Rééquilibrage de charge dans les solveurs hiérarchiques pour machines massivement parallèles

Paul Beziau

THÈSE PRÉSENTÉE

POUR OBTENIR LE GRADE DE

DOCTEUR
DE L'UNIVERSITÉ DE BORDEAUX

ECOLE DOCTORALE MATHÉMATIQUES ET

INFORMATIQUE

INFORMATIQUE

Par **Paul Beziau**

Rééquilibrage de charge dans les solveurs hiérarchiques pour machines massivement parallèles

Sous la direction de : **Raymond Namyst**

Soutenue le 25 décembre 2021

Membres du jury :

M. Brice GOGLIN	Directeur de Recherche	INRIA	Examinateur
M. Alfredo BUTTARI	Directeur de Recherche	CNRS	Rapporteur
M. Pierre FORTIN	Maître de Conférences	Université de Lille	Rapporteur
M. Raymond NAMYST	Professeur des universités	Université de Bordeaux	Directeur

Résumé

Dans le cadre de la résolution des équations de Maxwell sur des maillages 3D surfaciques, il est nécessaire de faire intervenir des solveurs algébriques coûteux en temps et en mémoire. Pour réduire ces coûts, il est possible de compresser les matrices manipulées, à l'aide de matrices hiérarchiques. L'utilisation de ce type de matrice permet de diminuer drastiquement les ressources nécessaires, mais induit un déséquilibre important dans la charge des différents processus impliqués dans le calcul. Ce déséquilibre peut limiter le passage à l'échelle des codes de simulation exploitant un grand nombre de nœuds de calculs.

Dans cette thèse nous nous sommes intéressés à réduire ce déséquilibre, en utilisant des méthodes de rééquilibrage statique et dynamique.

Les méthodes de rééquilibrage statique nous ont permis de changer la distribution des différents blocs de la matrice parmi les processus impliqués dans le calcul. Pour cela nous avons utilisé la notion de processus virtuels, qui nous a permis de redistribuer les matrices hiérarchiques parmi les différents processus. Nous avons également développé des modèles pouvant prédire le temps d'exécution des différentes parties du calcul, ainsi que son temps d'exécution total.

Nous avons ensuite étudié la faisabilité de différentes méthodes de rééquilibrage dynamique, toujours dans le contexte des matrices hiérarchiques. Nous avons analysé plusieurs techniques différentes, nous permettant de mettre en évidence les limites de plusieurs d'entre elles dans le cadre de systèmes de tâches généralistes. Nous avons également mis en évidence plusieurs difficultés d'implémentation de ces méthodes, et nous avons proposé plusieurs possibilités pour les résoudre.

La première contribution principale de cette thèse est une méthode de redistribution statique des blocs d'une matrice hiérarchique permettant un gain de 15% dans la durée de la factorisation de ce type de matrices.

La seconde contribution principale de cette thèse est une suite d'algorithmes permettant d'effectuer des vols de tâches en mémoire distribuée sur l'étape d'assemblage de matrices hiérarchiques, ce qui permet un rééquilibrage de cette partie, permettant de gagner jusqu'à 50% de temps d'exécution dans les cas les plus extrêmes. Ces algorithmes permettent également d'effectuer un vol de tâche sur un runtime généraliste, en respectant les dépendances entre les différentes parties des calculs.

Abstract

In the context of the resolution of Maxwell's equation on 3D surface meshes, it is necessary to use algebraic solvers which are costly in time and memory. To reduce these costs, an improvement has been to apply hierarchical compression techniques on matrices. The resulting matrices allows the required ressources to be drastically reduced, but induce an important load imbalance between the different processes involved in the computation. This imbalance is a limiting factor for the scalability of simulation codes with a large number of computational nodes.

In this thesis, we sought to reduce this imbalance, by using both static and dynamic load balancing methods.

Static load balancing methods have allowed us to change the distribution of the different blocks of the matrix among the processes involved in the computation. For this purpose we had set up the notion of virtual processes which allowed us to redistribute the hierarchical matrices among the different processes. We have also developed models that could predict the execution time of different parts of the computation, as well as its total execution time in order to dispatch computation efficiently.

Then, we have studied the feasibility of different dynamic load balancing methods, again in the context of hierarchical matrix. We have analyzed different techniques, allowing us to highlight the limits specifics to several of them in the context of general purpose task programming environnements. We have also highlighted several implementation difficulties and have proposed several possibilities to solve them.

The first major contribution of this thesis is a static redistribution method of the matrix blocks, which allows a gain up to 15% in the duration of the factorization of this kind of hierarchically compressed matrices.

The second major contribution of this thesis is the introduction of a a set of algorithms providing a work stealing mechanism into our task-based environment over distributed memory. This mechinasm not only completely balances the load of the assembly phase, but also allow to steal tasks within a task graph while properly respecting the dependencies between the differents parts of the computation.

Remerciements

Je souhaite tout d'abord remercier mon directeur de thèse, Raymond Namyst, ainsi que mes encadrants, Cédric Augonnet et Mathieu Kuhn pour m'avoir suivi pendant trois ans, et sans lesquels cette thèse n'aurait pas été possible. Merci également à David Goudin pour avoir suivi ma thèse avec bienveillance.

Merci également à Alfredo Buttari et à Pierre Fortin pour leur relecture attentive de ce manuscrit, ainsi que pour les différentes remarques qu'ils ont pu me faire sur celui-ci. Je les remercie aussi pour avoir participé au jury de ma soutenance. À ce sujet, je souhaite également remercier Brice Goglin pour avoir présidé ce jury.

Merci à Yann, Paul et Julien pour avoir supporté mes jeux de mots pendant tout ce temps ! Merci à Claire, Corentin, Damien, Gabriel, Justine, Stephane et tous ceux que j'ai pu rencontrer lors de ces trois ans pour toutes les discussions intéressantes que nous avons pu avoir sur un sujet ou sur un autre.

Enfin je souhaiterais remercier mes proches qui m'ont soutenu dans cette entreprise, tout d'abord mes parents ainsi que mes sœurs qui ont été là pendant toute cette période. Merci également à Lucile et Sylvain pour toutes ces soirées jeux de société et/ou film. Merci à Lætitia, Mélanie et Samuel pour avoir également été présents à la plupart de ces soirées, qui m'ont toujours permis d'avoir un bol d'air frais. Merci également à Antoine et Philibert pour m'avoir eux aussi encouragé pendant ce doctorat. Pour finir, merci à Maxime et Julie pour ces sorties au bassin d'Arcachon, qui m'ont apporté un dépaysagement bienvenu !

Table des matières

1 Introduction **7**
- 1.1 Contexte applicatif et industriel . . . 7
- 1.2 Objectifs et contributions . . . 8
- 1.3 Organisation du document . . . 8

2 Contexte Applicatif **10**
- 2.1 La simulation numérique pour le calcul du niveau de furtivité électromagnétique . . . 11
  - 2.1.1 Surface Équivalente Radar (SER) . . . 12
  - 2.1.2 Modèle physique et formulation intégrale . . . 13
  - 2.1.3 Méthode des éléments finis de frontière . . . 14
- 2.2 Solveurs linéaires directs pour la Méthode des éléments finis de frontière . . . 15
  - 2.2.1 Solveur direct dense . . . 15
  - 2.2.2 Solveur direct avec compression . . . 18
- 2.3 Un code industriel 3D surfacique pour le calcul de SER . . . 24
- 2.4 Exemples de simulations . . . 25
  - 2.4.1 Cônes sphères IEEE . . . 25
  - 2.4.2 Lanceur CNES (Workshop ISAE) . . . 26
  - 2.4.3 Cas test UAV (Workshop ISAE) . . . 26
- 2.5 Calculateurs utilisés pour les simulations . . . 26

3 Outils pour le calcul haute performance **29**
- 3.1 Architectures des supercalculateurs contemporains . . . 30
  - 3.1.1 Des supercalculateurs vectoriels aux grappes de PC . . . 31
  - 3.1.2 L'avènement du multicœur . . . 32
  - 3.1.3 Les accélérateurs de calculs . . . 34
- 3.2 Langages et outils pour l'exploitation des architectures à mémoire partagée . . . 35
  - 3.2.1 Programmation orientée "threads" . . . 35
  - 3.2.2 Programmation orientée "tâches" . . . 39
- 3.3 Dépendances hiérarchiques . . . 44
- 3.4 Programmation des architectures distribuées . . . 44
  - 3.4.1 Le modèle PGAS . . . 45
  - 3.4.2 *Message Passing Interface* . . . 45
  - 3.4.3 Accès mémoire distants . . . 46
- 3.5 Composition de paradigmes : MPI + X . . . 49
  - 3.5.1 Les difficultés posées par le mélange de deux modèles qui n'ont pas été conçus pour fonctionner ensemble . . . 49
  - 3.5.2 Le problème des tâches de communication . . . 50
- 3.6 Rééquilibrage de charge dans les environnements de programmation distribués . . . 52

4 Etat de l'art **54**
4.1 Solveurs denses 54
4.1.1 En mémoire partagée 55
4.1.2 En mémoire distribuée 56
4.2 Solveurs creux 57
4.2.1 Solveurs creux directs 57
4.2.2 Solveurs creux itératifs 57
4.3 Solveurs avec compression 58

5 Contributions **59**
5.1 Équilibrage statique 61
5.1.1 Influence du mapping sur les performances 61
5.1.2 Processus virtuels 65
5.1.3 Éviter les cas pathologiques à l'aide des processus virtuels 72
5.1.4 Rééquilibrage mémoire 76
5.1.5 Rééquilibrage guidé par une connaissance a priori des temps de calcul 80
5.1.6 Prédire les temps de calcul à l'aide de modèles de performance 87
5.1.7 Expériences avec une simulation du problème à N-corps (Barnes-Hut) 90
5.1.8 Extraction des métriques 93
5.2 Équilibrage dynamique 96
5.2.1 Délégation de tâches 97
5.2.2 Tâches distribuées 99
5.2.3 Principe du vol de travail 104
5.2.4 Détection de terminaison 110
5.2.5 Valorisation dans un code d'aérodynamique hypersonique 114
5.2.6 Vol de tâche coopératif 116
5.2.7 Vol de tâche one-sided 121
5.2.8 Vol de tâche par rendez-vous 124
5.2.9 Support pour le langage Fortran 126
5.2.10 Bilan 127

6 Conclusion et perspectives **129**
6.1 Contributions 129
6.2 Perspectives 129

Chapitre 1

Introduction

L'étude des phénomènes en électromagnétisme est un sujet majeur pour des domaines tels que l'avionique ou le spatial, où il s'agit de traiter les problématiques de compatibilité électromagnétique ou de furtivité radar pour ne citer que quelques exemples. La simulation numérique permet d'analyser avec précision les phénomènes tels que l'influence de la forme des objets sur les courants induits à la surface ou encore sur la réflexion des ondes électromagnétiques à leur contact.

1.1 Contexte applicatif et industriel

Pour effectuer ces simulations, les codes numériques doivent résoudre les équations de Maxwell de manière approchée, en opérant sur des maillages surfaciques 3D par exemple. Dans ce cas, la méthode des éléments finis de frontière peut être utilisée. On aboutit alors à la résolution d'un système linéaire $Ax = B$, A et B étant des matrices denses de plusieurs millions d'inconnues à coefficients complexes.

Le code de résolution utilisé au CEA s'appuie sur une méthode directe, particulièrement robuste, et adaptée à la résolution d'un très grand nombre de seconds-membres. Celui-ci se décompose en trois grandes étapes numériques : l'assemblage de la matrice A, sa factorisation, et enfin la résolution du système d'équation résultant. Le coût de ces étapes rend évidemment nécessaire l'utilisation de techniques de parallélisation permettant au solveur d'utiliser des clusters de machines manycores.

Pour pouvoir utiliser de manière efficace ces machines, les matrices manipulées ont été divisées en plusieurs blocs. Ces derniers ont ensuite été affectés aux différents processus impliqués dans le calcul, ce qui a par la suite permis d'obtenir une bonne efficacité dans le code parallèle.

Toutefois, même les solveurs parallèles performants atteignent rapidement leurs limites lorsque l'on atteint quelques centaines de milliers, ou quelques millions d'inconnues. Pour ces raisons, des techniques de compression de matrices ont été introduites. En particulier, la méthode de compression dite *Block Low-Rank* (BLR) consiste à approximer les blocs matriciels de rangs numériques faibles sous la forme d'un produit de deux matrices de plus petite taille. Les blocs ainsi compressés réduisent considérablement l'empreinte mémoire de la matrice, et abaissent également les coûts de calcul associés. Bien sûr, pour une taille de bloc choisie, une certaine proportion de blocs restent incompressibles, ce qui borne les gains pouvant être espérés en pratique. Si le partitionnement de la matrice en petits blocs augmente le nombre de blocs compressibles, il porte en revanche préjudice à l'efficacité des calculs s'appliquant aux blocs denses.

C'est pourquoi une nouvelle méthode de compression dite hiérarchique a été introduite, qui consiste à subdiviser les blocs non compressibles en répé-

tant l'étape de compression sur chacun des sous-blocs obtenus. Cette nouvelle méthode permet ainsi de réduire encore la complexité en temps et en espace à $O(nlog(n))$.

1.2 Objectifs et contributions

Si ces méthodes permettent effectivement une réduction du nombre d'opérations impliquées, ce gain n'est pas uniformément réparti sur la matrice, ce qui a pour effet d'introduire des déséquilibres de charge de calcul entre les différents blocs. De manière indirecte, cela aura donc un impact sur la charge des processeurs en cas de répartition naïve des blocs sur les unités de calcul. En effet, les parties peu ou pas compressées se concentrant typiquement sur quelques zones particulières de la matrice (notamment aux abords de la diagonale), une distribution des données ne prenant pas en compte cette caractéristique aura tendance à produire une répartition non équitable des calculs.

L'équilibrage de charge est par essence un problème intrinsèque à ce type de traitement puisque, comme avec les solveurs creux, un phénomène de « remplissage » apparaît au fur et à mesure de l'étape de factorisation. Dans un solveur avec compression hiérarchique, ce remplissage est dû à l'apparition de blocs de moins en moins compressibles. C'est l'imprévisibilité de ce phénomène qui rend les stratégies d'équilibrage purement statique inadéquates.

Cette thèse s'intéresse aux problèmes de répartition de charge dans des applications irrégulières, en particulier celles qui manipulent des structures de données hiérarchiques, telles que l'on en trouve dans les solveurs directs reposant sur des techniques de compression hiérarchique pour lesquels nous considérons non seulement la phase de construction de la matrice (assemblage), mais également la phase de factorisation qui met en oeuvre des dépendances complexes. Bien que la mise en oeuvre de techniques de compression, voire de solveurs avec des techniques de compression hiérarchiques, soit à l'heure actuelle particulièrement étudiée, il n'existe pas à l'heure actuelle, et à notre connaissance, de solveur hiérarchique distribué à grande échelle qui considère spécifiquement la problématique du rééquilibrage, qu'il soit statique, ou dynamique.

Les contributions de cette thèse sont les suivantes :

— La mise en place de modèles de coûts pour estimer le temps d'exécution lors de la factorisation d'une matrice compressée hiérarchiquement
— La mise en place d'un système que nous nommerons "processus virtuels" étendant le concept de distribution 2D bloc cyclique en vue de corriger la distribution des blocs
— L'introduction d'un système de vol de tâche pouvant être utilisé sur le support exécutif qui met en oeuvre le parallélisme de tâches au sein de notre solveur direct hiérarchique

1.3 Organisation du document

Ce document est organisé comme suit. Après cette introduction dans le Chapitre 1, nous introduisons le contexte dans lequel cette thèse s'inscrit dans le Chapitre 2, en exposant tout d'abord le problème physique résolu, et la méthode numérique associée dans la Partie 2.1. Dans la Partie 2.2, nous détaillons

comment le système linéaire associé à ce problème est résolu, et comment nous pouvons tirer parti des techniques de compression. Les Parties 2.3 et 2.4 présentent le code en tant que tel, ainsi que des exemples de calculs qui nous servirons pour évaluer les performances.

