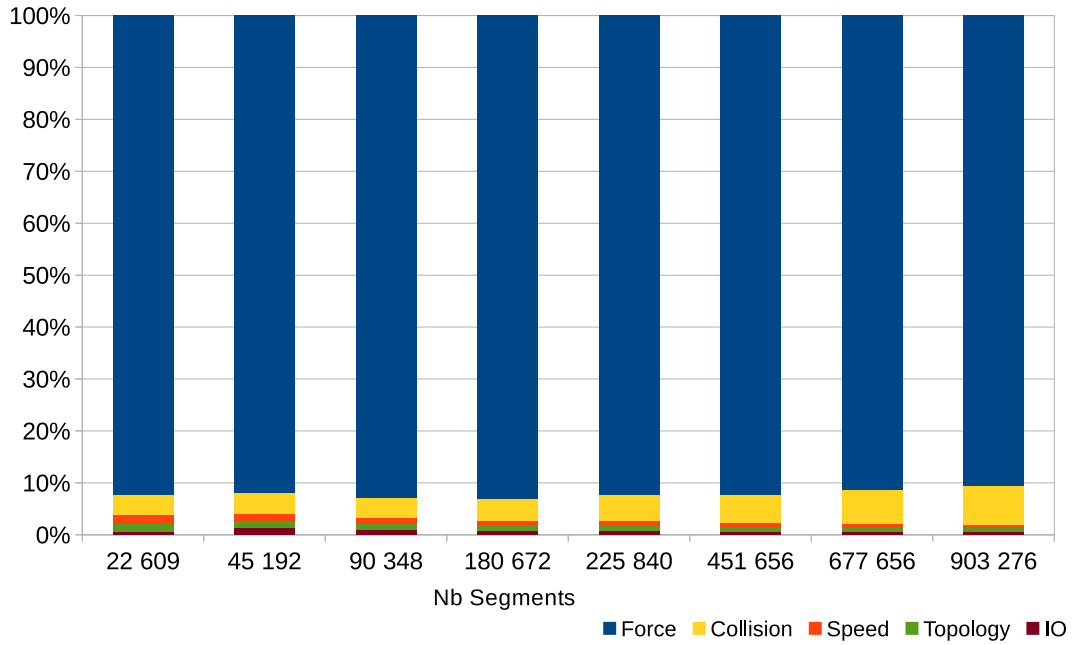


FIGURE 4.6 – Occupation des ressources CPU pour différentes tailles de simulation avec une hauteur d'arbre $H = 6$ pour le calcul FMM et $H = 7$ pour la détection des collisions.

même avec une forte densité (900 000 segments) le rapport de temps entre le calcul des collisions et le calcul du champ élastique reste stable avec respectivement environ 10% et 85% de l'utilisation des ressources dans toutes les situations.

La Figure 4.7 présente l'utilisation des ressources de calcul (temps CPU) sur un pas de temps où nous calculons seulement le champ proche ($P2P$) du champ élastique. Introduit dans le Chapitre 1.1.1, l'algorithme adaptatif du calcul des forces, permet de réutiliser le même champ lointain sur plusieurs pas de temps. Ainsi cette figure correspond à un pas de temps réutilisant le même champ lointain. Dans ce cadre là, nous avions évoqué le besoin de ne pas réduire la portée du champ proche en augmentant la hauteur de l'octree pour ne pas diminuer la part du champ proche sur la totalité du champ élastique. Par conséquent, comme on peut le voir sur la Figure 4.7, avec l'augmentation du nombre de segments le coût du calcul du champ élastique pour une hauteur $H = 6$ avec seulement l'opérateur $P2P$ est de plus en plus prohibitif alors que le temps passé à détecter les collisions reste stable avec une hauteur supérieure de l'octree $H = 7$. En l'état, avec ces travaux de thèse ce coût supplémentaire doit être consenti pour en retour bénéficier du gain offert par l'algorithme adaptatif. En effet, il est important pour notre algorithme de

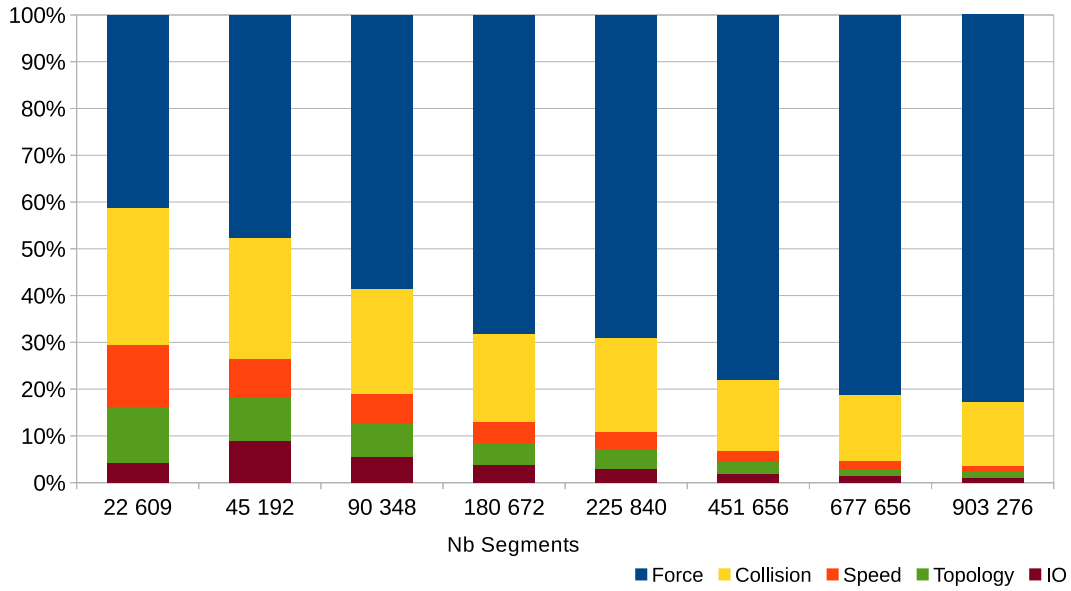


FIGURE 4.7 – Occupation des ressources de calcul pour différentes tailles de simulation avec une hauteur d'arbre $H = 6$ pour le calcul de force uniquement avec le champ proche ($P2P$) et $H = 7$ pour la détection des collisions.

calcul de force adaptatif de maintenir une bonne précision des calculs du champ élastique en conservant une forte contribution provenant des interactions proches. On peut alors notamment dans les configurations de dislocations homogènes et denses, mettre à jour moins fréquemment le champ lointain et économiser beaucoup de temps de calcul.

4.2.2 Comportement de la structure de données

Au niveau de la structure de données chaînée par blocs, les contributions sont multiples et se complètent, il est donc difficile d'étudier séparément les différents impacts de nos apports. D'un côté, nous avons l'organisation propre de la structure avec des blocs de taille fixe chaînés entre eux et groupés par *pool* lors de l'allocation. De l'autre coté, nous bénéficions de l'organisation en mémoire cohérente basée sur les accès par indirections aux données et des insertions selon la localité spatiale dans la boîte de simulation. A cela s'ajoute l'apport du tri qui renforce cette cohérence entre l'organisation des références dans l'octree et leur placement dans la structure chaînée.

Le cas présenté dans cette étude (Figure 4.8) est extrêmement dynamique. Nous appliquons des conditions aux bords périodiques avec des lignes infinies (sans nœuds ancrés) très mobiles de par la contrainte appliquée élevée de $550MPa$. Dans la structure de données, la taille du bloc B est très importante et la Figure 4.9 met en évidence l'apport de l'organisation en blocs. Elle montre l'influence de la taille B sur le temps de calcul du champ proche avec les accès par indirection aux données. Dans cette figure, avec des blocs de taille $B = 1$ nous sommes sur une organisation en liste chaînée classique tandis qu'avec une taille suffisamment grande pour contenir tous les éléments $B = 200\,000$ nous sommes avec un tableau classique. On constate que sur ce calcul du champ proche avec

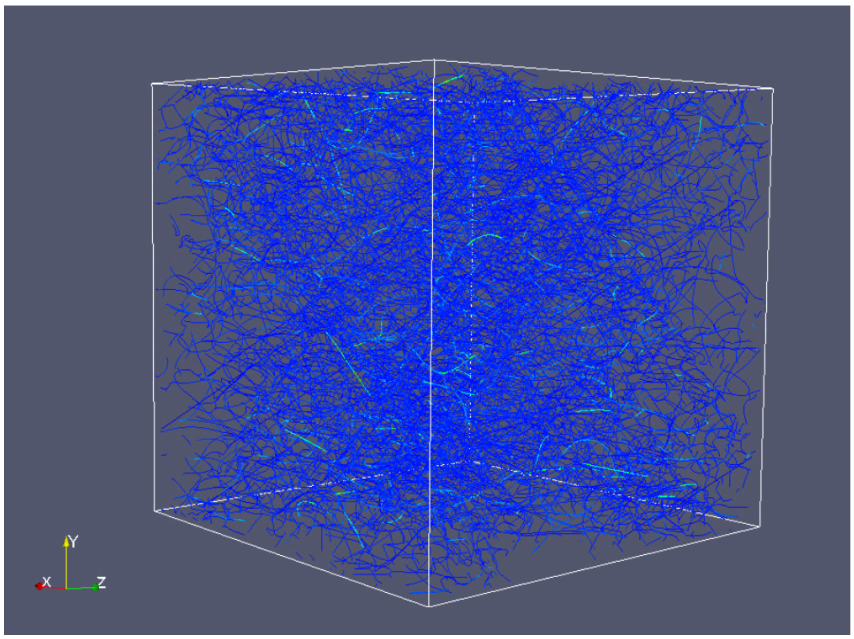


FIGURE 4.8 – Déformation de 2% et formation d'un réseau de dislocations complexe et dense. La simulation contenant plus de 100 000 segments est effectuée avec des conditions périodiques avec un pilotage par contrainte imposée $500MPa$.