Dans le Chapitre 3, nous donnons un panorama de l'évolution des machines (Partie 3.1), et des paradigmes de programmation (Partie 3.2). Nous présentons les dépendances hiérarchiques qui sont une extension du modèle classique de dépendances de tâches dans la Partie 3.3. Enfin, nous nous attardons sur les environnements de programmation en mémoire distribuée (Partie 3.4), et sur la difficulté pour composer les modèles de programmation dans la Partie 3.5.

Le Chapitre 4 rappelle l'état de l'art dans les domaines liés à cette thèse. Après un rappel de l'existant en ce qui concerne les solveurs denses (Partie 4.1) et creux (Partie 4.2), nous donnons ensuite un aperçu des travaux autour des solveurs avec des techniques de compression dans la Partie 4.3.

Nous exposons nos contributions dans une seconde partie. Le Chapitre 5.1 décrit les méthodes de redistribution statique que nous avons mises en place. Après avoir montré l'importance d'une bonne répartition des calculs et des données (5.1.1), nous exposons le concept de *processus virtuel* qui formalise un sous-ensemble des calculs que l'on peut choisir d'attribuer à tel ou tel processus (5.1.2), et montrons que ce concept permet d'éviter des cas de déséquilibres pathologiques (5.1.3). Nous étudions ensuite plusieurs méthodes pour attribuer ces processus virtuels aux différents processus, avec un ordonnancement guidé par la mémoire (5.1.4), ou par une estimation du temps de calcul des tâches (5.1.5 et 5.1.6). La Partie 5.1.8 détaille les mécanismes conçus dans notre support exécutif pour extraire ces métriques, et injecter des informations utiles pour l'ordonnancement.

Puis, dans le Chapitre 5.2 nous présentons différentes techniques que nous avons explorées afin de rééquilibrer la charge de manière dynamique. Une première stratégie consiste à déléguer l'exécution des tâches locales sur des processus distants que l'on suppose moins chargés (5.2.1). Une autre approche consiste à écrire les tâches avec un parallélisme de type MPMD avec des appels à MPI au sein de ces tâches (5.2.2). La Partie 5.2.3 introduit la notion de vol de tâches, et la Partie 5.2.4 présente le problème de terminaison introduit avec le vol de tâche. Les Parties 5.2.6, 5.2.7 et 5.2.8 présentent respectivement les approches de vols de tâches coopératif, *one-sided*, et par rendez-vous.

Enfin, nous concluons dans le Chapitre 6 avec un résumé des différentes avancées de cette thèse, avant d'exposer les perspectives ouvertes par ce travail.

Chapitre 2

Contexte Applicatif

Ce chapitre pose le contexte applicatif qui a motivé le travail de thèse exposé dans ce manuscrit. Ce travail s'installe dans un contexte de simulation industrielle sur le centre du CESTA au CEA DAM. En particulier, le code de simulation qui sera la matière première pour le rééquilibrage de charge sur architectures à mémoire distribuée a pour but l'évaluation du niveau de détectabilité électromagnétique d'objets soumis à un éclairement radar. Les effets électromagnétiques qui entrent en jeu pour la furtivité radar sont régis par les équations de Maxwell. Les sections suivantes détaillent d'abord le contexte applicatif ainsi que les méthodes numériques en oeuvre pour la résolution approchée des équations de Maxwell.

2.1 La simulation numérique pour le calcul du niveau de furtivité électromagnétique

Le niveau de furtivité des têtes nucléaires françaises fait partie des performances à garantir par la simulation sur le centre CESTA du CEA DAM. Les paragraphes suivants donnent des éléments de compréhension concernant la simulation numérique pour la furtivité électromagnétique à travers un des codes de calcul utilisé pour déterminer la Surface Equivalente Radar d'objets.

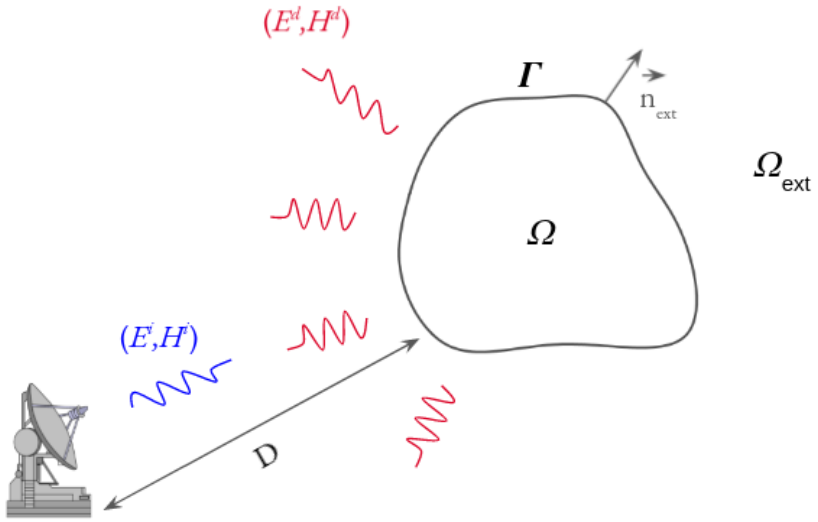


FIGURE 2.1 – Eclairement et détection d'un objet par un radar. Le radar émet un champ incident (E^i, H^i) qui engendre un champ électromagnétique (E^d, H^d) diffracté par l'objet. La détectabilité de l'objet dépend alors de la puissance du champ diffracté dans la direction du radar. Un objet furtif, au sens électromagnétique du terme, a donc pour but de limiter fortement l'intensité du champ diffracté.

2.1.1 Surface Équivalente Radar (SER)

La Surface Équivalente Radar (SER) est une quantité physique exprimée en mètres carrés. Elle quantifie le niveau de discrétion d'un objet en mesurant ses capacités de diffraction lors d'un éclairement par un radar comme sur la Figure 2.1. La SER varie selon la forme et la nature des matériaux qui composent l'objet, selon l'angle d'incidence du signal émis par le radar et de l'angle d'observation, mais aussi selon la fréquence d'émission du radar.

Plus précisément, la SER se définit comme le rapport entre le champ électrique incident et le champ électrique diffracté par l'objet. Les radaristes préfèrent souvent exprimer la SER σ_{dB} en décibels mètres carrés ($dB.m^2$). Dans cette unité, elle est donnée par la formule suivante :

$$\sigma_{dB} = \lim_{D \to \infty} 10 \log_{10} \left(4\pi D^2 \frac{|E^d|^2}{|E^i|^2} \right),$$

avec E^i le champ électrique incident, E^d le champ électrique diffracté par l'objet.

Pour un objet donné, il est alors possible de calculer sa SER selon un balayage angulaire. Lorsque le radar sert à la fois comme émetteur et comme récepteur, comme sur la Figure 2.1, on parle de balayage monostatique.

Il est également possible de dissocier le radar d'émission du radar d'observation. On parle alors de balayage bistatique. Dans ce cas, les angles d'émission et d'observation diffèrent, et plusieurs observations peuvent être effectuées pour un seul éclairement. Ceci permet par exemple de tracer un diagramme de SER dans le plan (voir la Figure 2.2).

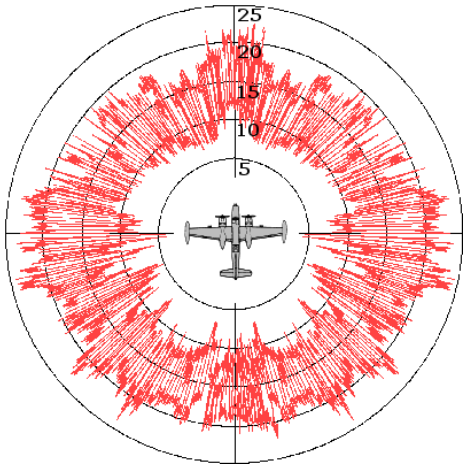


FIGURE 2.2 – Diagramme de SER du bombardier B-26 à 3GHz dans un plan (selon Skolnik). La SER de l'objet varie entre 5 et 25 $dB.m^2$ selon l'angle d'observation. (source : fr.wikipedia.org)

Afin de déterminer E^d en fonction de E^i, il est toutefois nécessaire de résoudre les équations qui modélisent les phénomènes électromagnétiques en jeu lors de l'éclairement d'un objet par une onde : les équations de Maxwell. Ces équations sont données dans le paragraphe suivant.

2.1.2 Modèle physique et formulation intégrale

Dans le milieu extérieur Ω_{ext} à l'objet dont on souhaite déterminer la SER (voir la Figure 2.1), le champ électromagnétique diffracté par l'objet $(\overrightarrow{E^d}, \overrightarrow{H^d})$ vérifie les équations suivantes :

$$\begin{cases} rot\overrightarrow{E^d} + i\omega\epsilon_0\overrightarrow{H^d} = \overrightarrow{0}, \\ rot\overrightarrow{H^d} - i\omega\mu_0\overrightarrow{E^d} = \overrightarrow{0}, \end{cases} \tag{2.1}$$

$$\vec{n} \wedge \overrightarrow{E^d} = -\vec{n} \wedge \overrightarrow{E^i}, \tag{2.2}$$

$$\lim_{\vec{r}\to\infty} |\vec{r}| \left| \overrightarrow{E^d} + \eta_0 \frac{\vec{r}}{|\vec{r}|} \wedge \overrightarrow{H^d} \right| = 0. \tag{2.3}$$

Parmi elles, les deux premières équations (2.1) forment un système : les équations de Maxwell. Elles modélisent les phénomènes liés à l'électromagnétisme. Elles sont données ici en régime harmonique, avec ω la pulsation qui dépend de la fréquence f de l'onde électromagnétique selon la relation $\omega = 2\pi f$. Les quantités ϵ_0 et μ_0 correspondent à la permittivité et à la perméabilité du milieu extérieur Ω_{ext}.

Les équations de Maxwell sont complétées par une condition à la surface de l'objet qui tient compte de son caractère conducteur, donnée par l'équation (2.2), où $\vec{n}$ est la normale sortante à l'objet. De par le caractère non borné du milieu extérieur, une condition de radiation à l'infini est également nécessaire, donnée ici par l'équation (2.3), où η_0 est l'impédance du milieu extérieur Ω_{ext}.

Il est possible de dériver une représentation intégrale de ces équations. Le passage à cette formulation intégrale permet de déduire le champ électromagnétique en tout point du domaine extérieur Ω_{ext} à partir de la connaissance des champs totaux à la surface Γ de l'objet.

On pose $\overrightarrow{J}$ et $\overrightarrow{M}$ les traces tangentielles des champs électriques et magnétiques totaux sur Γ, aussi appelés courants électriques et magnétiques. Ces courants sont définis par :

$$\begin{cases} \overrightarrow{J} = \vec{n} \wedge \left(\overrightarrow{H^i} + \overrightarrow{H^d} \right), \\ \overrightarrow{M} = \left(\overrightarrow{E^i} + \overrightarrow{E^d} \right) \wedge \vec{n}. \end{cases} \tag{2.4}$$

Dans le cas d'un matériau conducteur, le courant magnétique $\overrightarrow{M}$ est nul. Le problème revient alors à déterminer le courant électrique $\overrightarrow{J}$. D'après le théorème de représentation intégrale, pour tout point x qui n'est pas sur la surface de l'objet Γ, les champs électriques et magnétiques sont pleinement déterminés à partir de $\overrightarrow{J}$ de la manière suivante :

$$\begin{cases} \overrightarrow{E^d}(x) = -i\omega\mu_0 \int_\Gamma G(x,y)\overrightarrow{J}(y)d\Gamma(y) + \frac{1}{i\omega\epsilon_0} grad_x \int_\Gamma G(x,y) div_\Gamma \overrightarrow{J}(y)d\Gamma(y), \\ \overrightarrow{H^d}(x) = rot_x \int_\Gamma G(x,y)\overrightarrow{J}(y)d\Gamma(y), \end{cases} \tag{2.5}$$

où G est la fonction de Green, qui prend en compte la condition de radiation à l'infini de manière exacte, et qui est définie par :

$$G(x,y) = \frac{e^{ik|x-y|}}{4\pi|x-y|}. \tag{2.6}$$

Pour déterminer le courant J, on obtient une équation en passant à la limite sur Γ sur l'expression du champ électrique dans le système (2.4) :

$$-\vec{n} \wedge \overrightarrow{E^i}(x) = \vec{n} \wedge \left(-i\omega\mu_0 \int_\Gamma G(x,y) \overrightarrow{J}(y) d\Gamma(y) + \frac{1}{i\omega\epsilon_0} grad_x \int_\Gamma G(x,y) div_\Gamma \overrightarrow{J}(y) d\Gamma(y) \right) \tag{2.7}$$

Dans la littérature, on parle d'équation EFIE (*Electric Field Integral Equation*) pour l'équation (2.7). De manière analogue, une formulation MFIE (*Magnetic Field Integral Equation*) peut être écrite :

$$\vec{n} \wedge \overrightarrow{H^i}(x) = \frac{1}{2}\overrightarrow{J} - \vec{n} \wedge rot_x \int_\Gamma G(x,y) \overrightarrow{J}(y) d\Gamma(y) \tag{2.8}$$

Enfin, il est à noter que des formulations mixtes existent (CFIE).

Avec ces formulations, le problème de départ sur le domaine 3D non borné se réduit à la résolution d'un problème sur la surface de l'objet, qui est 2D et borné. La principale difficulté de calcul au sens numérique réside maintenant dans la singularité du noyau de Green lorsque $x = y$ (2.6).

Pour résoudre ces équations, il reste à mettre en oeuvre une méthode numérique afin de traduire le problème réduit 2D surfacique continu en un problème discret permettant le calcul d'une solution approchée sur un calculateur. Le paragraphe suivant donne des éléments de mise en oeuvre de la méthode des éléments finis de frontière sur la formulation intégrale des équations de Maxwell.

2.1.3 Méthode des éléments finis de frontière

La méthode des éléments finis de frontière (ou *Boundary Element Method*, BEM) fait l'objet d'une littérature fournie dans le cadre de la discrétisation des équations intégrales pour les problèmes de SER.

Afin de mettre en oeuvre la BEM, une formulation variationnelle du problème 2D surfacique doit d'abord être écrite. Pour cela, on multiplie l'équation choisie (EFIE, MFIE ou CFIE) par une fonction test $\overrightarrow{J'}$ et on intègre sur la surface de l'objet Γ. Pour l'équation EFIE (2.7), cela donne :

$$\int_\Gamma \overrightarrow{E^i}.\overrightarrow{J'}(x) d\Gamma(x) = \int_\Gamma \int_\Gamma G(x,y) \left[i\omega\mu_0 \overrightarrow{J}(y).\overrightarrow{J'}(x) + \frac{1}{i\omega\epsilon_0} div_\Gamma \overrightarrow{J}(y).div_\Gamma \overrightarrow{J'}(x) \right] d\Gamma(y) d\Gamma(x). \tag{2.9}$$

On considère ensuite une triangulation $\mathcal{T}$ de la surface Γ. L'élément fini de Raviart-Thomas de degré 1 est ensuite utilisé dans le cas du code qui fait l'objet de cette thèse. Pour chaque arête A_i du maillage, on associe :

- une fonction de base $\overrightarrow{\phi_i}$ a support sur la réunion des deux triangles adjacents à l'arête,
- et un degré de liberté J_i, qui est le flux du courant à travers A_i :

$$J_i = \int_{A_i} \overrightarrow{J}.\overrightarrow{\nu_i} dl,$$

avec $\overrightarrow{\nu_i}$ la normale à l'arête A_i.

Le courant électrique est ensuite décomposé selon cette base :

$$\overrightarrow{J}(x) = \sum_{i=1}^{n} J_i \overrightarrow{\phi_i}(x), \forall x \in \Gamma, \tag{2.10}$$

avec n le nombre d'arêtes dans le maillage. En injectant cette décomposition dans la formulation variationnelle de l'équation EFIE (2.9) par exemple, on aboutit à l'assemblage d'un système linéaire dense complexe $AX = B$. Les termes de la matrice A, encore appelée matrice d'impédance, sont donnés par :

$$A_{ij} = \int_\Gamma \int_\Gamma G(x,y) \left[i\omega\mu_0 \overrightarrow{\phi_i}(y).\overrightarrow{\phi_j}(x) + \frac{1}{i\omega\epsilon_0} div_\Gamma \overrightarrow{\phi_i}(y). div_\Gamma \overrightarrow{\phi_j}(x) \right] d\Gamma(y) d\Gamma(x). \tag{2.11}$$

On remarque que, dans le cas de l'EFIE, la matrice est symétrique. D'autre part, on voit également que les termes de la matrice ne dépendent que de la pulsation (et donc que de la fréquence) de l'onde incidente.

Pour des raisons de précision numérique, le nombre de triangles en oeuvre dans la triangulation $\mathcal{T}$ de Γ dépend de la taille de l'objet et de la fréquence d'émission f du radar. En effet, pour disposer d'une discrétisation suffisamment fine pour capturer efficacement les phénomènes électromagnétiques, la taille des arêtes de chaque triangle doit être de l'ordre de $\lambda/10$, avec $\lambda = c/f$ la longueur d'onde, où c est la vitesse de la lumière dans le vide.

Ainsi, un objet plus grand et une fréquence plus haute impliquent un nombre de degrés de liberté plus élevé, et donc à l'assemblage d'un système linéaire $AX = B$ plus volumineux.

La résolution du système linéaire peut alors se faire par un solveur algébrique. Dans le cadre du code de simulation 3D surfacique pour le calcul de SER qui constitue la matière première de ce travail de thèse, seules des méthodes directes sont considérées. Le paragraphe suivant dresse un panorama des solveurs linéaires directs pour la méthode des éléments finis de frontière.

2.2 Solveurs linéaires directs pour la Méthode des éléments finis de frontière

Pour des raisons de robustesse numérique, et de par les pré-requis en termes de précision imposés par le contexte industriel dans lequel cette thèse s'inscrit, seules des méthodes directes sont utilisées dans le cadre du code BEM pour le calcul de SER. Néanmoins, les techniques de compression, mono-niveau ou hiérarchiques permettent de réduire fortement les coûts algorithmiques en calcul et en mémoire tout en préservant les pré-requis industriels. La section suivante décrit la structure et l'algorithmique d'un solveur direct dense classique. Elle est immédiatement suivie par une section donnant des éléments de mise en place de techniques de compression afin de réduire les coûts en mémoire et en calcul du solveur.

2.2.1 Solveur direct dense

L'emploi d'un solveur direct dense pour la résolution du problème linéaire $AX = B$ issu de la méthode des éléments finis de frontière est fréquemment

rencontré dans la communauté de la simulation pour les ondes radar. Cette stratégie de résolution permet en général d'atteindre plus facilement des précisions très fines, comme imposé par notre contexte industriel, tout en étant plus robuste face à une méthode itérative.

Les méthodes directes se décomposent classiquement en deux phases, avec une première phase de factorisation, encore appelée décomposition. Dans cette phase, la matrice globale A est transformée en un produit de deux matrices triangulaires $A = LU$, avec L une matrice triangulaire inférieure et U une matrice triangulaire supérieure.

L'Algorithme 1 décrit les étapes nécessaires pour la décomposition LU d'une matrice carrée A de taille $n \times n$ inversible, supposée de termes diagonaux non nuls pour plus de simplicité. Cet algorithme est séquentiel, et opère colonne par colonne sur la matrice A. Le contenu de la matrice A est remplacé par les facteurs LU en cours de factorisation, cette implémentation étant classiquement rencontrée pour des raisons d'économies en coûts mémoire.

Dans le cas particulier d'une matrice symétrique définie positive, une variante, appelée décomposition de Cholesky, permet de se ramener à $A = LL^T$.

Algorithme 1 : Décomposition LU séquentielle sans pivotage

Entrées : Une matrice carrée A inversible de taille $n \times n$ telle que $\{a_{ii} \neq 0, 1 \leq i \leq n\}$

Résultat : Le contenu de la matrice A est remplacé par les facteurs L pour $\{a_{ij} | i < j, (i,j) \in \{1,...,n\}^2\}$ et U pour $\{a_{ij} | i >= j, (i,j) \in \{1,...,n\}^2\}$

```
1  pour k de 1 à n faire
2  |  pour i de k+1 à n faire
3  |  |  a_ik = a_ik / a_kk
4  |  fin
5  |  pour i de k+1 à n faire
6  |  |  pour j de k+1 à n faire
7  |  |  |  a_ij = a_ij − a_ik a_kj
8  |  |  fin
9  |  fin
10 fin
```

Une fois que la factorisation est déterminée, une seconde phase, appelée phase de résolution ou phase de descente/remontée, permet de déterminer X en fonction de B en s'appuyant sur la forme triangulaire des facteurs qui décomposent la matrice A. En effet, si par exemple $AX = LL^TX = B$, et de par la forme triangulaire de L et donc de L^T, il est possible de déterminer l'inconnue X par la mise en œuvre de deux étapes successives opérant ligne par ligne : une première étape, appelée descente, pour calculer Y tel que $LY = B$; et une seconde étape, appelée remontée, pour calculer X tel que $L^TX = Y$.

Une fois que le calcul de X est effectué, la méthode des éléments finis de frontière permet de se ramener aux courants électriques et magnétiques à la surface de l'objet diffractant, et ainsi d'en déduire la SER pour un angle d'observation donné.

Du point de vue des coûts algorithmiques, pour une matrice carrée A de

taille $n \times n$, les coûts en calcul sont dominés par l'étape de factorisation, qui requiert $\mathcal{O}(n^2)$ éléments en stockage mémoire et $\mathcal{O}(n^3)$ calculs. En pratique, ces coûts deviennent assez vite un obstacle majeur concernant les possibilités de simulation, principalement en ce qui concerne la simulation d'objets plus grands à des fréquences plus élevées.

Un moyen pour résoudre les systèmes linéaires denses de manière plus efficace sur les architectures de calcul modernes consiste exprimer les algorithmes pour la décomposition et pour la résolution par blocs. Il est en effet ainsi possible d'employer des opérations sur des blocs plutôt que sur des éléments en se reposant sur l'utilisation des routines BLAS3 (*e.g.* GEMM) et LAPACK (*e.g.* POTRF), bien connues pour être plus performantes que des opérations sur des éléments isolés (voir la Section 4.1). L'Algorithme 2 décrit par exemple la décomposition LL^T séquentielle par bloc. Il est possible de dériver un algorithme similaire pour la descente remontée par bloc.

Néanmoins, malgré l'utilisation de routines de calcul plus efficaces, les coûts en calcul et en mémoire restent les mêmes, dominés par l'opération de multiplication matricielle BLAS3 GEMM.

Algorithme 2 : Décomposition LL^T séquentielle par bloc

Entrées : Une matrice carrée par bloc $A = (A_{ij})_{i \leq j}$ inversible
Résultat : Le contenu de la matrice par bloc $A = (A_{ij})_{i \leq j}$ est remplacé par les facteurs par bloc $L(A_{ij})_{i \leq j}$ pour $1 < i < nb$

pour *k de 1 à n* **faire**
  $A_{kk} = L_{kk}L_{kk}^T$: POTRF(A_{kk})
  pour *i de k+1 à n* **faire**
    $A_{ik} = A_{kk}^{-1}A_{ik} = (L_{kk}L_{kk}^T)^{-1}A_{ik}$: TRSM(A_{ik}, A_{kk})
  fin
  pour *i de k+1 à n* **faire**
    pour *j de k+1 à i-1* **faire**
      $A_{ij} = A_{ij} - A_{ik}A_{jk}^T$: GEMM(A_{ij}, A_{ik}, A_{jk}^T)
    fin
    $A_{ii} = A_{ii} - A_{ik}A_{ik}^T$: SYRK(A_{ii}, A_{ik})
  fin
fin

Afin de résoudre des systèmes linéaires plus grands en des temps raisonnables, une première méthode consiste à exploiter des machines de calcul de grande envergure, qui imposent le recours à l'algorithmique parallèle. La Section 3 dresse un panorama des outils matériels et logiciels dédiés au calcul haute performance. La Section 4.1 donne un état de l'art des techniques de programmation pour les solveurs directs denses.

Une alternative, supplémentaire à la parallélisation, consiste à mettre en oeuvre des techniques de compression algébrique. La section suivante donne des éléments de mise en place de cette alternative.

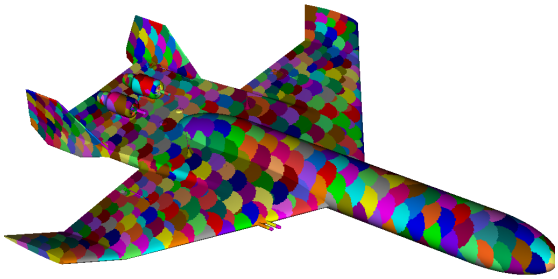


FIGURE 2.3 – Exemple de partitionnement d'un drone (ou UAV en anglais) par un algorithme de pavage pour la mise en œuvre d'une méthode de compression. Plus les sous-parties sont éloignées et plus les interactions entre celles-ci sont compressibles.

2.2.2 Solveur direct avec compression

Au devant des coûts en calcul et en mémoire élevés d'une approche dense directe, la mise en place de méthodes de compression algébrique apparaît intéressante afin de rendre accessible la simulation des problèmes d'ondes pour des maillages 3D surfaciques à des échelles plus élevées.

Sur le principe de la méthode des multipôles rapides (FMM [37]), les méthodes de compression algébrique reposent sur le principe que les interactions entre des sous-parties lointaines de l'objet sont de rang numérique r_k faible, où le rang numérique correspond au nombre de valeurs singulières supérieures à ϵv_0, avec $\epsilon > 0$ une précision numérique donnée, et v_0 la plus grande valeur singulière.

La mise en oeuvre une telle méthode de compression commence par un partitionnement de l'objet. La Figure 2.3 montre un exemple de partitionnement d'un drone par un algorithme de pavage, donnant lieu à une partition $\mathcal{P} = \{P_i, i \in \mathbf{N}\}$ de l'objet. Les inconnues de la matrice problème correspondante A, obtenue par application de la BEM, peuvent alors être re-numérotées conformément au partitionnement, une partie après l'autre. Ainsi, la matrice A se retrouve découpée en un ensemble de sous-blocs A_{ij}. Chaque bloc A_{ij} correspond à l'interaction entre les parties P_i et P_j de l'objet.

En ce qui concerne le caractère compressible ou non des blocs A_{ij}, un critère d'admissibilité permet de déterminer au préalable si les interactions représentées par le bloc sont de rang numérique faible. Ce critère est fonction de la distance entre les parties P_i et P_j ainsi que de leurs diamètres respectifs : un bloc A_{ij} est compressible dès lors que :

$$min(diam(P_i), diam(P_j)) < \eta \cdot d(P_i, P_j),$$

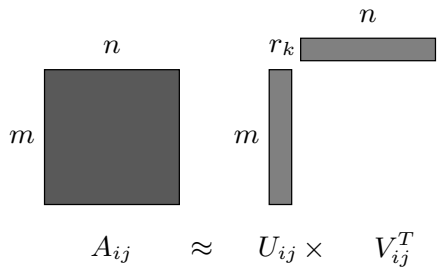


FIGURE 2.4 – Approximation de rang faible d'un bloc A_{ij} de taille $(m \times n)$ en un produit de deux matrices U_{ij} et V_{ij} de tailles $(m \times r_k)$ et $(n \times r_k)$, avec r_k le rang numérique de A_{ij}.

avec *diam* l'opérateur diamètre, d l'opérateur distance entre deux parties de l'objet, et η un paramètre réel positif donné [41, 66]. La vérification de ce critère d'admissibilité étant équivalente à une décroissance rapide des valeurs singulières du noyau de Green, il en résulte ainsi une interaction de rang numérique faible pour un seuil de précision fixé.

La compression peut alors se mettre en oeuvre sur un seul niveau, ou de manière hiérarchique. Les deux prochains paragraphes décrivent ces deux possibilités.

2.2.2.a La compression mono-niveau

Après partitionnement et application du critère de compressibilité, les blocs compressibles A_{ij} peuvent être approchés par le produit extérieur de deux matrices rectangulaires U et V^T de rang numérique r_k, tel que décrit par la Figure 2.4. On écrit alors $A_{ij} \approx U_{ij} \times V_{ij}^T$, où A_{ij} est de taille $m \times n$, U_{ij} de taille $m \times r_k$ et V_{ij} de taille $n \times r_k$. Le rang numérique r_k du bloc compressé est déterminé à partir d'une précision ϵ fixée. Plus précisément, r_k est le plus petit entier tel que $||A_{ij} - U_{ij} \times V_{ij}^T|| < \epsilon$.

S'il est possible de déterminer les facteurs U_{ij} et V_{ij}^T par le biais d'une décomposition aux valeurs singulières (SVD) tronquée sur A_{ij}, c'est à dire une SVD pour laquelle on néglige les valeurs singulières plus petites que ϵ et les vecteurs singuliers associés, l'ensemble des blocs A_{ij} qui composent la matrice peuvent être assemblés directement sous forme compressée par l'utilisation de l'algorithme Adaptive Cross Approximation (ACA) ou d'une de ses variantes (ACA+) [18]. Ceci permet de réduire considérablement les pré-requis en termes de mémoire et en calcul pour la phase d'assemblage de la matrice problème A, aboutissant à des matrices compressées telles qu'illustrées par la Figure 2.5.

L'emploi de méthodes de compression pour la résolution de systèmes linéaires par une méthode directe nécessite toutefois l'utilisation d'une algèbre linéaire particulière. En effet, les noyaux nécessaires à la factorisation LU ou LL^T par bloc (*e.g.* POTRF, TRSM ou GEMM, voir l'Algorithme 2) doivent être adaptés afin de pouvoir traiter le cas des blocs compressés.

Par exemple, en supposant que $A_{kk} = L_{kk}L_{kk}^T$ est de rang plein et que $A_{ik} = U_{ik}V_{ik}^T$ est compressé, l'opération TRSM(A_{ik}, A_{kk}) sur l'Algorithme 2 est adaptée pour tirer parti de la compression en calculant $V_{ik} = V_{ik}L_{kk}^{-T}$ en lieu et

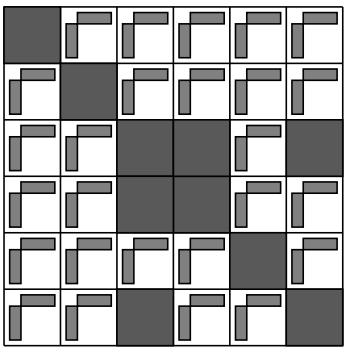


FIGURE 2.5 – Exemple de matrice assemblée et stockée au format compressé (mono-niveau, BLR), où chaque bloc est soit de rang plein, soit compressé sous forme UV^T. Les blocs diagonaux, qui représentent les interactions des sous-parties sur elles-mêmes, sont typiquement de rang plein.

place de $A_{ik} = A_{ik}L_{kk}^{-T}$. Il en résulte un coût opératoire moins élevé de l'ordre de $\mathcal{O}(r_k \times n^2)$ contre $\mathcal{O}(n^3)$, avec n la taille des blocs matriciels carrés A_{kk} et A_{ik}.