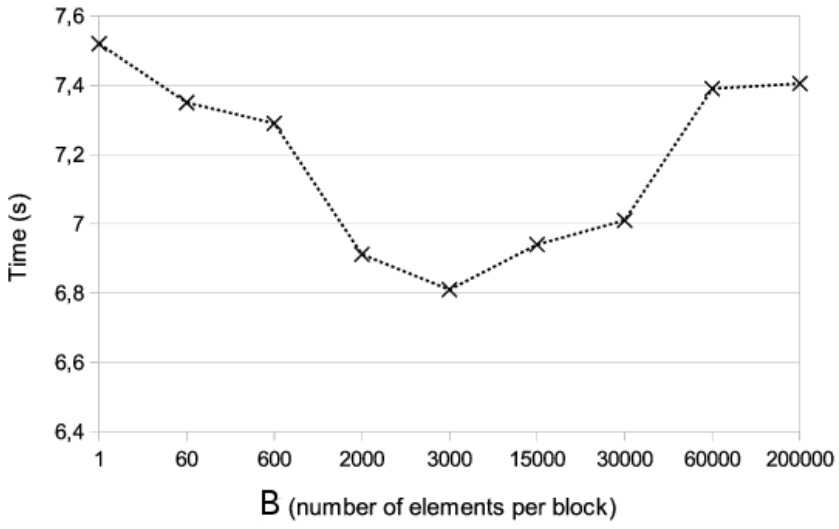


FIGURE 4.9 – Impact du paramètre B (taille du vecteur) sur le calcul des interactions entre segments (*P2P*).

les indirections de l'octree vers les structures chaînées, le temps moyen sur 500 itérations

est le plus rapide pour des blocs de taille 2 000 éléments. Comme attendu pour $B = 1$ et $B = 200\,000$ les performances sont les moins bonnes. La liste ($B = 1$) implique en permanence des accès aléatoires en mémoire du fait de l'allocation dynamique pour chaque donnée. Pour le tableau ($B = 200\,000$) il s'agit aussi d'un problème d'accès aléatoire dû à l'insertion des données qui est faite en fin de tableau et donc sans maîtrise sur la localité des données.

Dans la simulation la taille d'un bloc est donnée par $B \times DataSize$, avec $DataSize$ la taille du type de donnée. Soit par exemple pour le type *Segment*, 144 octets. Sur le calcul de force seuls les segments sont accédés par indirection pour récupérer le vecteur de Burgers, ainsi on comprend qu'avec des blocs de taille contenant en moyenne 2 000 données nous remplissons bien le niveau de cache L2 (256 Kbytes sur notre configuration) d'où de meilleurs résultats sur ce cas.

Cependant, avec les accès jusqu'aux nœuds dans l'algorithme de détection des collisions les niveaux de cache sont plus remplis et la taille optimale des blocs est alors plus faible. De plus, l'algorithme de tri n'est pas appliqué ici pour ne pas fausser la comparaison mais améliore la localité. Ainsi grâce au tri, en réduisant les accès aléatoires, on peut réduire la taille des blocs pour pouvoir mieux paramétrer la granularité pour les algorithmes parallèles comme nous le voyons avec la Figure 4.13. Dans la pratique la taille des blocs doit rester entre 1 000 et 2 000 éléments sur les machines Mistral. Ainsi pour la suite du cas présenté ici, nous utilisons des blocs de taille 1 100 éléments. Dans la Figure 4.10 nous

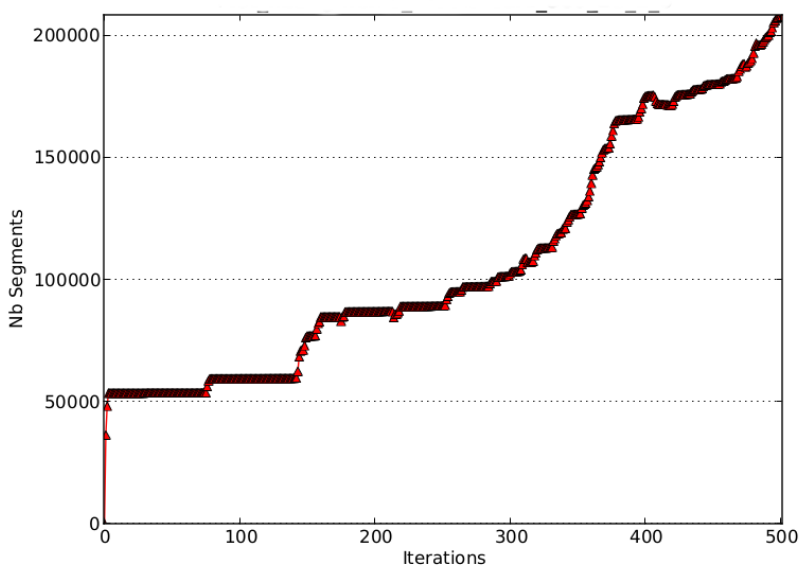


FIGURE 4.10 – Croissance de nombre de segments dans une simulation contenant des sources de FR.

montrons pour un cas très dynamique avec une forte contrainte appliquée l'évolution du nombre de segments. Nous observons que sur 500 itérations le nombre de segments passe de 50 000 à 200 000. Cela se traduit par beaucoup de déplacement des données et de très nombreuses insertions et suppressions dans la structure de données qui tendent à pénaliser

la simulation par le désordre occasionné.

La consommation mémoire totale de la simulation est présentée dans le Tableau 4.1. Celle-ci n'est pas un facteur limitant quelle que soit la station de travail utilisée en restant inférieure à $20Go$ même pour les plus grosses simulations. Ainsi cela confirme que l'utilisation d'un octree plus haut et la réplication de données dans les feuilles n'est pas une contrainte pour le type de simulations visées.

Nb segments	**Volume**(Mo)(H=7)	**Volume**(Mo)(H=6)
1 968	176.54	129.22
5 916	352.60	231.71
9 852	517.17	322.48
19 704	921.41	525.54
59 124	2 268.98	1 148.68
98 532	3 479.77	1 609.27
197 064	6 128.40	2 823.62
591 192	13 838.42	4 063.80
985 320	18 956.94	4 855.89

TABLE 4.1 – Consommation mémoire maximale de la simulation en fonction du nombre de segments pour une hauteur d'arbre H=6 et H=7.

Au niveau de cette structure de données par blocs, on paramètre la gestion du volume d'allocation afin de toujours laisser suffisamment de place pour permettre l'insertion de nouvelles données. La Figure 4.11 donne le pourcentage d'occupation de la structure au cours des itérations. On constate que plus 90% de l'espace alloué est utilisé en moyenne. La seule variation survient au plus fort de l'activation des lignes qui entraînent beaucoup de mouvement dans la structure entre la forte mobilité des segments et les nombreuses insertions et suppressions. L'occupation chute alors à 75% mais remonte par la suite pour se stabiliser au dessus de 90%. En maintenant le remplissage à 90% du total de la structure,

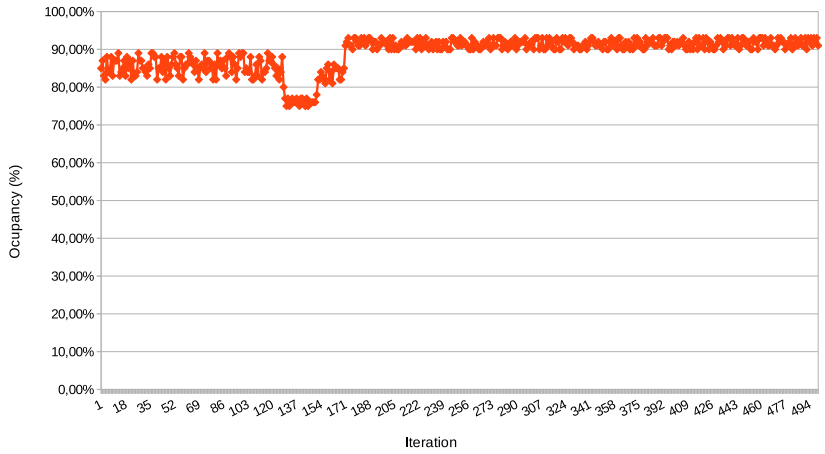
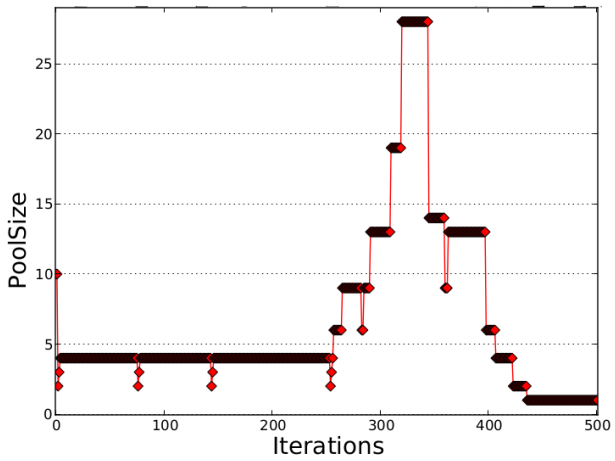


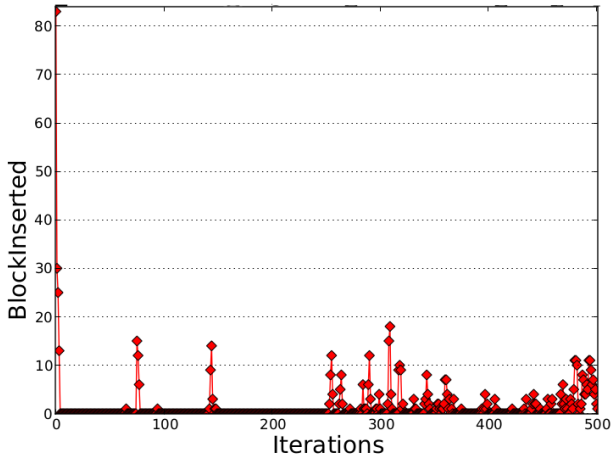
FIGURE 4.11 – Remplissage de l'espace alloué par des données dans la structure par blocs.

l'insertion lors des phases dynamiques est facilitée tandis que les chargements en mémoire

d'un bloc remplissent efficacement les niveaux de cache. Pour cette simulation comportant entre 50 000 et 200 000 segments avec un taux de remplissage de 90% et en utilisant une taille de bloc contenant 1 100 éléments, le nombre de blocs évolue entre 500 et 2 000 blocs pour contenir les nœuds et il en est de même pour contenir les segments. Par la suite, un tel nombre de blocs laisse une marge suffisante pour régler la granularité des algorithmes en parallèle où les threads se partagent le travail sur les blocs. Toujours dans les mêmes



(a) Évolution de la taille du pool d'allocation au cours de la simulation.



(b) Évolution du nombre de blocs insérés au cours de la simulation.

FIGURE 4.12 – Comportement de la structure de données par blocs durant une simulation.

conditions de test, lors d'une comparaison entre une structure de données désorganisée du fait des insertions et suppressions multiples et une structure triée nous observons une amélioration de 6% de la performance en temps CPU sur chacun des algorithmes d'interaction entre paires de segments (calcul des forces, détection de collisions). Cette optimisation par le tri montre l'importance de l'organisation mémoire pour avoir des accès

aux données par indirection efficaces. En plus de favoriser cette organisation cohérente, le tri nous aide à maintenir une bonne occupation de la structure en homogénéisant le remplissage des blocs. Enfin, notons que dans les deux tests (structure triée ou pas) en plus du tri, le pool d'allocation réduit la fragmentation de la mémoire en allouant de plus grande zone mémoire contiguës (groupes de blocs) que la taille d'un seul bloc. De ce fait, on réduit de façon automatique les accès purement aléatoire. Cependant le tri a un coût puisqu'il augmente de 3% en moyenne le temps d'une itération. Par conséquent, il est effectué tous les 75 pas de temps. Cela reste arbitraire mais l'expérience montre que cela est acceptable pour avoir suffisamment d'insertions et de suppressions sans pour autant que les accès soient aléatoires. Un choix plus automatique devra être implémenté pour mesurer le déplacement de données afin que le tri ne soit effectué uniquement que si cela est nécessaire.

Finalement, pour absorber le dynamisme, le pool d'allocation a une taille variable évoluant selon la fréquence d'allocation. Sur la Figure 4.12a et la Figure 4.12b, on constate que la simulation est relativement stable jusqu'à l'activation des nombreuses lignes après 250 pas de temps. Ainsi, la taille du pool reste stable et peu de blocs sont insérés jusqu'à ce que la phase dynamique démarre. Ensuite, avec l'augmentation de la fréquence des appels à la fonction d'insertion, la taille du pool augmente pour allouer suffisamment d'espace. Enfin, malgré des insertions toujours nombreuses comme on peut le voir sur la Figure 4.12b, la fréquence des appels systèmes diminue car l'algorithme insère des blocs provenant dans la pile des blocs disponibles, ainsi la taille du pool diminue. Si on poursuit la simulation et que le nombre de segments continue de croître avec la même tendance la taille du pool va augmenter à nouveau une fois les blocs disponibles dans la pile consommés.

Par les différents aspects mis en avant dans cette section, nous avons montré comment les optimisations sur les structures de données permettent de faire face au dynamisme de ces simulations. Différentes contributions tel que le pool d'allocation, l'insertion par localité et la réorganisation mémoire permettent cette adaptation face à l'évolution permanente du système. Cependant, la complexité et la variété entre les machines de calcul ainsi que les types de simulations obligent l'utilisateur à posséder une connaissance fine du code pour paramétrer la simulation (taille des blocs) afin d'obtenir la meilleure performance.

4.2.3 Performance du parallélisme

La Figure 4.13 illustre sur le cas test précédent, l'efficacité du parallélisme par tâches mis en œuvre sur la structure chaînée. Sur un algorithme de calcul de l'indice de Morton et de remise à zéro des forces nous maintenons une efficacité supérieure à 80% en utilisant 20 threads. Ce test est mené en moyennant l'efficacité sur 500 pas de temps. On prend donc en compte l'augmentation du nombre de segments pour démontrer aussi que l'insertion et le déplacement des données ne désorganise pas la structure de données. En effet, l'insertion et la suppression pourrait rendre le remplissage des blocs plus hétérogène et faire chuter l'efficacité du parallélisme en créant des tâches trop irrégulières. Pour cela, la méthode d'insertion par localité ainsi que le tri (effectué tous les 75 pas de temps), permettent de grouper les données dans les blocs. On maintient ainsi une forte occupation des blocs.

Enfin notre algorithme, pour grouper les tâches introduit à la Section 3.1.1, adapte la granularité pour maintenir une bonne efficacité tout au long des itérations. Pour mesurer le bénéfice, nous comparons le temps de calcul entre un algorithme attribuant un seul bloc à chaque tâche contre notre algorithme dynamique groupant les blocs pour faire des macro-tâches. Les résultats sont toujours favorables à notre algorithme dynamique mais fluctuent selon l'organisation de la structure (occupation, nombre de blocs). En moyenne, en attribuant seulement 1 seul bloc à chaque tâche nous constatons une augmentation de plus de 11% du temps de calcul toujours en utilisant les 20 threads disponibles. On peut expliquer ce résultat par le fait qu'une tâche est très légère et par conséquent en multipliant les tâches on paye le coût de l'ordonnancement openMP entre les 20 threads. On justifie alors l'importance d'agréger les blocs ensembles pour créer des macro-tâches et réduire le surcoût d'ordonnancement lié à openMP. Selon le nombre de blocs dans la simulation et le nombre de tâches affectées par thread, notre algorithme forme des groupes de 10 à 25 blocs par tâche.

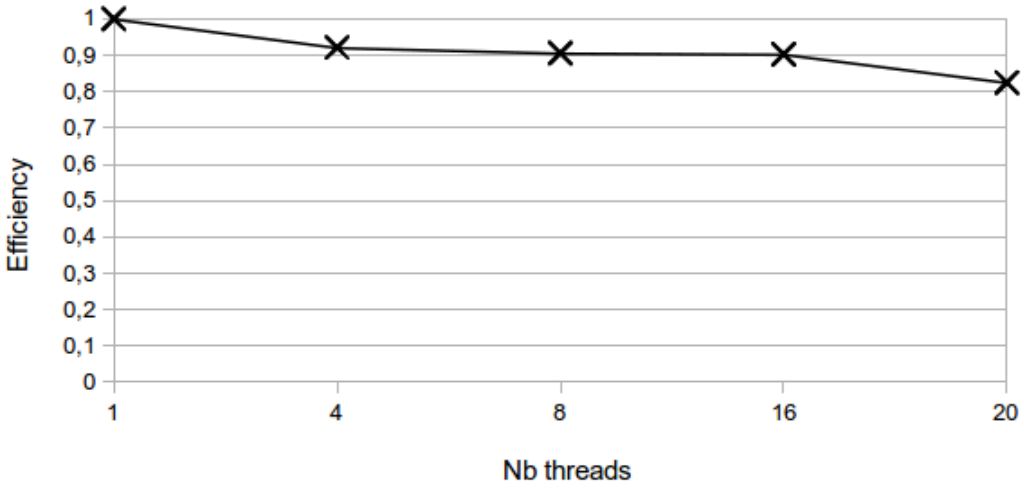


FIGURE 4.13 – Efficacité parallélisme sur la structure de données par blocs.

Dans la suite le parallélisme de la simulation est étudié sur des charges constantes de segments et sur un pas de temps complet. Le Tableau 4.2 présente les accélérations pour une simulation comportant une charge constante de 129 516 segments. Jusqu'à 20 threads l'efficacité reste supérieure à 70%. Dans ce test l'accélération est mesurée sur la totalité du pas de temps ce qui inclut des algorithmes qui ne bénéficient pas du parallélisme comme les écritures pour les sorties sur les fichiers. Cela explique en partie la légère diminution de l'efficacité plus on utilise de threads. A cela s'ajoute les phases de pré-calcul comme l'ordonnancement des tâches ou la préparation des cellules pour l'algorithme FMM qui ne sont pas parallélisés, ni recouvertes.