En outre, il est souvent nécessaire d'implémenter plusieurs versions de chaque noyau, selon le type des opérandes. Par exemple, pour un GEMM qui effectue $C = C - AB^T$, il y a $2^3 = 8$ possibilités selon la nature de A, B et C (compressible ou non), chacune donnant lieu à une implémentation différente.

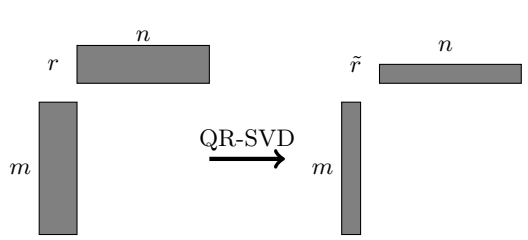
Ensuite, dans certains cas, l'introduction de la compression dans l'opération GEMM peut faire appel à un nouveau type d'opération : les opérations de recompression. Dans le cas d'une multiplication avec 3 matrices compressées, le GEMM devient :

$$U_{ij}V_{ij}^T - U_{jk}V_{jk}^T \left(U_{ik}V_{ik}^T\right)^T = (U_{ij}|U_{jk}) \left(V_{ij}|U_{ik}V_{ik}^T V_{jk}\right)^T,$$

où l'opérateur | désigne la juxtaposition de matrices, utilisée ici pour effectuer la soustraction de matrices de rang faible. Le rang numérique de la matrice résultat est alors la somme des rangs des matrices en entrée du GEMM : $r_{ij} = r_{ik} + r_{kj}$. En général, ce rang n'est pas optimal. De plus, les opérations de mise à jour GEMM pour le bloc A_{ij} étant susceptibles d'intervenir pour plusieurs étapes k d'une factorisation bloquée (voir l'Algorithme 2), ce rang numérique peut grandir au point de dépasser les coûts en mémoire et en calcul nécessaires pour le bloc A_{ij}. Il est donc nécessaire de faire intervenir une opération qui vise à limiter la croissance du rang numérique provenant de la juxtaposition des matrices compressées.

Afin de garder le contrôle sur le rang numérique suite aux additions de rang faible, il doit être réduit par un algorithme de recompression de type SVD tronquée, QR-SVD (voir la Figure 2.6) ou RRQR (Rank Revealing QR). Ces opérations faisant intervenir des noyaux LAPACK particulièrement coûteux (décompositions QR, SVD), elles deviennent le nouveau point chaud de la décomposition $A = LU$ ou $A = LL^T$, par opposition au GEMM dans le cas d'une matrice non compressée [1].

1. Pour plus de précisions, voir la section 1.1.5.1 page 17 de la référence [18]



$$
\begin{aligned}
AB^T &= QR\left(A\right)\left(QR\left(B\right)\right)^T \\
AB^T &= \left(Q_A R_A\right)\left(Q_B R_B\right)^T \\
AB^T &= Q_A R_A R_B^T Q_B^T \\
AB^T &= Q_A SVD\left(R_A R_B^T\right) Q_B^T \\
AB^T &= Q_A U \Sigma V^H Q_B^T \\
AB^T &\approx Q_A U \overline{\Sigma} V^H Q_B^T \\
AB^T &\approx \left(Q_A U \overline{\Sigma}\right)\left(Q_B V^*\right)^T
\end{aligned}
$$

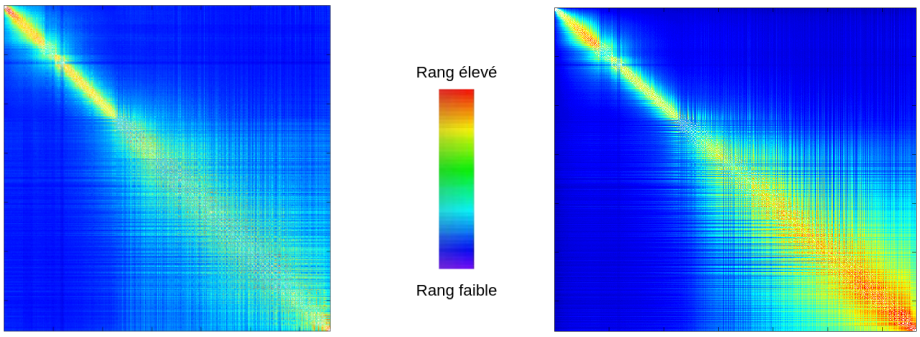
FIGURE 2.6 – Algorithme de recompression QR-SVD. Pour une précision donnée ϵ, cet algorithme recompresse un bloc sous forme AB^T avec un rang r en un bloc de rang numérique optimal $\tilde{r}$. Cette opération est implémentée par deux décompositions QR (GEQRF), une SVD (GESVD), et deux multiplications avec les matrices orthogonales obtenues par décomposition QR (UNMQR). La matrice diagonale Σ est tronquée de manière à garder les $\tilde{r}$ valeurs singulières supérieures à ϵ dans $\overline{\Sigma}$.

Enfin, il est notable que l'adoption de la compression telle qu'illustrée par la Figure 2.5 introduit du déséquilibre de charge en calcul et en mémoire entre les différents blocs qui composent la matrice problème. En effet, par opposition au cas sans compression, les blocs ne sont plus tous de la même taille, et ne sont plus tous de la même nature (*i.e.* compressés ou non).

De plus, les blocs compressés n'ont pas tous le même rang numérique. Pire encore, le rang numérique d'un bloc est susceptible d'évoluer en cours de factorisation de par les opérations de recompression qui interviennent dans le GEMM pour certaines configurations. La Figure 2.7 montre l'évolution du rang numérique des blocs d'une matrice compressée avec la méthode mono-niveau pour la simulation d'un lanceur du CNES de $1,8$ millions de degrés de liberté dont les détails sont donnés en Section 2.4.2. On constate que le rang numérique initial des blocs est variable (Figure 2.7a), et que les rangs après factorisation sont globalement plus élevés (Figure 2.7b).

Dans le cadre d'un algorithme de factorisation par blocs parallèles, où les blocs de la matrice seraient distribués, ceci pourrait conduire à un déséquilibre de charge en mémoire et en calcul important entre les unités de calcul, résultant en des exécutions parallèles de plus en plus inefficaces à mesure que le nombre de ressources en calcul augmente.

Les techniques de compression ci-avant présentées permettent de réduire drastiquement le nombre de ressources, les temps de calcul et la mémoire nécessaire pour la simulation de cas industriels en furtivité radar. Les complexités mesurées en pratique sont en $\mathcal{O}(n^{2,11})$ en calcul et $\mathcal{O}(n^{1,44})$ en mémoire, contre respectivement $\mathcal{O}(n^3)$ et $\mathcal{O}(n^2)$ sans compression. Par rapport à une méthode dense classique, elles permettent de simuler des cas plus grands en termes de nombre d'inconnues. Elles autorisent également la réalisation de balayages en fréquence et angulaires plus fins. Le paragraphe suivant montre qu'il est encore possible de réduire les coûts de simulation en considérant une approche hiérarchique pour la mise en place de la compression.



(a) Rangs numériques des blocs avant la factorisation

(b) Rangs numériques des blocs après la factorisation

FIGURE 2.7 – Rangs numériques des blocs matriciels d'une matrice compressée par bloc avec la méthode mono-niveau avant et après la factorisation. On constate qu'un phénomène de remplissage numérique se produit, qui se traduit par l'augmentation générale du rang des blocs matriciels.

2.2.2.b La compression hiérarchique

Une manière de diminuer encore les coûts en calcul et en mémoire du code BEM consiste en l'adoption d'une technique de compression hiérarchique, basée sur l'utilisation des $\mathcal{H}$-matrices [51]. Cette technique réutilise les différents ingrédients de base de la compression mono-niveau : partitionnement, critère de compressibilité, forme de rang faible, algèbre linéaire compressée. Elle étend cette base en munissant les blocs d'une structure hiérarchique, qui peut se représenter sous la forme d'un arbre quaternaire.

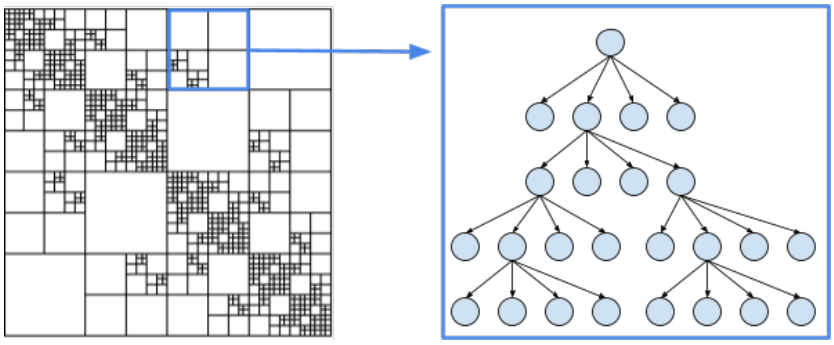


FIGURE 2.8 – Exemple de structure $\mathcal{H}$-matrice. Les blocs diagonaux, qui représentent les interactions des sous-parties de l'objet avec elles-mêmes, sont généralement moins compressés. La structure de chaque bloc $\mathcal{H}$-matrice peut être représentée par un arbre quaternaire.

Pour la mise en oeuvre de la compression hiérarchique, la matrice est à nouveau subdivisée en sous-blocs relativement à un partitionnement de l'objet. Ensuite, le critère de compressibilité est appliqué de manière récursive à l'intérieur de chaque bloc : si le bloc est compressible, il est assemblé sous forme compressée par la méthode ACA ou une de ses variantes. Sinon, le bloc est

$$\mathcal{H}\text{-POTRF}\left(\begin{array}{|c|c|}\hline \mathcal{H}_{11} & \\ \hline \mathcal{H}_{21} & \mathcal{H}_{22} \\ \hline \end{array}\right) = \left\{\begin{array}{l} \mathcal{H}\text{-POTRF}(\mathcal{H}_{11}) \\ \mathcal{H}\text{-TRSM}(\mathcal{H}_{11}, \mathcal{H}_{21}) \\ \mathcal{H}\text{-SYRK}(\mathcal{H}_{21}, \mathcal{H}_{22}) \\ \mathcal{H}\text{-POTRF}(\mathcal{H}_{22}) \end{array}\right.$$

FIGURE 2.9 – Opération POTRF hiérarchique : la factorisation symétrique hiérarchique sur l'ensemble du bloc ($\mathcal{H}$-POTRF, à gauche) peut se décomposer récursivement en quatre opérations hiérarchiques sur les sous-blocs de premier niveau. Cette décomposition récursive peut se poursuivre jusqu'à atteindre des noyaux d'algèbre linéaire de base (compressés ou denses).

divisé en 4 sous-blocs de tailles égales. Le procédé recommence sur chacun des sous-blocs jusqu'à atteindre une taille de bloc fixée, pour laquelle le bloc est assemblé sous forme dense.

La Figure 2.8 donne un exemple de $\mathcal{H}$-matrice par bloc (à gauche). Elle est composée de 4×4 blocs hiérarchiquement compressés. La structure de chaque bloc peut se représenter avec un arbre quaternaire (ou *quadtree* en anglais), tel que l'arbre qui décrit le bloc à l'intersection de la ligne 1 et de la colonne 3 de la matrice (à droite). Typiquement, les blocs diagonaux sont moins bien compressés sur cet exemple, ces blocs représentant à nouveau les interactions des sous-parties de l'objet sur elles mêmes.

Du point de vue des opérations algébriques pour la factorisation de matrices, l'algèbre linéaire compressée doit être enrichie afin de pouvoir traiter des blocs hiérarchiques. Concrètement, afin de gérer des $\mathcal{H}$-matrices, les noyaux de calcul sont rappelés récursivement sur les sous-blocs, jusqu'à se ramener à un cas de base (opération compressée ou dense, voir la Figure 2.9). Il est notable que, pour la multiplication de matrices (GEMM), il s'agit à présent d'implémenter et de maintenir $3^3 = 27$ versions du noyau, selon le type des opérandes (bloc dense, compressé ou $\mathcal{H}$-matrice).

En plus des noyaux de recompression, des opérations de subdivision ou d'agglomération de matrices de rang faible sont parfois nécessaires afin d'opérer sur des blocs de taille compatible. Si l'opération de subdivision est assez simple à mettre en oeuvre, l'opération d'agglomération nécessite l'utilisation des mêmes noyaux LAPACK que pour l'opération de compression, ce qui en fait une opération particulièrement coûteuse [2].

Toutes les opérations nécessaires pour la factorisation de matrice par bloc peuvent alors être définies, ce qui permet de garder l'Algorithme 2 pour la factorisation LL^T par exemple. Les coûts algorithmiques se voient diminuer encore, pour passer à $\mathcal{O}(n \log(n))$ en temps et en calcul contre respectivement $\mathcal{O}(n^{2,11})$ et $\mathcal{O}(n^{1,44})$ pour la compression mono-niveau. Ainsi, des simulations avec un nombre de degrés de liberté encore plus élevé et des études paramétriques de plus grandes envergures peuvent être menées en des temps raisonnables, compatibles avec une démarche de production industrielle.

Néanmoins, l'ajout de la notion de hiérarchie augmente considérablement la complexité du point de vue de la programmation d'une telle méthode de résolution directe pour résoudre $AX = B$, tant du point de vue séquentiel,

2. Pour plus de détail, voir la section 1.6.6, pages 18 et 19 de la référence [18]

de par le nombre de noyaux de calcul à implémenter, que du point de vue parallèle, de par la complexité des structures de données qui entrent en jeu. Une présentation de l'implémentation parallèle d'un tel solveur est donnée dans la Partie 4.3.

De plus, le caractère hiérarchique apporte encore de la variabilité en termes de charge en mémoire et en calcul d'un bloc à l'autre. Tout ceci rend encore plus difficile la tâche de distribuer équitablement les blocs matriciels sur plusieurs entités de calcul en vue d'une exécution parallèle efficace sur de grands calculateurs, tâche que cette thèse se propose d'effectuer.

2.3 Un code industriel 3D surfacique pour le calcul de SER

Tous les ingrédients présentés dans ce chapitre contribuent à l'élaboration d'un code 3D surfacique pour la résolution des équations de Maxwell en vue du calcul de la SER d'objets diffractants. De par les éléments qu'il contient, le code se décompose classiquement en 4 parties :

1. La phase d'assemblage : elle assemble la matrice problème A et le second membre B par la méthode des éléments finis de frontière. Selon le solveur mis en oeuvre, la matrice problème est assemblée intégralement ou directement sous forme compressée.
2. La phase de décomposition ou de factorisation : la matrice A est décomposée en facteurs triangulaires LU ou LL^T, selon son caractère symétrique ou non. Cette phase est classiquement la plus coûteuse et représente généralement le facteur limitant en termes de simulation accessible.
3. La phase de descente/remontée ou de résolution : cette phase calcule $X = A^{-1}B$ en s'appuyant sur la décomposition en facteur triangulaires précédemment obtenue. Cette phase peut représenter une fraction significative des coûts en calcul, selon le nombre d'angles d'incidence désirés pour la simulation.
4. La phase d'exploitation des résultats : les courants surfaciques et/ou la SER pour chaque angle d'incidence et d'observation sont déterminés à partir de la solution X précédemment calculée.

Le code ainsi produit met également en jeu tout un éventail d'éléments physiques et numériques complexes (fils, jonctions, matériaux, impédance, ...), permettant la résolution de cas riches d'un point de vue de la physique des ondes électromagnétiques. Plusieurs scénarii de balayage sont également disponibles : balayages fréquentiels et/ou balayages angulaires. Les calculs de SER peuvent se faire de manière monostatique, c'est à dire que l'observateur et l'émetteur sont confondus, ou de manière bistatique, où l'observateur et l'émetteur sont deux entités différentes.

Depuis les années 90, ce code tire parti des supercalculateurs du CEA/DAM, par la mise en oeuvre d'algorithmes parallèles dans chacune des phases qui le composent. Historiquement, seul le solveur direct dense était disponible pour la résolution du système linéaire. L'introduction des méthodes de compression a débuté dans les années 2010. Elles ont été introduites sur la base du solveur direct dense parallèle existant.

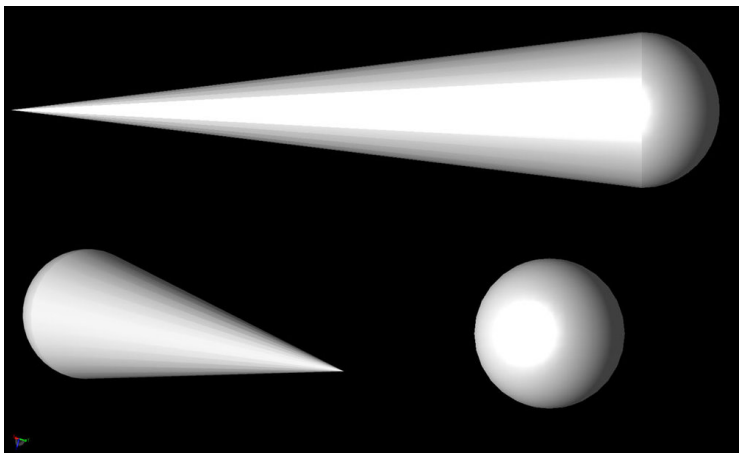


FIGURE 2.10 – Géométrie de cône-sphère [76]

Le prochain chapitre discute des différents outils matériels et logiciels pour le calcul haute performance. Certains de ces outils interviennent dans la parallélisation du code de simulation autour duquel cette thèse gravite. Plus de détails seront donnés concernant la parallélisation de ce code, et plus particulièrement du solveur pour l'algèbre linéaire sans et avec compression, dans la Section 4.1.

2.4 Exemples de simulations

Nous donnons ici une rapide description de quelques cas tests qui ont servi à valider nos développements logiciels, et d'en mesurer les performances.

Certains cas d'études sont notamment issus du workshop ISAE, dont l'objectif est de proposer aux différents industriels du domaine des cas tests communs afin de valider et de comparer leurs codes respectifs.

2.4.1 Cônes sphères IEEE

La géométrie dite de "cône sphère IEEE" [82] est une géométrie de référence dont les caractéristiques sont bien connues, et qui sert de référence dans la communauté de l'électromagnétisme non seulement pour les simulations, mais également pour les mesures radars [90]. Bien que simple, ce problème peut s'avérer délicat à traiter de manière précise, ce qui constitue donc un bon cas de validation pour notre code. Il s'agit d'un objet de 60 cm de long dont la partie avant est un cône de révolution, et dont l'arrière est une demi-sphère, tel qu'illustré dans la Figure 2.10. Nous considérons ici un objet parfaitement métallique.

Il est donc simple de générer automatiquement cette géométrie et de la mailler avant de la simuler avec notre code. Pour obtenir différentes tailles de problèmes, nous avons considéré un éclairement sous différentes fréquences, ce qui amène à des discrétisations plus ou moins fines, et à un nombre d'inconnues d'autant plus important que la longueur d'onde est faible. Le cas que nous considérons dans ce document correspond à un éclairement par une onde avec une fréquence à 18 GHz, et nécessite de résoudre un problème avec 103000 inconnues.

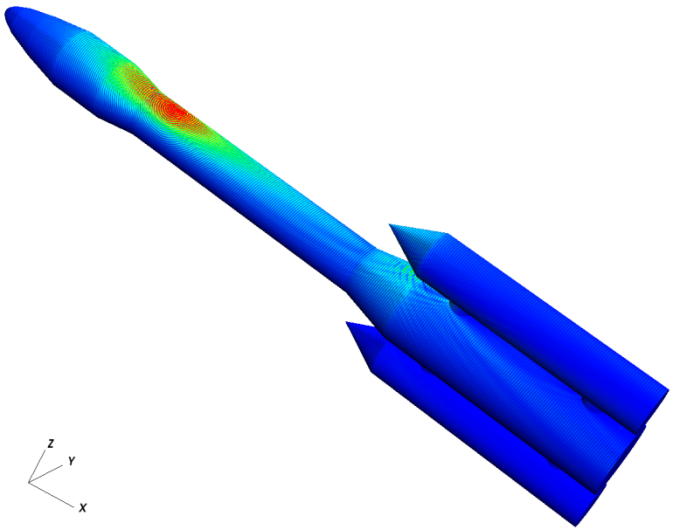


FIGURE 2.11 – Modules des courants électriques induits par une antenne à la surface du lanceur CNES

2.4.2 Lanceur CNES (Workshop ISAE)

Le cas "lanceur CNES" est un cas du Workshop ISAE 2014. Il s'agit de calculer le gain d'une antenne située sur une géométrie de lanceur. Ce résultat est visible dans la Figure 2.11.

Il s'agit d'un cas test qui comporte 1650875 inconnues, ce qui aurait nécessité 300000 heures de calcul cumulées sur 15000 coeurs de la machine Tera100. Ce cas test est donc un des problèmes pour lesquels l'utilisation de techniques de compression s'est avérée essentielle. La résolution de ce problème à l'aide d'un solveur BLR n'a en effet nécessité que 2500 heures de calcul cumulées, et un solveur hiérarchique à peine 1000 heures.

2.4.3 Cas test UAV (Workshop ISAE)

Le cas test "UAV" du workshop ISAE consiste à calculer les courants induits par un radar à la surface d'un drone (ou UAV en anglais). Il s'agit d'un cas dont la géométrie à plusieurs échelles peut poser des problèmes aux techniques de compression, et aux algorithmes de partitionnement sous-jacents.

Plusieurs versions de ce problème ont étés proposées lors des éditions 2014 et 2016 (avec une trappe constituée de matériaux absorbants), mais la Figure 2.12 représente le cas de 2016, pour lequel il est nécessaire de résoudre un problème avec 800000 inconnues.

2.5 Calculateurs utilisés pour les simulations

Les cas présentés dans ce document ont été effectués sur différentes machines. Celles-ci sont toutes des calculateurs composés de plusieurs nœuds de calculs reliés entre eux par un réseau haute performance.

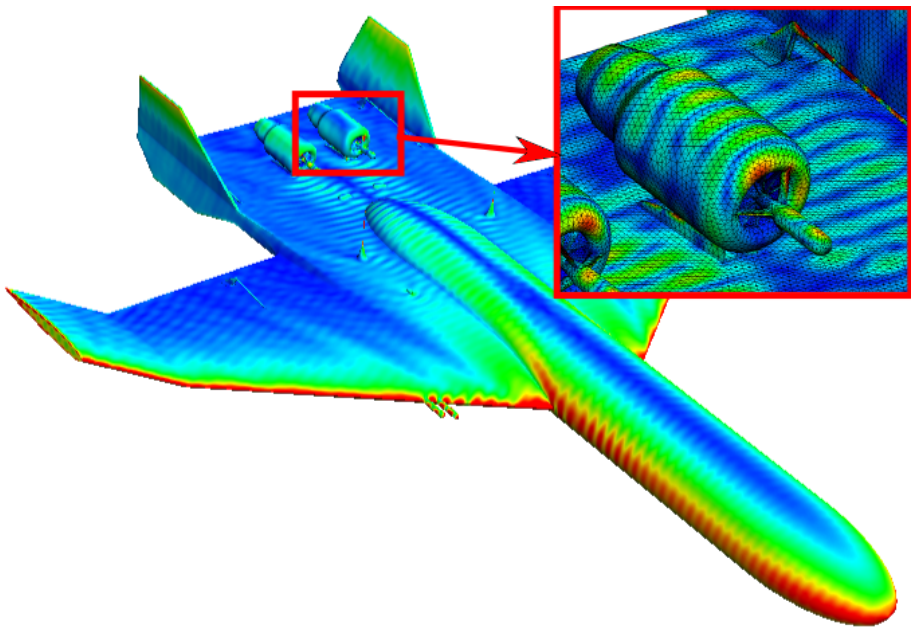


FIGURE 2.12 – Modules des courants électriques à la surface d'un drone (UAV)

Le premier est le calculateur TERA1000-1, classé 318^{eme} au classement du top500 en octobre 2021 [1] et 45^{eme} lors de son entrée au classement en juin 2016. Ce calculateur est une machine de classe *pétaflopique*, et possède plus de 70 000 cœurs de calcul. Parmi les ressources disponibles au sein des différentes partitions de la machine, on retrouve des processeurs classiques et des cartes graphiques. La partition sur laquelle les tests ont été effectués possède des nœuds ayant chacun deux processeurs Haswell de 16 cœurs multithreadés et cadencés à 2,3 GHz. Ils possèdent le support des instructions SSE et AVX2. 128 Go de mémoire vive sont installés par nœud, et un réseau InfiniBand FDR les relie.

Nous avons également utilisé la machine TERA1000-2, qui est classée 34^{eme} au classement du top500 en octobre 2021 et était classée 14^{eme} lors de son entrée au classement en juin 2018. Ce calculateur est également une machine de classe pétaflopique, et possède quant à lui plus de 550 000 cœurs de calculs. Les tests présentés ont été exécutés sur des nœuds possédant un processeur Xeon Phi ayant 68 cœurs multithreadés quatre fois chacun et cadencés à 1,4 GHz. Ils possèdent le support des instructions SSE, AVX2 et AVX512. Les différents noeuds sont interconnectés par un réseau Atos BXI [42]. On peut noter que l'implémentation de MPI au dessus du réseau BXI actuellement à notre disposition (basée sur la bibliothèque OpenMPI) ne permet pas d'utiliser les primitives basées sur du RDMA accessibles dans le matériel.

Enfin, nous avons utilisé le calculateur INTI, qui est une plateforme pour évaluer les différentes technologies de calculateurs. Il s'agit d'une machine à plus petite échelle que les machines du programme TERA. La partition sur laquelle nous avons effectué nos calculs est une partition composée de 1440 cœurs. Chacun des 32 noeuds est constitué de 2 processeurs de la génération Skylake, et possèdent 24 cœurs multithreadés 2 fois chacun. Enfin, les différents nœuds de

calculs sont reliés par un réseau Infiniband.

Chapitre 3

Outils pour le calcul haute performance

Dans ce chapitre nous allons présenter les différentes techniques utilisées dans le domaine du calcul haute performance. En effet, l'utilisation de modèles physiques et numériques toujours plus élaborés accroissent constamment les besoins en capacité de calcul. Pour résoudre ce dernier, il est nécessaire de disposer d'une part de machines adaptées, et d'autre part de méthodes de programmation permettant de tirer parti de ces machines dont la complexité ne cesse de croître.

Nous allons donc d'abord présenter les architectures des calculateurs modernes, puis nous évoquerons les différents principes de programmation permettant d'exploiter ces calculateurs. Nous rappellerons quels sont les défis à relever pour concevoir des environnements de programmation capables d'exploiter des machines complexes à grande échelle tout en offrant une programmabilité satisfaisante.

3.1 Architectures des supercalculateurs contemporains

Les besoins toujours croissant en puissance de calcul ont conduit à démultiplier la taille des calculateurs utilisés, passant d'une puissance de quelques mégaflops dans les années 1960 à plusieurs centaines de pétaflops en 2021.

Pour cela, la première méthode employée a consisté à faire augmenter la fréquence des processeur. En effet, pour une même architecture, doubler la fréquence revient à doubler les performance du calculateur, sans nécessiter d'adaptation pour les programmes exécutés. Cette technique a conduit à utiliser des fréquences toujous plus élevées, atteignant plusieurs GHz. Par exemple nous pouvons citer le pentium 4 qui a été conçu pour fonctionner à 5GHz.

Cependant, la consommation électrique, et donc la production de chaleur est proportionnelle au carré de la fréquence : en doublant la fréquence de fonctionnement, la consommation d'énergie est quadruplée.

La relation quadratique entre fréquence d'horloge et consommation électrique constitue donc un frein majeur. Fort heureusement, l'évolution des procédés de gravure des processeurs s'accompagne d'une amélioration de la taille des transistors, et donc d'une augmentation continue de la densité de transistors pour une surface donnée. La fameuse loi de Moore stipule ainsi que le nombre de transistors double tous les 18 mois.

Disposer de plus de transistors pour une même surface donne un avantage majeur puisqu'il est désormais possible de dupliquer certaines ressources au sein d'une même puce, comme les unités de calculs flottants, ou même l'intégralité du cœur de calcul

Contrairement à l'augmentation de la fréquence qui s'accompagne d'une évolution quadratique de la consommation électrique, le fait de doubler le nombre de transistors pour doubler la puissance de calcul ne fait que doubler la consommation électrique.

Cette relation linéaire entre la quantité de transistors et l'énergie consommée offre donc une stratégie complémentaire avec une évolution de la fréquence de fonctionnement. Toutefois, cette approche n'est pas transparente pour le programmeur puisque contrairement à la fréquence qui n'affecte en rien les modèles de programmation, cette stratégie nécessite d'introduire du parallélisme au sein

des applications.

Cette stratégie a également permis d'interrompre l'augmentation de la fréquence des processus, et même de la réduire de manière significatives depuis le milieu des années 2000.

Dans cette partie, nous allons évoquer les différentes catégories de parallélismes présents dans des supercalculateurs contemporains. Dans la Partie 3.1.1 nous présenterons la notion de calculateurs vectoriels, ainsi que l'utilisation des réseaux de communication. Dans la Partie 3.1.2 nous présenterons le parallélisme multicœur. Enfin, dans la Partie 3.1.3, nous présenterons l'utilisation des accélérateurs de calculs.

3.1.1 Des supercalculateurs vectoriels aux grappes de PC

Les premiers calculateurs étaient composés d'une seule machine. cependant, nous avons vu qu'il n'est pas possible d'augmenter indéfiniment la fréquence des processeurs pour augmenter le nombre de calculs traités par unité de temps, et qu'il est vite préférable d'introduire du parallélisme pour exploiter un plus grand nombre de transistor simultanément.

Une première méthode pour paralléliser les calcul consiste à exploiter un parallélisme de type SIMD, pour Single Instruction Multiple Data, soit instruction seule, données multiples en français. Ce parallélisme exploite le fait que certains calculs soient composés d'une suite rapproché des mêmes instructions, sur des données différentes et applique donc une même instructions sur un vecteur de données, d'où le nom de calcul vectoriel parfois donné.

Cependant, ce parallélisme au sein d'une même machine atteint vite ses limites. Pour continuer d'augmenter le parallélisme, et donc les capacité de calcul des machines, une solution naturelle est de découper les calculs en plusieurs parties indépendantes, et d'exécuter chacune de ces parties sur un calculateur différent quand cela est possible.

Il est alors naturel de relier ces calculateurs par un réseau, ce qui permet aux différentes parties du calcul de communiquer au cour de leur exécution, et donc de pouvoir exécuter des calculs moins indépendants.

Ces communications nécessitent, pour être effectuées de manière efficace, que leur latence soit le plus faible possible, et que leur débit soit le plus élevé possible. Cette contrainte sur la latence impose aux calculateurs de se situer physiquement proche les uns les autres, à cause de la latence intrinsèque de tout système de communications.

De plus, des développements constants ont lieux pour augmenter le débit des communications. En effet, les capacité des nœuds de calculs augmentant continuellement, il est nécessaire que la vitesse des communications augmente elle aussi, pour ne pas devenir un point de plus en plus limitant dans le parallélisme des calculs.