Avant d'étudier la scalabilité de la version hybride, nous devons déterminer le bon nombre de threads à affecter à un processus MPI sur un nœud. Le Tableau 4.3 présente

Threads	Temps (s)	Accélération	Efficacité (%)
1	54.09	1	100
2	31.27	1.73	84.8
4	17.77	3.04	76.9
8	8.75	6.18	77.2
16	4.53	11.94	74.6
20	3.78	14.3	71.5

TABLE 4.2 – Efficacité du parallélisme en mémoire partagée de 1 à 20 threads.

les résultats d'efficacité sur un seul nœud lorsque l'on utilise toutes les ressources. Pour cela le produit du nombre de processus MPI sur le nœud par le nombre de threads par processus MPI est constant et égal au nombre de cœurs de la machine soit 20 dans notre expérimentation. Sur la dernière ligne du tableau on retrouve une version en mémoire partagée avec un seul processus utilisant 20 threads qui permet, par comparaison, de mesurer le surcoût des communications. On constate que sur un seul nœud ce choix d'utiliser un seul processus est le plus performant. Aucune communication n'est nécessaire et nos différents développement permettent aux structures de données et aux accès mémoire de bénéficier de l'organisation hiérarchique de la mémoire. Cependant, dans les autres cas avec plusieurs processus et donc des communications, nous ne constatons pas une trop forte dégradation de la performance. Cela se traduit par une bonne efficacité au niveau de l'implémentation de la version du code avec décomposition de domaine. Le volume de communication est bien recouvert par les calculs et les phases de synchronisation sont peu coûteuses sur l'ensemble du pas de temps. De plus, le coût calculatoire du champ de force élastique relativise l'impact des autres algorithmes. Le cas le moins performant

MPI × Threads	Temps (s)	Efficacité
20 × 1	1.95	68.7
10 × 2	1.85	72.5
5 × 4	2.12	62.4
4 × 5	1.86	72.1
2 × 10	1.81	74.0
1 × 20	**1.79**	74.8

TABLE 4.3 – Efficacité du parallélisme en mémoire partagée en variant le nombre de processus.

est celui utilisant 5 processus MPI et 4 threads par processus. La dégradation s'explique facilement par des accès mémoires distants. En effet les 4 threads d'un processus MPI accèdent à deux bancs mémoires situé sur deux sockets différentes (Figure 4.1).

Finalement, le Tableau 4.4 présente l'efficacité de notre code jusqu'à 160 cœurs pour le même cas avec une simulation distribuée sur plusieurs nœuds contenant au total 450 000 segments. Le temps de référence est pris avec une version en mémoire partagée sur 1 nœuds avec un processus et 20 threads (ligne 1 Tableau 4.4). Nous montrons que sur la totalité d'un pas de temps nous avons une efficacité de 70% jusqu'à 8 nœuds soit 160

cœurs. La consommation de ressources de calcul (temps CPU) est très similaire au cas non distribué avec un faible volume de communication en dehors des algorithmes de calcul du champ élastique et de la détection des collisions qui représentaient déjà plus de 95% du temps de calcul.

Unités calcul	Temps (s)	Accélération	Efficacité (%)
1 nœud (20 cœurs)	27.8	1.0	100
2 nœuds (40 cœurs)	15.4	1.80	90.2
3 nœuds (60 cœurs)	12.1	2.29	76.5
4 nœuds (80 cœurs)	9.1	3.05	76.3
5 nœuds (100 cœurs)	7.09	3.96	75.9
6 nœuds (120 cœurs)	6.11	4.53	75.5
7 nœuds (140 cœurs)	5.91	4.71	67.2
8 nœuds (160 cœurs)	4.98	5.58	69.7

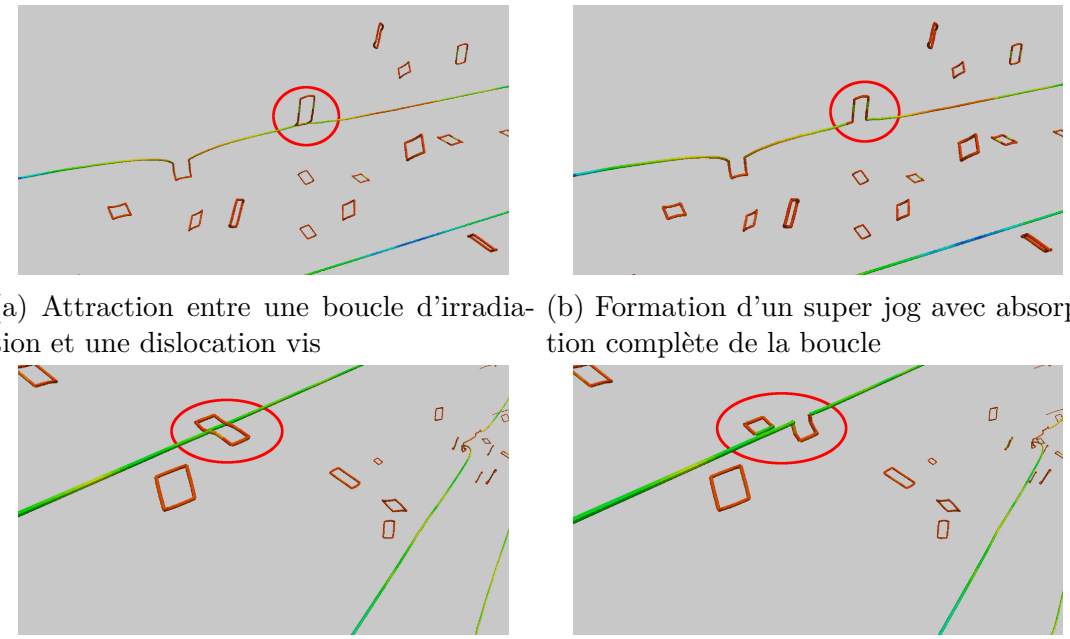
TABLE 4.4 – Efficacité du parallélisme hybride de 1 à 8 nœuds avec 20 threads par processus MPI.

Les tests dans la version hybride du code ont prouvé leur efficacité sur des simulations de 20 000 pas de temps. Cependant, pour ouvrir les perspectives de ces travaux, des tests préliminaires ont été effectués sur le cluster Curie du CEA en montant jusqu'à 256 nœuds et plus d'un millions de segments. On note sur de telles configurations des pertes d'efficacité au cours de itérations notamment dues aux synchronisations entre les sous domaines. Ces problèmes interviennent principalement dans les algorithmes de calcul de force et de détection des collisions. Le problème de l'équilibrage de charge devient central avec la multiplication des nœuds et des communications ainsi de nouvelles études sont à entreprendre pour avoir accès à de plus gros clusters.

4.3 Perspectives d'études : Formation bandes claires dans le zirconium irradié

Avec ces résultats préliminaires les premières simulations répondant au challenge industriel sont opérationnelles. Le cas test 2 introduit dans 4.1.3, qui rassemble les conditions initiales de ce challenge, présente le même comportement en terme d'utilisation des ressources et d'efficacité parallèle que l'étude précédente. La Figure 4.14 présente les capacités du code à appréhender tout un ensemble de mécanismes de nettoyage des boucles. Deux mécanismes de balayage sont ici présentés. La Figure 4.14b présente la formation d'un super jog suite à la collision entre une boucle prismatique et une dislocation vis. La Figure 4.14d montre le phénomène d'absorption partielle d'une boucle par le passage d'une dislocation vis. L'évolution d'un tel système comportant beaucoup de boucles quasi immobiles, est beaucoup plus lente avec des simulations pouvant nécessiter 45 000 pas de temps pour passer de 90 000 segments à environ 150 000. Les Figures 4.15, 4.16, 4.17 sont des images extraites de ces simulations avec la formation d'un canal contenant des dislocations mobiles diminuant ainsi la densité de boucles. En partant avec une source

de FR au centre du grain, la contrainte appliquée combinée à des conditions périodique permet la multiplication des lignes. Sur ces simulations, l'épaisseur de la bande de dislocation mobile d'environ 400Å (Figure 4.16) se crée par les décalages générés par les deux mécanismes de nettoyage présentés précédemment. Ainsi, une analyse physique de ce type de simulation va pouvoir être effectuée au cours d'un nouveau cycle de recherche. Cette étude pourra se focaliser sur les matériaux irradiés comportant une grande densité de défauts tel que celles introduites ici.



(a) Attraction entre une boucle d'irradiation et une dislocation vis

(b) Formation d'un super jog avec absorption complète de la boucle

(c) Collision entre une boucle d'irradiation et une dislocation vis.

(d) Absorption partielle d'une boucle suite à la collision avec une dislocation vis

FIGURE 4.14 – Mécanismes de balayage des boucles par le passage d'une ligne de dislocation vis.