Enfin, nous pouvons noter une évolution dans les méthodes de communications entre les différents nœuds de calculs : si dans un premier temps celles-ci prenaient la forme de communications point à point, impliquant un émetteur et un récepteur, il est aujourd'hui possible des communications de type RDMA.

Nous présenterons l'utilisation de cette dernière forme dans la Partie 3.4.3.

3.1.2 L'avènement du multicœur

Nous allons maintenant présenter l'utilisation de plusieurs cœurs au sein d'un même nœud de calcul. Ces cœurs sont semblables à un processeur à part entière, mais plusieurs d'entre eux peuvent être disposés sur la même puce. Dans la Partie 3.1.2.a nous présenterons les processeur multicœurs symétriques. Puis, dans la Partie 3.1.2.b nous évoquerons les processeurs multicœurs non symétriques. Enfin dans la Partie 3.1.2.c nous parlerons de la combinaison de cette méthodes avec les réseaux présentés précédemment.

3.1.2.a Architecture multicœur symétrique

Le premier modèle mémoire historiquement apparu est le modèle SMP, pour Symmetric MultiProcessing. Ce modèle permet aux différentes unités de calcul d'accéder à la mémoire sans que l'une d'entre elles soit privilégiée. Les différents cœurs des processeurs ont alors les mêmes temps d'accès à la mémoire vive quelque soit leur position sur le calculateur.

Puis, avec l'amélioration constante des machines de calcul, nous pouvons observer une augmentation de la vitesse de calcul, ainsi qu'une réduction des temps d'accès à la mémoire vive. Cependant, ces évolutions ne se sont pas effectuées à la même vitesse, ralentissant les accès mémoire en comparaison des vitesses de calcul.

Pour réduire cette différence, des systèmes de caches ont été mis en place. Ceux-ci consistent en l'ajout d'une mémoire de taille plus petite, mais plus rapide, et permettant des débits de données plus élevés et une latence plus faible que la mémoire principale.

Il est communément observé que les données ayant la probabilité la plus élevée de subir un accès en lecture ou écriture sont les données ayant déjà subi récemment un accès mémoire. Il a donc été mis en place des systèmes automatiques plaçant automatiquement les données manipulées dans le cache pour accélérer les accès futurs, ce qui permet de rendre l'utilisation de ces caches par les différents algorithmes transparent.

Les caches ayant une taille limitée, il est alors nécessaire d'automatiser également l'éviction des données, ce qui permet par la suite d'inscrire d'autres variables dans le cache. Il existe plusieurs algorithmes différents pour cette gestion de cache, chacun ayant des avantages et des inconvénients différents. Par exemple, nous pouvons citer à correspondance préétablie qui requiert peu de logique mais dont les données peuvent être évincées alors même qu'elles sont plus récentes que d'autres variables, et le cache pleinement associatif qui requiert des algorithmes plus complexes, mais où les données sorties du cache sont forcément les plus anciennes. De nos jours les caches utilisés sont souvent de type dit N-way, qui peut être vu comme une solution hybride permettant de limiter les défauts des deux solutions.

Enfin, pour accélérer les accès à ces caches, il est possible de mettre en œuvre plusieurs niveaux de caches, de plus en plus petits mais de plus en plus rapides. Les processeurs modernes possèdent trois niveaux de ces caches nommés respectivement L1, L2 et L3 du plus petit et rapide au plus grand et lent.

3.1.2.b Architecture multicœur non symétrique

Cette architecture à base de caches permet de réduire grandement la latence moyenne des accès mémoire, cependant, deux limitations principales peuvent être notées : premièrement il est complexe de permettre à plusieurs cœurs d'accéder à la même mémoire simultanément. Deuxièmement, la taille des caches est limitée, de par la complexité de créer des mémoires rapides de grande taille.

La multiplication des cœurs va donc induire une limitation de la quantité de mémoire cache disponible par cœur. Pour réduire ce problème il est possible d'utiliser plusieurs caches et de les assigner à des cœurs différents, ou à des groupes de cœurs.

Il est également possible de multiplier seulement les plus petits niveaux de cache, en faisant partager les plus grands niveaux sur plusieurs cœurs. En pratique, c'est cette architecture qui a été sélectionnée dans les processus modernes, ce qui permet de limiter le partage des mémoires les plus rapides, tout en mutualisant les ressources moins critiques.

Dans ce cas, si une donnée a été migrée sur le cache d'un cœur, elle sera rapidement accessible par celui-ci, comme dans le cas précédent. En revanche, les autres cœurs n'y ayant pas accès, ils ne pourront pas profiter de l'accélération du cache, et devront donc faire un accès à la première mémoire en commun. Celle-ci étant forcément plus lente, nous pouvons observer que l'accès sera alors lui aussi plus lent que pour le premier cœur.

Cette différence dans les temps d'accès rend les accès mémoire non uniformes, suivant le cœur ayant effectué l'accès. Ce type de mémoire est donc nommé NUMA, pour *Non Uniform Memory Access*.

Cependant, s'il est possible que plusieurs threads copient une même donnée dans leurs caches respectifs, il n'est pas souhaitable qu'ils la conservent tous une fois cette donnée modifiée, pour permettre de garder une mémoire cohérente entre les différents threads. Il existe plusieurs protocoles de cohérence de cache. Sans rentrer dans le détail, la majorité de ces protocoles se basent sur une machine à état. Ces états sont gérés en interne par le système de cohérence des caches et permettent d'invalider les caches des autres threads en cas de modification de données.

Nous noterons qu'il est cependant nécessaire de prendre en compte cette différence dans les accès, car celle-ci peut grandement modifier les comportements des différents algorithmes. Par exemple, un thread effectuant des écritures sur des données peut ralentir d'autres threads effectuant des écritures sur ces mêmes données, en particulier si ces données étaient auparavant situées dans le cache de ces autres threads.

Enfin, nous pouvons noter qu'il existe également des système possédant plusieurs nœuds mémoire, chacun étant constitués d'une partie de la mémoire et des cœurs de calculs. Dans ces systèmes, les cœurs d'un nœud peuvent toujours accéder à la mémoire d'un autre, au prix de temps d'accès augmentés. Cette augmentation va, à nouveau, induire une différence dans les temps d'accès mémoire des différents cœurs, ce qui peut ralentir significativement le calcul.

3.1.2.c MPI+X

Dans la partie 3.1.1, nous avons vu qu'une des méthodes de parallélisme est de mettre en réseau plusieurs unités de calcul différentes. Dans les Parties 3.1.2.a

et 3.1.2.b nous avons vu une autre méthode de parallélisme qui consiste quant à elle à mettre en commun la mémoire entre plusieurs unités de calcul.

Si ces deux méthodes peuvent sembler s'opposer, il peut en fait être intéressant de les utiliser de manière complémentaire. En effet, les méthodes de parallélisme en mémoire partagée sont complexes à faire passer à l'échelle. À l'inverse, la mise en réseau des différents processus possède une limite bien plus haute, mais subit une latence plus importante.

Dans la Partie 3.5, nous verrons plus en détail l'implication du point de vue de la programmation d'un tel modèle hybride.

3.1.3 Les accélérateurs de calculs

Les CPUs, pour *Central Processing Unit* servent à exécuter les calculs d'un nœud de calcul. Ce sont des unités de calcul généralistes, dans le sens où ils ne sont pas spécialisés pour un type de calcul en particulier.

En revanche, pour un certain nombre d'applications, il est utile de disposer de cartes d'extension, pour pouvoir exécuter des opérations spécifiques, comme les entrées-sorties, ou certains types de calculs précis, ce qui permet d'optimiser les cartes pour ces calculs.

Dans cette partie nous présenterons deux types d'accélérateurs de calcul : les GPUs dans la Partie 3.1.3.a, et les KNLs dans la Partie 3.1.3.b.

3.1.3.a Cartes graphiques (GPUs)

Les GPUs, pour *Graphics Processing Unit* en anglais, sont des cartes d'extension permettant originellement de gérer l'affichage des ordinateurs, en 2D puis en 3D de nos jours.

Cet affichage est coûteux en calcul, et nécessite donc un processeur dédié. Cependant, si les calculs sont coûteux ils sont organisés d'une manière bien précise. En effet, les calculs nécessaires sont d'une part très fortement parallèles, et sont constitués d'autre part d'un grand nombre de fois le même calcul sur des données différentes.

Si les GPUs ont été optimisés pour l'affichage, il est aujourd'hui possible de les utiliser pour effectuer des calculs arbitraires. Cependant, ces cartes étant optimisées pour effectuer simultanément un grand nombre de fois le même calcul, il n'est pas efficace de les utiliser dans un contexte ne permettant pas ce type de traitement.

Cependant, même si cette contrainte les rend complexes à programmer, ces cartes possèdent une grande puissance de calcul qui peut être exploitée par certains types de calcul.

3.1.3.b Intel KNL

L'architecture Xeon phi est une architecture moderne de processus se basant sur l'architecture x86_64, et qui utilise le même jeu d'instructions, permettant d'exécuter nativement des programmes compilés pour des nœuds de calculs utilisant des processeurs x86_64.

Cette architecture a été conçue pour offrir un environnement fortement parallèle : elle propose d'utiliser des vecteurs de 512 bits, en plus d'un grand nombre de cœurs de calculs qui sont chacun multithreadés 4 fois.

Les premiers Xeon phi étaient sous la forme d'une carte d'extension, qui devait donc être utilisée en même temps qu'un autre processeur. Cependant, la gamme de Xeon phi knight landing, couramment abrégée en KNL, existe sous deux formes différentes : premièrement comme une carte d'extension, comme les gammes précédentes, et deuxièmement comme processeur dédié, ce qui permet de créer des machines composées uniquement de KNL.

Dans cette configuration, on peut considérer qu'il ne s'agit pas d'un accélérateur à proprement parler, contrairement aux architectures de type GPUs présentées précedemment. Ce sont en effet des processeurs qui se veulent généralistes, et sur lequel tourne un système d'exploitation *classique*, plutôt qu'une ressource de calcul sur laquelle on déporte du calcul. Cela signifie que les modèles de programmation sont les mêmes que sur les machines classiques, si ce n'est qu'il faut supporter un grand nombre de coeurs par noeud, et des coeurs aux performances séquentielles limitées.

3.2 Langages et outils pour l'exploitation des architectures à mémoire partagée

Plusieurs modèles de programmation ont émergé au fil du temps pour proposer des interfaces de programmation flexibles pour le programmeur tout en s'efforçant de rester proches des capacités du matériel, tout en offrant si possible une bonne programmabilité.

3.2.1 Programmation orientée "threads"

La notion de processus est apparue dans les années 60, avec la conception des systèmes multiprogrammés, c'est-à-dire capables d'exécuter plusieurs programmes de manière concurrente. Un processus est une instance de programme en cours d'exécution gérée par le noyau du système d'exploitation qui contrôle son ordonnancement et, plus généralement, sa consommation des ressources (CPU, mémoire, I/O). Sur une machine multiprocesseur, le système d'exploitation peut placer plusieurs processus actifs sur des processeurs différents, leur permettant ainsi de s'exécuter réellement en parallèle. Les processus constituent donc un moyen naturel d'exploiter les architectures multiprocesseurs, à condition d'utiliser en sus une interface de communication inter-processus. Cependant, l'utilisation de primitives de communication explicites alourdit considérablement un programme parallèle, surtout lorsque l'on sait que l'architecture sous-jacente offre une mémoire physique partagée par tous les processeurs. Il est donc naturel que l'on ait cherché à développer des paradigmes permettant d'exploiter cette propriété au niveau des programmes parallèles.

3.2.1.a Pthreads

À la fin des années 50 naît le concept de *coroutine* [38], destiné à faciliter la programmation de traitements naturellement concurrents [40] . Une coroutine est une routine dont l'exécution peut être temporairement suspendue pour reprendre l'exécution d'une autre coroutine. Cela instaure *de facto* un ordonnancement collaboratif explicite entre différents flots d'exécution. La mise en

œuvre d'un tel mécanisme requiert l'allocation d'une pile d'exécution séparée pour chaque coroutine.

Les coroutines sont d'une certaine façon l'ancêtre des processus légers, ou *threads*. Ces derniers disposent toujours de leur propre pile, mais le transfert de l'exécution d'un thread à l'autre (on parle de *changement de contexte*) est désormais implicite et c'est un ordonnanceur qui choisit le prochain thread qui s'exécutera sur le processeur.

Signalons que l'implémentation des threads peut être réalisée au niveau utilisateur, sous la forme d'une bibliothèque (e.g. Les *Green Threads* de Java), auquel cas le système d'exploitation ignore tout de l'existence éventuelle de threads au sein d'un processus. On parle alors de modèle N :1 (*user-level threading*). L'utilisation de threads ne permet pas, dans ce cas de figure, d'exploiter plusieurs processeurs : les threads sont simplement ordonnancés à tour de rôle de manière séquentielle.

À l'opposé, le support des threads au sein du noyau est arrivé assez tardivement dans les systèmes d'exploitation, et c'est le système Unix SunOS 5.0 (Sun Microsystems) qui, au début des années 90, a probablement proposé une des versions les plus abouties du multithreading. L'implémentation de Sun se payait même le luxe d'être hybride, en s'appuyant sur un support noyau et une bibliothèque de niveau utilisateur pour multiplexer l'ordonnancement d'un grand nombre de threads utilisateurs au-dessus d'un pool raisonnable de threads noyaux. On parle alors de modèle M :N (*hybrid threading*).

Il a fallu attendre 1996 pour qu'une première implémentation de threads implantés au niveau noyau Linux (modèle 1 :1, aka *kernel-level threading*) soit disponible : LinuxThreads [64]. Il s'agissait d'une implémentation incomplète de la norme *POSIX Threads* qui s'appuyait sur l'appel système `Clone` (disponible à partir du noyau 2.0) permettant de créer une nouvelle activité noyau sans forcément allouer un nouvel espace d'adressage. En 2006, grâce au support ajouté dans le noyau Linux 2.6, une véritable implémentation des threads noyaux a pu être proposée : NPTL (*Native POSIX Threads Library*). Un exemple de programme utilisant l'interface normalisée POSIX Threads est fourni en Figure 3.1.

Les threads ont d'abord été utilisés pour améliorer les performances des applications ayant fréquemment recours à la création/destruction de processus, tels que les serveurs web ou les serveurs de fichiers par exemple. L'intérêt d'utiliser des threads dans ce type d'application résidait dans une meilleure réactivité obtenue grâce au coût réduit des mécanismes de gestion des processus. Plus tard, les threads ont progressivement été utilisés pour paralléliser des codes de simulation sur machines multiprocesseurs, principalement en raison de la grande souplesse offerte par la mémoire partagée entre tous les threads. La parallélisation d'une application séquentielle en utilisant le multithreading demeure généralement moins intrusive qu'une parallélisation sur architecture distribuée.

La programmation concurrente en mémoire partagée apporte, en contrepartie, de multiples contraintes liées non seulement aux accès mémoire concurrents (*race conditions*), mais aussi à la cohérence mémoire offerte par la machine ainsi qu'aux optimisations mises en place par le compilateur. Ainsi, il est beaucoup plus délicat d'« échanger des informations » entre threads qu'il n'y paraît, et il est souvent impératif de recourir à l'utilisation de mécanismes de synchronisation (verrous, variables de conditions, sémaphores, etc.). Si la fonction première de ces outils est d'implanter des protocoles de synchronisation entre threads (exclusion mutuelle, barrières, etc.), il est important de se rappeler que ces mé-

```c
unsigned NBTHREADS = 16;

void *thread_func (void *arg)
{
  int me = (int)(intptr_t)arg;

  printf ("Hello from thread %d\n", me);

  return NULL;
}

int main (int argc, char *argv[])
{
  pthread_t pids[NBTHREADS];

  for (int i = 0; i < NBTHREADS; i++)
    pthread_create (&pids[i], NULL, thread_func, (void *)(intptr_t)i);

  for (int i = 0; i < NBTHREADS; i++)
    pthread_join (pids[i], NULL);

  return 0;
}
```

FIGURE 3.1 – Squelette de programme utilisant l'interface POSIX Threads pour lancer une équipe de processus légers.

canismes forment des points de mise en cohérence de la mémoire (vidage des tampons d'écriture des processeurs, etc.) et assurent que le compilateur ne puisse réordonner les instructions autour de ces points.

L'utilisation de l'interface POSIX Threads offre au programmeur une grande liberté quant à la façon de paralléliser un programme. En revanche, le placement des threads sur les cœurs sous-jacents nécessite l'utilisation de primitives non standard, ou d'outils externes (e.g., numactl). Par ailleurs, l'interface en elle-même n'aide pas à écrire des programmes portables d'une architecture à l'autre (e.g., choix du nombre de threads) : il est de la responsabilité du programmeur d'utiliser des bibliothèques de découverte de topologie et d'adopter une discipline de programmation stricte pour assurer cette portabilité.

3.2.1.b OpenMP

OpenMP est le standard de programmation élaboré par l'*OpenMP Architecture Review Board* regroupant de nombreux acteurs industriels et académiques de la communauté du calcul intensif. La norme OpenMP définit des extensions aux langages C, C++ et Fortran sous forme de *directives* (i.e. `#pragma omp`) de programmation. Ces directives indiquent au compilateur comment il doit « paralléliser » le code. Le compilateur ne vérifie pas que les directives sont correctement utilisées, comme par exemple que le code soit réellement parallélisable : c'est la responsabilité du programmeur d'utiliser OpenMP correctement.

La version 1.0 de la spécification du langage date de 1997. À l'époque, les machines cibles étaient principalement des machines multiprocesseurs à mémoire partagée uniforme (*Symmetric Multiprocessing*) et la parallélisation était surtout focalisée sur le parallélisme de boucle. Plus tard, le parallélisme de tâches a

été introduit pour permettre d'extraire du parallélisme dans des schémas de calcul moins réguliers que des boucles (spécification 3.0). En 2013, la spécification 4.0 d'OpenMP introduit même la possibilité de générer du code dont l'exécution sera déléguée à un accélérateur (FPGA, GPU).

Le modèle de programmation OpenMP est fondé sur l'utilisation sous-jacente d'équipes de threads qui travailleront de manière collaborative pour se répartir la charge de calcul des régions parallèles d'un code. En fait, un programme OpenMP est une succession de régions séquentielles et de régions parallèles. Une région parallèle est explicitement délimitée par le programmeur, et déclenche la création d'une équipe de threads. Dans la Figure 3.2, le bloc de code suivant la directive `#pragma omp parallel` (ligne 3) est exécuté par une équipe de threads.

```
1  {
2      ... // sequential code
3      #pragma omp parallel
4      {
5          ... // code executed by every thread
6          #pragma omp for
7          for (int i = 0; i < N; i++)
8            do_something (i);
9          ... // code executed by every thread (after an implicit barrier is met)
10     }
11     ... // sequential code
12 }
```

FIGURE 3.2 – La directive OpenMP `parallel` permet de créer une équipe de threads

Les threads d'une région parallèle exécutent chacun le bloc de code de manière concurrente, jusqu'à rencontrer une directive de synchronisation, ou de partage de travail (*OpenMP work sharing construct*).

La directive de partage de travail la plus communément utilisée est assurément celle qui permet de distribuer les itérations d'une boucle parmi l'équipe de threads d'une région parallèle. Cette directive est illustrée en Figure 3.2, ligne 6. Il existe une multitude de clauses optionnelles permettant de spécifier quelles sont les variables qui doivent rester partagées ou, au contraire, être transformées en variables privées aux threads, de choisir la stratégie de distribution des itérations, ou même de supprimer la barrière implicite de fin de boucle.

Les approches à base de directives de programmation telles que OpenMP encouragent une parallélisation incrémentale du code et permettent parfois de conserver la structure originelle du code séquentiel. Dans ce cas, on peut retrouver le comportement séquentiel original du programme en ignorant les directives OpenMP à la compilation. L'inconvénient est qu'en partant d'un code séquentiel, le programmeur reste parfois prisonnier d'un schéma de pensée « séquentiel » peu adapté à une parallélisation de degré massif.

En effet, la loi de Amdahl [5] implique que le gain maximal apporté par le parallélisme est contraint par la proportion de calcul séquentiel restant. Par exemple, si 90% de l'exécution est parallélisée, les 10% de calcul séquentiels restant limiterons le gain théorique maximal à un facteur 10.

La première version d'OpenMP est sortie à une époque où les supercalcu-

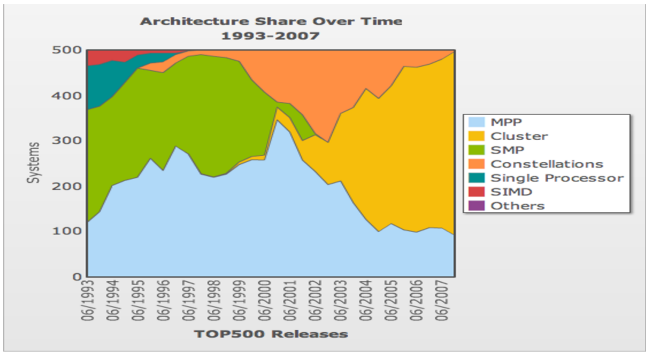


FIGURE 3.3 – Représentation des différentes architectures de supercalculateurs du TOP 500 entre 1993 et 2007 (source : top500.org)

lateurs à mémoire commune étaient en nette perte de vitesse (voir la représentation des architectures SMP à partir de 1997 sur la Figure 3.3). C'est en effet à cette période qu'ont émergé les grappes de calcul (*PC Clusters*) qui étaient des architectures reliant via des réseaux rapides (Myrinet, Quadrics, Infiniband) des ordinateurs de type station de travail. Ces architectures se sont progressivement démocratisées face aux supercalculateurs traditionnels en raison de performances honorables pour une évolutivité très supérieure et un coût moindre. Pendant longtemps, ces grappes ont été majoritairement constituées d'ordinateurs possédant un faible nombre de processeurs (2, 4), ce qui explique qu'OpenMP n'ait pas connu le succès escompté à sa sortie. Il suffisait alors d'utiliser des approches à base de processus (e.g., MPI) pour exploiter correctement ces architectures.

Un peu plus tard, la ré-émergence des architectures à accès mémoire non-uniforme (NUMA) due au basculement vers des puces multicoeurs, a ré-introduit la nécessité de disposer de modèles de programmation à grain fin. Malheureusement, OpenMP a, là encore, souffert d'un support tardif pour les architectures NUMA, notamment en matière de maîtrise du placement des calculs et des données. En définitive, il a réellement fallu attendre l'arrivée des fonctionnalités de calcul à base de tâches (décrites ci-après, section 3.2.2.b, page 41) dans la norme OpenMP pour voir la communauté s'ouvrir plus largement à ce langage.

3.2.2 Programmation orientée "tâches"

Le modèle de programmation "orienté thread", dans lequel chaque activité parallèle est propulsée par un thread indépendant, n'est efficace que si la durée des activités est longue (autrement dit, si le parallélisme est à gros grain). Dans le cas d'un parallélisme plus dynamique exhibant de nombreuses activités parallèles de courte durée, le coût de création, de destruction et d'ordonnancement des threads devient rédhibitoire.

C'est pourquoi, depuis longtemps, de nombreux programmes parallèles s'appuient sur un *pool* de threads fixe (souvent un par cœur) et utilisent des outils de partage de travail (*worksharing constructs*) entre threads pour répartir les activités dynamiquement. La directive OpenMP `#pragma omp for` est un exemple d'outil permettant de distribuer les indices de l'intervalle d'itération

d'une boucle parmi les threads sous-jacents. Plus généralement, il est possible d'utiliser des outils de synchronisation (e.g., verrous) pour mettre en place un "distributeur" d'activité dynamique, pour peu que l'on puisse facilement décrire les activités et leurs paramètres sous une forme canonique. Cependant, la programmation devient rapidement fastidieuse en l'absence d'un support au niveau du langage.

C'est pour faciliter l'écriture de programmes parallèles impliquant l'utilisation dynamique et souple d'activités parallèles à grain fin que la notion de "tâche" est apparue dans les langages de programmation parallèles (Cilk, Intel TBB, OpenMP, etc.). L'idée est d'introduire une construction syntaxique permettant de capturer un morceau de code et son environnement (variables ou paramètres) et de laisser le soin à un moteur d'exécution sous-jacent d'exécuter la tâche de manière asynchrone sur l'une des unités de calcul de la machine. Dans certains langages, il est possible de spécifier précisément, pour chaque variable visibles par la déclaration de la tâche, si la valeur doit être capturée par copie (variable privée) ou si la variable doit être référencée au moyen de son adresse (variable partagée).

Hormis dans le contexte d'une application exhibant un parallélisme trivial, certaines tâches sont liées à d'autres par des contraintes de précédence (e.g., une tâche B ne peut s'exécuter tant qu'une tâche A n'est pas terminée) lorsque l'une produit des résultats en sortie que d'autre prennent en entrée. Les modèles de programmation à base de tâches permettent donc d'établir des **dépendances** entre les tâches, ce qui forme un graphe de tâches implicite dans lequel chaque arête marque une dépendance. Le support d'exécution est alors en charge, à tout moment, d'ordonnancer uniquement les tâches dont toutes les dépendances ont été satisfaites.

Différents mécanismes ont été proposés pour établir des dépendances entre les tâches. Dans le langage Cilk [49, 21], par exemple, les dépendances ne sont possibles qu'entre une tâche et ses tâches filles : un mot clé `sync` permet d'introduire un point de synchronisation où la «continuation de la tâche en cours» ne pourra reprendre que lorsque toutes les tâches filles créées par la tâche courante seront terminées. Le langage OpenMP possède une construction similaire (`#pragma omp taskwait`), mais propose également d'inférer les dépendances entre tâches en fonction des accès aux données (lecture/écriture) spécifiés explicitement. Pour une série de tâches créées en séquence, les dépendances sont ajoutées de manière à ce que l'éxécution soit conforme à une exécution séquentielle du point de vue de l'intégrité des données. On appelle cette façon de générer un graphe de tâche la méthode STF (*Sequential Task FLow*).

De nombreux langages et supports d'exécution offrent un support natif pour la programmation à base de tâches (StarPU [13], Charm++ [57], Intel TBB [75] ou encore Cilk [49]). Nous en présentons quelques uns parmi les plus caractéristiques.

3.2.2.a Cilk

Cilk [49, 21] est un système de parallélisation en mémoire partagée. Il se base sur une extension du langage C, en y rajoutant des mots-clés permettant d'exprimer le parallélisme (voir exemple Figure 3.4). À l'origine, la génération de tâches dans Cilk obéissait à un schéma récursif : les tâches créent des tâches filles, qui à leur tout créent des tâches filles, etc. Pour équilibrer les calculs entre

les différents threads, il utilise du vol de travail, en se basant sur du « *work first principle* », qui stipule que les threads exécutent prioritairement les tâches en profondeur, et que les vols de tâches tentent de voler les tâches les plus hautes dans l'arbre, susceptibles de générer davantage de parallélisme que les tâches de niveau inférieur.

L'implémentation de Cilk est très sophistiquée, et s'appuie sur un compilateur capable de fabriquer une version du code très proche du code séquentiel d'origine (aka « dégénérescence séquentielle des tâches ») qui reste donc très efficace pour la partie exploratoire en profondeur. Le surcoût du parallélisme est reporté sur la phase de vol elle-même, qui est supposée être peu fréquente. Le vol de tâches est mis en œuvre au moyen de structure de données *lock-free*. Le choix du processeur victime s'effectue aléatoirement.

```
1  cilk int fib (int n)
2  {
3    if (n < 2) return n;
4    else {
5      int x, y;
6      x = spawn fib (n-1);
7      y = spawn fib (n-2);
8      sync;
9      return x + y;
10   }
11 }
```

FIGURE 3.4 – Exemple d'implémentation parallèle avec Cilk 5 de la fonction Fibonacci s'appuyant sur un arbre de tâches généré récursivement

Cilk a été un langage précurseur en ce qui concerne l'utilisation des tâches à grain fin pour le calcul parallèle, et a inspiré de nombreux effort de recherche, notamment le projet XKaapi [50].

3.2.2.b OpenMP 3.5+

OpenMP est un système de parallélisation en mémoire partagée, se basant sur l'utilisation de directives de pré-processeur pour exprimer le parallélisme. Depuis la version 3.5 de ce système, le support des tâches a été rajouté à la norme. Comme pour les sections parallèles, la création de tâche se fait à l'aide d'une unique directive, à laquelle peuvent être ajoutés des mots-clés, pour préciser le contexte de ces tâches.

Par exemple, il est possible d'indiquer, grâce à ces mots-clés, quelles données sont manipulées, et donc quelles sont les dépendances entre les différentes tâches.

Enfin, OpenMP permet aux threads de se synchroniser avec les différentes tâches, avec la directive *#pragma omp taskwait*. Lors de ces synchronisations, la tâche en cours d'exécution est mise en pause, et le thread courant exécute de nouvelles tâches, ce qui lui permet de continuer à effectuer des calculs et donc de ne pas réduire l'efficacité du code.

La Figure 3.5 montre la création de trois tâches en utilisant ce système. Les deux premières vont respectivement écrire dans les variables data1 et data2, et peuvent être exécutées en parallèle.

```
{
    ... // sequential code
    #pragma omp parallel
    {
        ... // a single thread create the tasks
        #pragma omp single
        {
            this task produce data1
            #pragma omp task depend(out: data1)
            f(data1);

            //this task produce data2
            #pragma omp task depend(out: data2)
            g(data2);

            //this task use data1 and data 2
            #pragma omp task depend(in: data1,data2)
            h(data1, data2);

            // the submiter thread wait the execution of the tasks before exit the single section
            #pragma omp taskwait
        }
        ... // all thread execute the tasks in the implicit barrier
    }
    ... // sequential code
}
```

FIGURE 3.5 – La directive OpenMP `task` permet de créer des tâches de calcul

La dernière tâche va utiliser les valeurs produites, et ne peut donc pas être exécutée tant que les deux premières tâches n'ont pas terminé leur exécution. Enfin, le thread soumettant les tâches attend l'exécution des tâches soumises, ce qui permet de s'assurer qu'elles ont effectivement été exécutées à la fin de notre exemple.

3.2.2.c Supports exécutifs pour du parallélisme de tâche

Afin de permettre au plus grand nombre d'applications d'adopter un parallélisme à base de tâches sans nécessiter leur réécriture dans un nouveau langage de programmation, de nombreuses bibliothèques proposant une interface C/C++ dédiée à la gestion de tâches ont vu le jour. Ces « supports d'exécution » (ou *runtime systems* en anglais) servent ainsi de couche de portabilité sur lesquelles peuvent s'appuyer aisément des applications de simulation numérique telles que les solveurs en algèbre linéaire par exemple.

Le solveur PLASMA [44], par exemple, s'appuyait au départ sur un tel support exécutif nommé Quark. Ce dernier proposait un ordonnancement à faible surcôut de tâches au-dessus d'architectures multicœurs [91]. Plus récemment, l'implémentation de PLASMA repose directement sur le standard OpenMP, dont les évolutions suivent les mêmes principes que ceux des supports exécutifs sur CPU, tout en offrant une meilleure portabilité puisqu'il s'agit d'un standard [92].