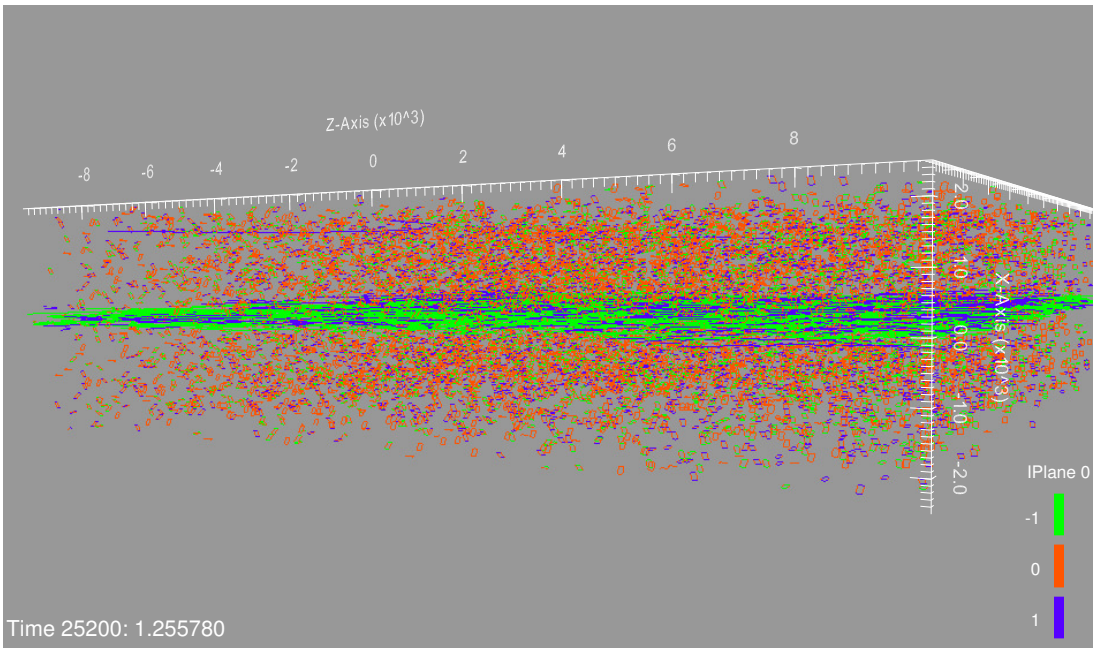


FIGURE 4.15 – Formation d'une bande de dislocation mobiles.

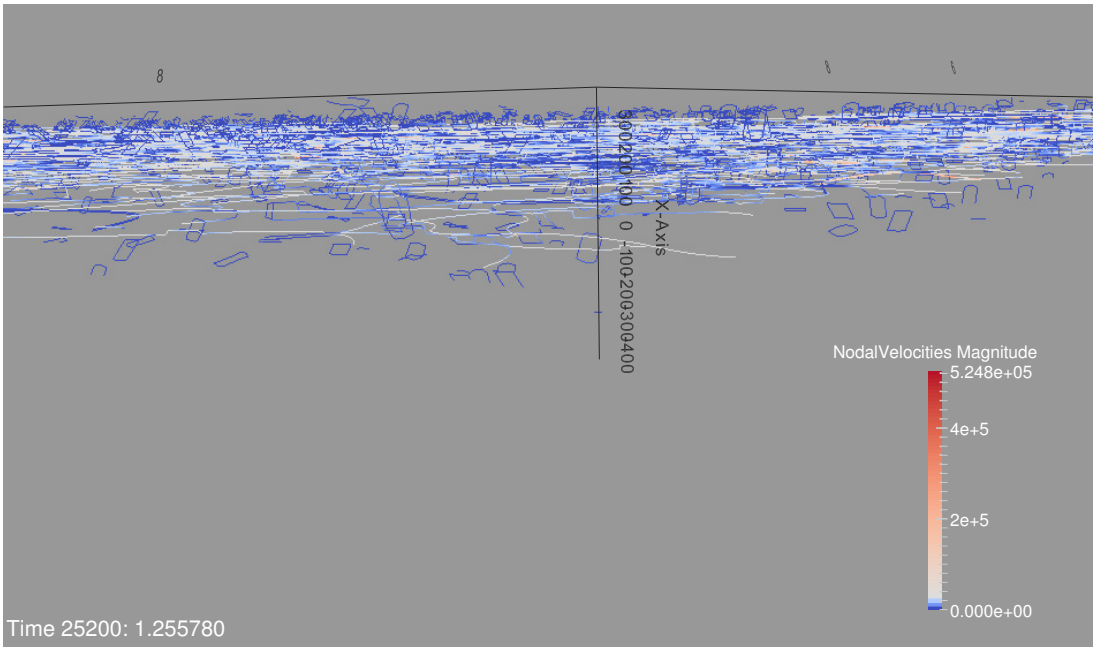


FIGURE 4.16 – Zoom sur une bande de dislocation mobiles dans la boîte de simulation.

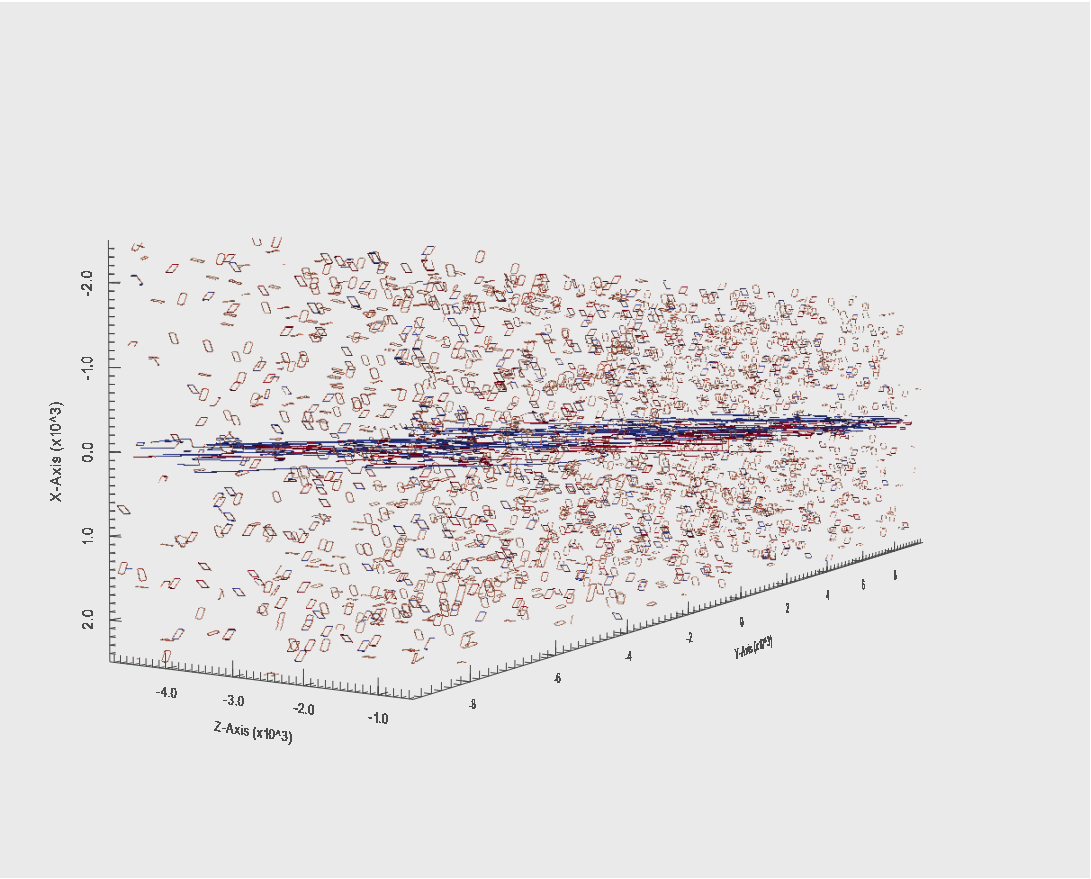


FIGURE 4.17 – Formation d'une bande de dislocation mobiles.

Conclusion et perspectives

Bilan des contributions

Les travaux de cette thèse ont porté sur les aspects calcul haute performance pour aller vers des simulations massives en dynamique des dislocations. Pour cela la complexité du problème a été étudiée, en pointant deux aspects algorithmiques fondamentalement incompatibles. D'un côté, les phases de calculs intensifs nécessitent une approche algorithmique performante en tenant compte de la hiérarchie mémoire, des capacités vectorielles et multi-cœurs des machines modernes. De l'autre côté, l'aspect dynamique lié au mouvement des dislocations est peu intensif en terme d'arithmétique mais modifie constamment l'organisation mémoire limitant ainsi la performance de la simulation.

Dans un premier temps, nous avons étudié les différents algorithmiques intervenant dans la simulation. Tout d'abord pour le calcul des forces, nous avons présenté comment un couplage dynamique entre la méthode des multipôles rapide et la méthode du rayon de coupure permettait d'obtenir des gains d'efficacité sans détériorer la précisions des calculs. Ensuite, des optimisations pour l'algorithme de détection de collision des segments ont été développées. Pour diminuer le volume de calculs à traiter, on considère d'une part le découpage hiérarchique de la boîte de simulation, liée à la méthode des multipôles rapide pour introduire un rayon de coupure et d'autre part, en utilisant le fait que les segments se déplacent dans leurs plans de glissement.

Dans un deuxième temps nous avons présenté les différents développements et optimisations pour nos structures de données ; en prenant en compte les spécificités de chaque algorithme, tant pour l'ordre et la localité des accès, que pour la modification induite par le déplacement des segments. Nous avons introduit deux structures de données, une héritée de la bibliothèque ScalFMM, pour gérer la décomposition du domaine et une autre contenant le maillage adaptée aux différentes phases algorithmiques. L'octree, dont l'organisation est basée sur l'indexation de Morton offre une décomposition spatiale du domaine de simulation. Il nous permet d'organiser les données en mémoire selon leur localité dans l'espace. Au second niveau nous avons créé une structure de données par blocs pour appréhender au mieux le dynamisme, tout en offrant un ensemble d'algorithmes pour gérer la cohérence en mémoire et donc maintenir l'efficacité de la simulation. Nous avons présenté pour cela les primitives d'insertion et de suppression optimisées pour prendre en compte la localité spatiale des données avec leurs interconnexions. Un algorithme de ré-ordonnancement est ajouté pour maintenir la correspondance entre les structures des segments, des nœuds et l'octree.

Enfin, nous proposons un parallélisme hybride classique OpenMP/MPI. Un niveau fin pour garder le bénéfice des optimisations mémoire et faire collaborer les threads à travers les différents niveaux de cache et un parallélisme grossier pour distribuer la charge de calcul pour les simulations massives. Le parallélisme en mémoire partagée est traité via un paradigme de tâches disponible depuis la version 3.1 du modèle OpenMP. Il permet de répartir le travail de manière plus souple entre les unités de calcul notamment en paramétrant la granularité des tâches ; comme par exemple en agrégeant de manière dynamique les blocs de données, ou les feuilles d'une même couleur pour le calcul du champ proche. Le parallélisme distribué utilise une décomposition du domaine en distribuant des intervalles d'indices de Morton entre les processus. Ainsi nous présentons les différentes modifications apportées aux algorithmes pour utiliser cette décomposition de domaine avec un recouvrement minimal seulement au niveau des segments partagés. Enfin, nous terminons par un algorithme d'équilibrage de charge itératif peu coûteux, en considérant uniquement les voisins directs sur l'intervalle des indices de Morton. Cet algorithme exécuté de façon régulière est adapté à la spécificité des simulations qui se déséquilibrent rapidement tout au long des itérations.

Dans le dernier chapitre, nous analysons les performances du code dans un cas réel d'utilisation. Nous mettons en avant l'apport des différentes contributions présentées dans les chapitres précédents. Nous constatons que la nouvelle version du code permet d'utiliser de manière efficace les ressources d'un cluster moderne. La méthode des multipôles rapide apporte la brique de base indispensable pour avoir une complexité linéaire sur l'ensemble de la simulation. Cet algorithme représente tout de même aux alentours de 90% de l'utilisation du temps CPU et reste un des axes de recherche prioritaire pour l'amélioration des performances. Ces résultats permettent tout de même d'envisager de démarrer une nouvelle phase de test et de recherche pour la physique de matériaux.

Perspectives

Les perspectives d'un tel travail sont nombreuses tant le domaine nécessite des compétences diverses.

D'un point de vue physique, qui reste la motivation première de telles simulations, les capacités nouvelles du code permettent d'envisager de nouvelles analyses. Le challenge pour étudier le mécanisme de formation de bandes claires dans le zirconium irradié est toujours d'actualité et nous avons démontré que nous étions capables d'atteindre des temps de simulation raisonnables pour les densités de dislocations souhaitées. Aussi, l'impact des mécanismes de montée et de glissement dévié dans la formation de super structure et dans le phénomène d'écrouissage seront l'une des voies prochainement explorées. Ces études menées dans un cadre de simulation à grande échelle permettront d'éviter les effets statistiques

D'un point de vue numérique, les travaux actuellement en cours à Inria par Pierre Blanchard, dans l'écriture de noyaux plus optimisés pour la méthode des multipôles rapide qui domine le pas de temps sont en cours. Un nouveau formalisme pour limiter le coût du calcul avec un champ anisotrope est aussi en phase d'intégration. Le couplage de code

entre une simulation de dynamique des dislocations à grande échelle et un modèle continu de type éléments finis sera aussi à envisager pour traiter de grands domaines.

Enfin d'un point de vue algorithmique qui concerne directement ces travaux, plusieurs axes peuvent être considérés pour accéder à des simulations de taille supérieure. Tout d'abord, comme nous l'avons mis en évidence en profilant le code, le calcul des interactions directes est très coûteux. Un travail plus bas niveau est nécessaire pour écrire des noyaux de calcul dédiés aux architectures. Que ce soit en vectoriel avec les fonctions intrinsèques (Avx, Avx2) ou alors en CUDA pour les GPU. Dans un second temps, et il s'agit d'une tendance générale dans la communauté du calcul hautes performances, nous pouvons penser à intégrer un moteur d'exécution pour bénéficier de la totalité des unités de calcul des plate-formes de plus en plus hétérogènes (GPU, Xeon Phi, multi-cœurs, multi-nœuds). Ces moteurs d'exécution doivent permettre sans paramétrage de l'utilisateur une utilisation optimale des ressources des calculateurs dans toute leur diversité. Des travaux dans un état avancé sont déjà en cours d'intégration pour la bibliothèque ScalFMM. Cette évolution devrait être facilement intégrée dans le code par une simple mise à jour de la bibliothèque. On doit pouvoir s'attendre à une diminution du coût de l'algorithme de calcul du champ de force en améliorant l'efficacité parallèle. Le second axe à prioriser correspond à la principale limite soulevée par ces travaux à savoir le calcul distribué à grande échelle. En augmentant la puissance de calcul (plusieurs nœuds), les simulations peuvent grossir avec des densités de dislocations plus hétérogènes et l'équilibrage de la charge de calcul est alors d'autant plus problématique. Une première approche fonctionnelle de la décomposition de domaine est proposée ici mais, même sans pousser les tests sur des dizaines de nœuds, de fortes limites apparaissent. La redistribution est limitée aux voisins directs sur l'intervalle des indices de Morton et est effectuée de façon statique (tous les 50 pas de temps). Ce problème est abordé par différentes approches en DD [83]. La gestion du déséquilibre peut être plus dynamique en calculant à chaque pas de temps l'attente entre les processus et un graphe tel qu'ils sont employés dans la communauté de l'équilibrage dynamique de graphe peut être envisagé. Ce graphe permettrait de synchroniser les processus en exprimant leurs dépendances en terme de charge (nombre de segments) et de communications notamment pour ce qui est du calcul du champ élastique par la méthode des multipôles rapide.

[]

Bibliographie

[1] E. D. Demaine A. Brodnik, S. Carlsson, J. I. Munro et R. Sedgewick : Resizable arrays in optimal time and space. *6th Workshop on Algorhtms and Data Structures (WADS)*, 1663 of LNCS, 1999.

[2] Bramas Agullo et Darve Coulaud : Task-based fmm for heterogeneous architectures. Inria research report, Inria, 2014.

[3] James Ahrens, Berk Geveci et Charles Law : 36 paraview : An end-user tool for large-data visualization. *The Visualization Handbook*, page 717, 2005.

[4] Mike P Allen et Dominic J Tildesley : *Computer simulation of liquids*. Oxford university press, 1989.

[5] VI Alshits et VL Indenbom : Mechanisms of dislocation drag. *Dislocations in Solids, edited by FRN Nabarro*, 7, 1986.

[6] R Amodeo et Nasr Mostafa Ghoniem : Dynamical computer simulation of the evolution of a one-dimensional dislocation pileup. *International journal of engineering science*, 26(7):653–662, 1988.

[7] A Arsenlis, M Rhee, G Hommes, R Cook et J Marian : A dislocation dynamics study of the transition from homogeneous to heterogeneous deformation in irradiated body-centered cubic iron. *Acta Materialia*, 60(9):3748–3757, 2012.

[8] Athanasios Arsenlis, Wei Cai, Meijie Tang, Moono Rhee, Tomas Oppelstrup, Gregg Hommes, Tom G Pierce et Vasily V Bulatov : Enabling strain hardening simulations with dislocation dynamics. *Modelling and Simulation in Materials Science and Engineering*, 15(6):553, 2007.

[9] Cédric Augonnet, Samuel Thibault, Raymond Namyst et Pierre-André Wacrenier : Starpu : a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation : Practice and Experience*, 23(2):187–198, 2011.

[10] David Baraff : Fast contact force computation for nonpenetrating rigid bodies. *In Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 23–34. ACM, 1994.

[11] Josh Barnes et Piet Hut : A hierarchical o (n log n) force-calculation algorithm. *Nature*, 324:446–449, 1986.

[12] B.Devincre : *Études de la plasticité des solides cristallins par dynamique des dislocations à l'échelle mésoscopique*. Manuscrit hdr, Paris XI-Orsay, 2005.

[13] Anastasia Braginsky et Erez Petrank : Locality-conscious lock-free linked lists. *In Distributed Computing and Networking*, pages 107–118. Springer, 2011.

[14] L. M. Brown : The self-stress of dislocations and the shape of extended nodes. *Philosophical Magazine*, 10(770001989):441–466, 1964.

[15] Vasily Bulatov et Wei Cai : *Computer simulations of dislocations*, volume 3. Oxford University Press, 2006.

[16] Vasily Bulatov, Wei Cai, Jeff Fier, Masato Hiratani, Gregg Hommes, Tim Pierce, Meijie Tang, Moono Rhee, Kim Yates et Tom Arsenlis : Scalable line dynamics in paradis. *In Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 19. IEEE Computer Society, 2004.

[17] Vasily V Bulatov et Wei Cai : Nodal effects in dislocation mobility. *Physical review letters*, 89(11):115501, 2002.

[18] Vasily V Bulatov, Moon Rhee et Wei Cai : Periodic boundary conditions for dislocation dynamics simulations in three dimensions. *In MRS Proceedings*, volume 653, pages Z1–3. Cambridge Univ Press, 2000.

[19] J.M. Burgers : Some considerations on the fields of stress connected with dislocations in a regular crystal lattice. i. *Koninklijke Nederlandse Akademie van Wetenschappen*, 42(4):292–325, 1939.

[20] Wei Cai, Athanasios Arsenlis, Christopher R Weinberger et Vasily V Bulatov : A non-singular continuum theory of dislocations. *Journal of the Mechanics and Physics of Solids*, 54(3):561–587, 2006.

[21] GR Canova, Yves Bréchet, LB Kubin, Benoit Devincre, Vassilis Pontikis et M Condat : 3d simulation of dislocation motion on a lattice : application to the yield surface of single crystals. *Solid State Phenomena*, 35:101–106, 1993.

[22] Barbara Chapman, Gabriele Jost et Ruud van der Pas : *Using OpenMP : Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[23] J. Senger C.Motz, D.Weygand et P. Gumbsch : Initial dislocation structures in 3-d discrete dislocation dynamics and their influence on microscale plasticity. *Acta Materialia*, 57:1744–1754, 2009.

[24] O Coulaud, P Fortin et J Roman : High performance BLAS formulation of the adaptive Fast Multipole Method. *Mathematical and Computer Modelling*, 51(3-4):177–188, 2010.

[25] R.LeSar C.Zhou, S.Biner : Discrete dislocation dynamics simulations of plasticity at small scales. *Acta Materialia*, 58(3):1565–1577, 2009.

[26] Robert Schöne et Matthias S.Müller Daniel Molka, Daniel Hackenberg : Memory performance and cache coherency effects on an intel nehalem multiprocessor system. *In Parallel Architectures and Compilation Techniques (PACT)*, page pages 261–270, 2009.

[27] Roland de Wit' : The continuum theory of stationary dislocations. *Solid State Physics*, 10:249–292, 1960.

[28] Erik D Demaine : Cache-oblivious algorithms and data structures. *Lecture Notes from the EEF Summer School on Massive Data Sets*, 8(4):1–249, 2002.

[29] Devincre : Dislocation dynamics simulations of slip systems interactions and forest strengthening in ice single crystal. *Philosophical Magazine*, 93(1-3):235–246, 2013.

[30] B Devincre et M Condat : Model validation of a 3d simulation of dislocation dynamics : discretization and line tension effects. *Acta metallurgica et materialia*, 40(10):2629–2637, 1992.

[31] B Devincre et LP Kubin : Simulations of forest interactions and strain hardening in fcc crystals. *Modelling and Simulation in Materials Science and Engineering*, 2(3A):559, 1994.

[32] B Devincre, R Madec, G Monnet, S Queyreau, R Gatti et L Kubin : Modeling crystal plasticity with dislocation dynamics simulations : The 'micromegas' code. *Mechanics of Nano-objects. Presses de l'Ecole des Mines de Paris, Paris*, pages 81–100, 2011.

[33] Lucile Dézerald, Lisa Ventelon, François Willaime, Emmanuel Clouet et David Rodney : Large scale ab initio calculations of extended defects in materials : screw dislocations in bcc metals. *In SNA+ MC 2013-Joint International Conference on Supercomputing in Nuclear Applications+ Monte Carlo*, page 01306. EDP Sciences, 2014.

[34] D.Garcia-Rodriguez : *Optimisation d'un code de dynamique de dislocations pour l'étude de la plasticité des aciers ferritiques.* Manuscrit, UNIVERSITE DE GRENOBLE, 2011.

[35] Jack J Dongarra, Jeremy Du Croz, Sven Hammarling et Iain S Duff : A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)*, 16(1):1–17, 1990.

[36] Julie Drouet, Laurent Dupuy, Fabien Onimus, Frédéric Mompiou, Simon Perusin et Antoine Ambard : Dislocation dynamics simulations of interactions between gliding dislocations and radiation induced prismatic loops in zirconium. *Journal of Nuclear Materials*, 449(1):252–262, 2014.

[37] L Dupuy et MC Fivel : A study of dislocation junctions in fcc metals by an orientation dependent line tension model. *Acta materialia*, 50(19):4873–4885, 2002.

[38] Rex A Dwyer : A faster divide-and-conquer algorithm for constructing delaunay triangulations. *Algorithmica*, 2(1-4):137–151, 1987.

[39] Frigo et Al. : F.f.t.w. : an adaptive software architecture for the fft. *Proceedings of the Acoustics, Speech, and Signal Processing*, 3:pp. 1381–1384, 1998.

[40] Michael A. et Al. : Optimal cache-oblivious mesh layouts. *Theory of Computing Systems*, 48:269–296, 2009.

[41] K. Farrell, T. S. Byun et N. Hashimoto : Deformation mode maps for tensile deformation of neutron-irradiated structural alloys. *Journal of Nuclear Materials*, 335(2004):471–486, 2004.

[42] Francesco Ferroni, Edmund Tarleton et Steven Fitzgerald : Gpu accelerated dislocation dynamics. *Journal of Computational Physics*, 272:619–628, 2014.

[43] MC Fivel, TJ Gosling et GR Canova : Implementing image stresses in a 3d dislocation simulation. *Modelling and Simulation in Materials Science and Engineering*, 4(6):581, 1996.

[44] AJE Foreman : Junction reaction hardening by dislocation loops. *Philosophical Magazine*, 17(146):353–364, 1968.

[45] AJE Foreman et MJ Makin : Dislocation movement through random arrays of obstacles. *Philosophical magazine*, 14(131):911–924, 1966.

[46] Pierre Fortin : *High performance parallel hierarchical algorithmic for N-body problems*. Theses, Université Sciences et Technologies - Bordeaux I, novembre 2006.

[47] FC Frank et WT Read Jr : Multiplication processes for slow moving dislocations. *Physical Review*, 79(4):722, 1950.

[48] M. L. Fredman et M. Saks : The cell probe complexity of dynamic data structures. *Proc. of STOC*, page 345–354, 1989.

[49] Leonor Frias, Jordi Petit et Salvador Roura : Lists revisited : cache conscious stl lists. *In Experimental Algorithms*, pages 121–133. Springer, 2006.

[50] Matteo Frigo, Charles E Leiserson, Harald Prokop et Sridhar Ramachandran : Cache-oblivious algorithms. *In Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 285–297. IEEE, 1999.

[51] Matteo Frigo, Charles E Leiserson, Harald Prokop et Sridhar Ramachandran : Cache-oblivious algorithms. *In Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 285–297. IEEE, 1999.

[52] DJ Gardner, CS Woodward, DR Reynolds, G Hommes, S Aubry et A Arsenlis : Implicit integration methods for dislocation dynamics. *Modelling and Simulation in Materials Science and Engineering*, 23(2):025006, 2015.

[53] Nasr M Ghoniem et R Amodeo : *Computer Simulaltion of Dislocation Pattern Formation*, volume 3. Trans Tech Publ, 1991.

[54] D Gomez-Garcia, B Devincre et LP Kubin : Dislocation patterns and the similitude principle : 2.5 d mesoscale simulations. *Physical review letters*, 96(12): 125503, 2006.

[55] Leslie Greengard et Vladimir Rokhlin : A fast algorithm for particle simulations. *Journal of computational physics*, 73(2):325–348, 1987.

[56] William Gropp, Ewing Lusk, Nathan Doss et Anthony Skjellum : A high-performance, portable implementation of the mpi message passing interface standard. *Parallel computing*, 22(6):789–828, 1996.

[57] Maurice Herlihy et J Eliot B Moss : *Transactional memory : Architectural support for lock-free data structures*, volume 21. ACM, 1993.

[58] P. B. Hirsch, R. W. Horne et M. J. Whelan : Lxviii. direct observations of the arrangement and motion of dislocations in aluminium. *Philosophical Magazine*, 1(7):677–684, 1956.

[59] John P Hirth et Jens Lothe : *Theory of dislocations*. John Wiley\ & Sons, 1982.

[60] Derek Hull et David J Bacon : *Introduction to dislocations*, volume 257. Pergamon Press Oxford, 1984.

[61] Derek Hull et David J Bacon : *Introduction to dislocations*. Butterworth-Heinemann, 2001.

[62] Martin J Hÿtch, Jean-Luc Putaux et Jean-Michel Pénisson : Measurement of the displacement field of dislocations to 0.03 å by electron microscopy. *Nature*, 423(6937):270–273, 2003.

[63] Klaus Iglberger et Ulrich Rüde : Massively parallel rigid body dynamics simulations. *Computer Science-Research and Development*, 23(3-4):159–167, 2009.

[64] IMB Inria, LABRI : Plafrim plate-forme (https ://plafrim.bordeaux.inria.fr/).

[65] Laxmikant V Kale et Sanjeev Krishnan : *CHARM++ : a portable concurrent object oriented system based on C++*, volume 28. ACM, 1993.

[66] Markus Kowarschik et Christian Weiss : An overview of cache optimization techniques and cache-aware numerical algorithms. *In Algorithms for Memory Hierarchies*, pages 213–232. Springer, 2003.

[67] Ladislas Kubin : *Dislocations, mesoscale simulations and plastic flow*, volume 5. Oxford University Press, 2013.

[68] Ladislas P Kubin, G Canova, M Condat, Benoit Devincre, V Pontikis et Yves Bréchet : Dislocation microstructures and plastic flow : a 3d simulation. *Solid State Phenomena*, 23:455–472, 1992.

[69] Monica D Lam, Edward E Rothberg et Michael E Wolf : The cache performance and optimizations of blocked algorithms. *ACM SIGOPS Operating Systems Review*, 25(Special Issue):63–74, 1991.

[70] C Lemarchand, B Devincre et LP Kubin : Homogenization method for a discrete-continuum simulation of dislocation dynamics. *Journal of the Mechanics and Physics of Solids*, 49(9):1969–1982, 2001.

[71] Tau Leng, Rizwan Ali, Jenwei Hsieh, Victor Mashayekhi et Reza Rooholamini : An empirical study of hyper-threading in high performance computing clusters. *Linux HPC Revolution*, 2002.

[72] Fuchang Liu, Takahiro Harada, Youngeun Lee et Young J Kim : Real-time collision culling of a million bodies on graphics processing units. *In ACM Transactions on Graphics (TOG)*, volume 29, page 154. ACM, 2010.

[73] Kubin Madec, Devincre : From dislocation junctions to forest hardening. *Physical review letters*, 89(25):255508, 2002.

[74] E. Martinez, J. Marian, a. Arsenlis, M. Victoria et J. M. Perlado : Atomistically informed dislocation dynamics in fcc crystals. *Journal of the Mechanics and Physics of Solids*, 56(May):869–895, 2008.

[75] M.Fivel : *Simulations de la plasticité cristalline et transitions d'échelles*. Manuscrit hdr, Université Joseph Fourier de Grenoble, 2008.

[76] Ronald E Miller et David Rodney : On the nonlocal nature of dislocation nucleation during nanoindentation. *Journal of the Mechanics and Physics of Solids*, 56(4):1203–1223, 2008.

[77] Toshio Mura : *Micromechanics of defects in solids*, volume 3. Springer, 1987.

[78] FRN Nabarro : Dislocations in a simple cubic lattice. *Proceedings of the Physical Society*, 59(2):256, 1947.

[79] L Nicola, E Van der Giessen et A Needleman : 2d dislocation dynamics in thin metal layers. *Materials Science and Engineering : A*, 309:274–277, 2001.

[80] E. Orowan : Zur kristallplastizität. i. *Zeitschrift für Physik*, 89(9-10):605–613, 1934.

[81] Yu N Osetsky et David J Bacon : An atomic-level model for studying the dynamics of edge dislocations in metals. *Modelling and simulation in materials science and engineering*, 11(4):427, 2003.

[82] M Peach et JS Koehler : The forces exerted on dislocations and the stress fields produced by them. *Physical Review*, 80(3):436, 1950.

[83] Olga Pearce, Todd Gamblin, Bronis R de Supinski, Tom Arsenlis et Nancy M Amato : Load balancing n-body simulations with highly non-uniform density. *In Proceedings of the 28th ACM international conference on Supercomputing*, pages 113–122. ACM, 2014.

[84] R Peierls : The size of a dislocation. *Proceedings of the Physical Society*, 52(1):34–37, 1940.

[85] M. Polanyi : Über eine art gitterstörung, die einen kristall plastisch machen könnte. *Zeitschrift für Physik*, 89(9-10):660–664, 1934.

[86] V. Raman et S.S. Rao : Succint dynamic dictionaries and trees. *Proc. of Automata*, 2719 of LNCS(357–366), 2003.

[87] St Raoul : Intergranular brittle fracture of a low alloy steel induced by grain boundary segregation of impurities : influence of the microstructure. Rapport technique, CEA/Saclay, Dept. d'Etudes du Comportement des Materiaux (DECM), 91-Gif-sur-Yvette (France), 1999.

[88] Moono Rhee, HM Zbib, JP Hirth, H Huang et T De la Rubia : Models for long-/short-range interactions and cross slip in 3d dislocation simulation of bcc single crystals. *Modelling and Simulation in Materials Science and Engineering*, 6(4):467, 1998.

[89] R.Madec : Parallélisation du code de dynamique des dislocations micromegas. Research report, Onera, 2011.

[90] F Roters, P Eisenlohr, L Hantcherli, DD Tjahjanto, TR Bieler et D Raabe : Overview of constitutive laws, kinematics, homogenization and multiscale methods in crystal plasticity finite-element modeling : Theory, experiments, applications. *Acta Materialia*, 58(4):1152–1211, 2010.

[91] Radu Rugina et Martin Rinard : Automatic parallelization of divide and conquer algorithms. *In ACM SIGPLAN Notices*, volume 34, pages 72–83. ACM, 1999.

[92] Hans Sagan : *Space-filling curves*, volume 18. Springer-Verlag New York, 1994.

[93] KW Schwarz : Simulation of dislocations on the mesoscopic scale. i. methods and examples. *Journal of Applied Physics*, 85(1):108–119, 1999.

[94] A Serra et DJ Bacon : Atomic-level computer simulation of the interaction between $\frac{1}{3}\langle 1\ 1\ \bar{2}\ 0\rangle\{1\ \bar{1}\ 0\ 0\}$ dislocations and $\frac{1}{3}\langle 1\ 1\ \bar{2}\ 0\rangle$ interstitial loops in α-zirconium. *Modelling and Simulation in Materials Science and Engineering*, 21(4):045007, 2013.

[95] VB Shenoy, RV Kukta et R Phillips : Mesoscopic analysis of structure and strength of dislocation junctions in fcc metals. *Physical Review Letters*, 84(7):1491, 2000.

[96] X.J Shi : *ETUDE PAR SIMULATIONS DE DYNAMIQUE DES DISLOCATIONS DES EFFETS D'IRRADIATION SUR LA PLASTICITE DE LA FERRITE A HAUTE TEMPERATURE*. Manuscrit, UNIVERSITE PIERRE ET MARIE CURIE, 2014.

[97] Chan-Sun Shin, Marc C Fivel, Marc Verdier et SC Kwon : Numerical methods to improve the computing efficiency of discrete dislocation dynamics simulations. *Journal of Computational Physics*, 215(2):417–429, 2006.

[98] Ryan B Sills et Wei Cai : Efficient time integration in dislocation dynamics. *Modelling and Simulation in Materials Science and Engineering*, 22(2):025003, 2014.

[99] Geoffrey Ingram Taylor : The mechanism of plastic deformation of crystals. part i. theoretical. *Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character*, pages 362–387, 1934.

[100] Marc Tchiboukdjian : *Algorithmes parallèles efficaces en cache Applications à la visualisation scientifique*. Spécialité informatique, UNIVERSITÉ DE GRENOBLE, 2010.

[101] Dmitry Terentyev, Napoleón Anento et Anna Serra : Interaction of< 100> loops with carbon atoms and< 100> dislocations in bcc fe : An atomistic study. *Journal of Nuclear Materials*, 420(1):9–15, 2012.

[102] Dmitry Terentyev, P Grammatikopoulos, DJ Bacon et Yu N Osetsky : Simulation of the interaction between an edge dislocation and a< 100> interstitial dislocation loop in α-iron. *Acta Materialia*, 56(18):5034–5046, 2008.

[103] Erik Van der Giessen et Alan Needleman : Discrete dislocation plasticity : a simple planar model. *Modelling and Simulation in Materials Science and Engineering*, 3(5):689, 1995.

[104] A Vattré, B Devincre, F Feyel, R Gatti, S Groh, O Jamond et A Roos : Modelling crystal plasticity by 3d dislocation dynamics and the finite element method : the discrete-continuous model revisited. *Journal of the Mechanics and Physics of Solids*, 63:491–505, 2014.

[105] A Vattré, B Devincre, F Feyel, R Gatti, S Groh, O Jamond et A Roos : Modelling crystal plasticity by 3d dislocation dynamics and the finite element method : the discrete-continuous model revisited. *Journal of the Mechanics and Physics of Solids*, 63:491–505, 2014.

[106] Todd Veldhuizen : Expression templates. *C++ Report*, 7(5):26–31, 1995.

[107] M Verdier, M Fivel et In Groma : Mesoscopic scale simulation of dislocation dynamics in fcc metals : Principles and applications. *Modelling and Simulation in Materials Science and Engineering*, 6(6):755, 1998.

[108] Stephen R. Walli : The posix family of standards. *StandardView*, 3(1):11–17, mars 1995.

[109] Zhiqiang Wang, Nasr Ghoniem, Sriram Swaminarayan et Richard LeSar : A parallel algorithm for 3d dislocation dynamics. *Journal of computational physics*, 219(2):608–621, 2006.

[110] Jie Deng Wei Cai et Keonwook Kang : A short course on ddlab and paradis. Rapport technique, Department of Mechanical Engineering, Stanford University, Stanford, 2005.

[111] D Weygand, J Senger, C Motz, W Augustin, V Heuveline et P Gumbsch : High performance computing and discrete dislocation dynamics : Plasticity of micrometer sized specimens. *In High Performance Computing in Science and Engineering'08*, pages 507–523. Springer, 2009.

[112] Vladislav Yastrebov, Georges Cailletaud et Frédéric Feyel : Couplage de codes éléments finis et dynamique discrète de dislocations. *In CSMA 2013-11ème colloque national en calcul des structures*, pages 2–p, 2013.

Annexe A

Liste des publications

— Arnaud Etcheverry, Large scale hybrid implementation in dislocation dynamics simulation ECCOMAS-CFD 2014 conference in Barcelona, SPAIN
— Arnaud Etcheverry, Pierre Blanchard, Olivier Coulaud, Laurent Dupuy, OptiDis : a MPI/OpenMP Dislocation Dynamics Code for Large Scale Simulations, MMM 2014 conference in Berkley, USA
— Arnaud Etcheverry ; OptiDis, a parallel dislocation dynamics code for large scale simulations, Workshop OptiDis 2014