Les supports d'exécution à base de tâches ont réellement prouvé leur utilité avec l'adaption des cartes graphiques (GPU) en tant qu'accélérateurs de calcul

dans le domaine du HPC. Contrairement aux approches statiques de type « *offloading* » où le programmeur indique explicitement quels sont les calculs dont l'exécution doit être déportée sur GPU, les supports d'exécution peuvent dynamiquement décider d'exécuter chaque tâche sur CPU ou sur GPU en fonction de nombreux paramètres (algorithme d'ordonnancement, état de la machine, etc.).

Le support exécutif StarPU [13], par exemple, propose une interface de tâches hétérogènes qui peuvent être réparties automatiquement entre les différentes ressources de calcul disponibles au sein d'un noeud (CPU, GPU, ...). StarPU automatise les mouvements de données et la synchronisation des calculs. Une attention toute particulière est également portée par StarPU au problème de l'ordonnancement des calculs entre les différentes unités de calcul. Cela permet donc d'exploiter efficacement des machines multi-coeur potentiellement hétérogènes. En revanche, si des extensions de StarPU existent pour cohabiter efficacement avec MPI [10], ou automatiser les échanges à l'aide de MPI, ce modèle de programmation n'offre pas de support immédiat pour un rééquilibrage des calculs sur une machine à mémoire distribuée, avec des mécanismes de vol de tâche par exemple.

L'environnement de programmation PARSEC vise quant à lui explicitement les architectures hétérogènes sur des machines à mémoire distribuée. Contrairement à StarPU qui se base sur un modèle de programmation où les dépendances de données sont généralement déduites automatiquement des accès aux données (modèle aussi appelé *Sequential Task Flow*, ou STF), le graphe de tâches utilisé par PARSEC est obtenu à partir d'une description de haut niveau de l'algorithme appelée JDF [23], similaire aux travaux de Cosnard *et al.* [39]. Cela permet à PARSEC d'obtenir d'excellents résultats en termes de scalabilité, et d'utilisation des accélérateurs, même si cette approche se limite en grande partie à ce qui peut s'exprimer à l'aide de JDF, c'est à dire à des problèmes d'algèbre linéaire la plupart du temps.

D'autres systèmes tels que KAAPI offrent un support pour les machines équipées d'accélérateurs, tout en offrant certaines fonctionnalités de rééquilibrage de charge [50].

Ainsi, de nombreuses solutions de type support exécutifs sont désormais à la disposition des applications, des bibliothèques, ou même des environnements de compilation afin d'abstraire la complexité des machines manycores, avec ou sans accélérateurs [85]. A l'instar de l'environnement OmpSs [46] et d'OpenMP, on constate que l'évolution des langages et des standards de programmation est très liée avec l'évolution des travaux autour des supports exécutifs.

Le CEA, a fait le choix de développer son propre système de gestion de tâches – nommé libtask – pour mieux répondre aux exigences du solveur hiérarchique présenté en section 2.3, ainsi qu'aux contraintes de la couche de communication des supercalculateurs du CEA. À l'instar des supports d'exécution présentés précédemment, libtask permet de soumettre des graphes de tâches de manière asynchrone. Un accent particulier a été mis sur la possibilité de soumettre des tâches de communication interruptibles, c'est-à-dire des tâches qui peuvent supporter l'implantation de protocoles de communication complexes et asynchrones dont la progression s'effectue par étapes. Ces tâches de communication disposent du même système de résolution des dépendances que les autres, et s'intègrent donc parfaitement dans le déroulement du flow de l'application (cf Section 3.5.2). Le second point important est que libtask est capable de gérer finalement des dépendances sur des données hiérarchiques, comme détaillé dans

la section ci-après.

3.3 Dépendances hiérarchiques

Dans la Partie 2.2.2.b, nous avons vu que le solveur utilisé s'appuyait sur des matrices compressées hiérarchiquement. Dans le cadre de ce type de structures hiérarchiques, l'accès à un élément en lecture et/ou en écriture induit implicitement l'accès à ses fils, avec le même mode d'accès.

Ceci se traduit, dans le cadre d'un solveur à base de tâche, en la nécessitée de créer des dépendances sur l'accès à ces fils dès lors qu'une ou plusieurs tâches accèdent aux parents.

De la même manière, une tâche ne peut manipuler les parents d'une matrice si cette matrice est en train d'être manipulée par une autre tâche, ce qui se traduit cette fois par le positionnement de dépendances chez ses parents.

En revanche, nous pouvons noter que l'accès aux frères de cette même matrice est possible sans conflits, ce qui permet donc d'effectuer des calculs sur ces différentes parties en parallèle, et donc d'augmenter l'efficacité du calcul.

Cependant, le positionnement correct de ces dépendances peut être complexe à mettre en place pour permettre aux calculs s'effectuant sur plusieurs sous-matrices différentes de s'effectuer en parallèle, et d'autre part garantir l'absence de conflits.

Il est ainsi difficilement envisageable de positionner manuellement ces dépendances. Un système automatique a donc été mis en place, qui positionne automatiquement des dépendances sur les matrices filles et mères concernées. Cette mise en place a permis d'augmenter la quantité de calcul parallèle au sein du solveur, et donc l'efficacité de ce dernier [11].

3.4 Programmation des architectures distribuées

La programmation des architectures distribuées pose une contrainte majeure : celle de ne pas pouvoir reposer sur une mémoire commune qui serait partagée par tous les processeurs de la machine. Depuis que de telles architectures existent, deux approches s'opposent. Il y a d'un côté les approches qui proposent un modèle de programmation dans lequel les échanges de données entre les processus sont effectués de manière explicite par l'utilisation de primitives de communication disséminées dans le code. De l'autre côté, il y a les approches implicites qui fournissent l'illusion d'une mémoire (virtuellement) partagée entre les processus.

Cette dernière approche, qui est majoritairement constituée de "Systèmes à Mémoire Virtuellement Partagée" (*Software SVM systems* en anglais), avait pour objectif initial d'utiliser des modèles de programmation prévus pour des architectures à mémoire commune (tels que Pthreads) avec un minimum de modifications sur architectures distribuées. Cet axe de recherche a été intensivement exploré dans les années 90 (Ivy, Treadmarks, Munin, etc.), et certaines technologies réseaux ont même été développées pour faciliter les accès mémoire à distance de manière complètement transparente pour un processeur (e.g., Scalable Coherent Interface, avec une implémentation matérielle proposée par Dolphin CS). Toutefois, programmer une machine distribuée "comme si" la

mémoire était physiquement partagée impose de choisir un modèle de cohérence mémoire un peu plus relâché que le modèle de "cohérence strictement séquentielle" des données (qui n'est d'ailleurs généralement pas supporté par certaines architectures à mémoire commune, tant il serait coûteux à implanter). Ainsi, le rêve d'une mémoire partagée virtuelle au-dessus d'une architecture distribuée se heurte à un compromis entre la transparence totale au prix de l'utilisation de protocoles de cohérence des données extrêmement coûteux, et un protocole de cohérence efficace imposant des points de synchronisation (i.e. mise en cohérence) explicites des données. Dans ce second cas, les bonnes performances sont obtenues grâce au fait que la cohérence des données n'est effectuée qu'en certains points du programme. Le modèle de programmation PGAS (Partitioned Global Address Space) s'inscrit dans cette catégorie et constitue aujourd'hui la seule alternative crédible aux modèles de programmation à base d'échanges de messages.

3.4.1 Le modèle PGAS

Le modèle PGAS [93] (Partitioned global address space) permet ainsi d'exposer un espace d'adressage où des segments de mémoire peuvent être locaux, ou partagés avec d'autres processus. Ce modèle est à la base de langages ou d'environnements comme Coarray Fortran [69], UPC [29], Chapel [31], X10 [33], Legion [16], ou OpenSHMEM [32]. On peut en particulier remarquer que le système de communication GASNET offre une alternative à MPI qui permet la mise en oeuvre de tels modèles de programmation [22].

Bien que ces modèles facilitent *a priori* grandement la conception d'applications complexes sur des machines à mémoire distribuée, leur adoption reste en pratique encore limitée. Simplifier ou cacher les mécanismes de communication ne suffit par ailleurs pas à concevoir des algorithmes efficaces à grande échelle puisqu'il faut toujours veiller à un bon équilibrage de charge et éviter les échanges superflus ou avec des synchronisations potentiellement implicites mais inefficaces.

Les objectifs de cette thèse visant à rééquilibrer la charge d'applications irrégulières restent donc pleinement valables sur des systèmes de communication dits modernes, même si certains mécanismes comme le vol de tâche ou les actives messages sont rendus plus simples. Il sera donc intéressant de combiner nos techniques destinées à décider comment rééquilibrer la charge avec ces approches modernes, en particulier sur des architectures avec des systèmes mémoire complexes potentiellement équipées de mémoires avec des performances hétérogènes (e.g. HBM), ou avec des accélérateurs.

3.4.2 *Message Passing Interface*

MPI (*Message Passing Interface*) est un standard permettant d'exprimer un parallélisme distribué. Ce standard propose d'utiliser une séparation de la mémoire entre les différents processus, et d'utiliser des communications explicites pour l'échange d'informations. Les processus impliqués peuvent être placés sur des nœuds de calculs séparés, mais aussi se partager un même nœud, MPI pouvant alors remplacer les méthodes de mémoire partagée présentées dans la Partie 3.2.

Ce standard se reposant sur l'utilisation de plusieurs processus distants, les modifications de la mémoire d'un processus ne sont pas directement visibles par les autres. Pour communiquer les résultats des différents calculs, il est alors nécessaire d'effectuer des communications explicites, en se reposant sur la bibliothèque MPI.

Une première façon d'utiliser ces communications est d'utiliser des communications point à point bloquantes. Celles-ci mettent en jeu un processus origine, qui envoie les données, et un processus cible, qui reçoit les données. Ces communications sont dites bloquantes car les appels à la bibliothèque MPI interrompent le fil d'exécution des appelants jusqu'à ce que la communication soit terminée.

Il est parfois intéressant de faire participer plus de deux processus aux méthodes de communication, par exemple pour distribuer les données parmi les différents processus. Dans ce cas, l'utilisation de communications point à point va nécessiter l'utilisation de plusieurs appels à la bibliothèque MPI, et nécessite de prendre en compte l'architecture du réseau pour que ces communications soient efficaces.

MPI propose donc des communications collectives, auxquelles participent tous les processus impliqués dans le transfert. Ces communications peuvent être optimisées pour tenir compte du réseau, et donc réduire le temps nécessaire à leur exécution.

Les communications MPI sont également disponibles en mode non bloquant. Dans ce mode les participants aux communications peuvent déclencher les envois et réceptions sans bloquer leur fil d'exécution, ce qui leur permet d'exécuter d'autres actions, comme poster d'autres communications ou effectuer d'autres calculs par exemple.

Dans ce dernier cas, les communications sont recouvertes par des calculs, ce qui permet d'obtenir une meilleure efficacité des calculs, au prix d'une plus grande complexité des algorithmes pour prendre en compte cet asynchronisme.

Enfin, un dernier type de communication, dit « one sided » est proposé par MPI. Dans celui-ci, les différents processus vont désigner des zones mémoire dans lesquelles les autres participants pourront accéder en lecture ou en écriture, sans nécessiter de participation du processus local.

Ces communications permettent donc de réutiliser une grande partie des algorithmes développés en mémoire partagée, au prix d'une plus grande latence des accès mémoire. De plus, comme en mémoire partagée et contrairement aux modes de communication décrits précédemment, il est nécessaire de veiller à ce que les accès simultanés ne soient pas conflictuels.

3.4.3 Accès mémoire distants

Dans la partie précédente, nous avons présenté le standard MPI. Nous avons vu que les communications pouvaient s'effectuer en mode point à point, collectif ou en accès direct. Dans cette partie nous commencerons par présenter l'historique de ces communications en accès direct avant de présenter plus en détail la gestion de ce type d'accès dans la bibliothèque MPI.

3.4.3.a Historique

Dans la Partie 3.2, nous avons vu que plusieurs threads d'un même processus se partagent la même zone mémoire. De manière similaire, la mémoire utilisée

par les différents processus d'une machine peut être accédée par différentes cartes d'extension, telles que les cartes graphiques branchées sur les ports PCI.

De cette façon, il est possible de réduire le temps passé par le CPU à transférer les données de la mémoire à la carte, cette action étant effectuée par la carte d'extension. De plus, il est également possible d'interagir directement avec la carte, sans passer par le système d'exploitation via des appels systèmes qui peuvent être coûteux en temps.

C'est en 1995 qu'il a été proposé pour la première fois d'utiliser cette méthode pour les cartes réseaux, afin de permettre une bonne efficacité des communications, notamment de petite taille [87]. Sur les réseaux le supportant, une évolution logique a alors été d'exploiter cette possibilité d'accès à la mémoire par les interfaces réseaux pour permettre aux différents processus de manipuler la mémoire des autres processus.

En 1997 la version 2.0 de la norme MPI est publiée, et propose dans une de ses sections de pouvoir utiliser cette technologie à l'aide de la bibliothèque. Dans la partie suivante, nous allons présenter cette partie de la norme MPI.

3.4.3.b Le mécanisme de RDMA dans MPI

Depuis la version 2.0 de la norme MPI, une section nommée « *One-Sided Communications* » détaille l'utilisation des communications en accès directs dans le contexte de MPI. Dans cette partie, nous allons maintenant détailler cette partie de la spécification.

Une première spécificité réside dans le dédoublement des zones mémoire dédiées aux accès directs. En effet, la norme MPI considère ces zones comme étant séparées en deux copies : une copie privée, accessible localement, et une copie publique, accessible à distance par les autres processus.

Deux modèles mémoire sont proposés par MPI pour la synchronisation de ces deux zones mémoire : le premier proposant une unification de la mémoire. Dans ce modèle les copies sont toujours identiques, aux synchronisations près. Le second propose au contraire une séparation des zones mémoire, les synchronisations se faisant à certains points précis de l'exécution.

De la même manière, deux méthodes sont proposées par la norme MPI pour synchroniser les deux zones mémoire : une méthode dite à cible active (dénommée « *active target communication* » dans la norme), et une méthode dite à cible passive (dénommée « *passive target communication* » dans la norme).

La méthode de synchronisation à cible active impose la participation active et explicite des deux processus impliqués dans l'échange de données. Comme le souligne la norme MPI, cette méthode se rapproche des communications point à point, à la différence qu'un seul processus indique quel est le format des données échangées.

De cette façon, il est par exemple possible d'effectuer des transferts de données irrégulières sans devoir mettre en place de système de consensus permettant aux deux processus de connaître le format des données échangées au début des communications.

En revanche, comme les deux processus participent aux échanges, il est nécessaire de prévoir un système de rendez-vous permettant aux deux processus de participer activement aux communications. Cette limitation peut s'avérer contraignante dans le cadre d'un équilibrage dynamique, où il peut être intéressant d'effectuer des rééquilibrages opportunistes.

À l'inverse la méthode à cible passive permet de transférer les données d'un processus à l'autre en ne nécessitant l'intervention que d'un seul des deux processus.

Il est alors possible à deux processus distincts d'initier simultanément un transfert vers un troisième processus. Ce dernier ne participant pas aux échanges, il ne peut pas arbitrer en faveur d'un processus ou d'un autre.

Or, de la même façon qu'en mémoire partagée, il n'est pas possible d'effectuer plusieurs accès en écriture simultanée sur la même variable via le mécanisme MPI RDMA.

Pour garantir l'exclusion de l'accès, il est imposé par la norme MPI de verrouiller la zone mémoire du processus cible avant les accès, et de libérer le verrou une fois ceux-ci terminés.

Ces points de synchronisation permettent une grande amplitude dans l'implémentation de cette partie de la norme MPI. Par exemple, il est possible de simuler des accès RDMA sur des réseaux point à point, en implémentant la libération des verrous comme un envoi MPI en mode point à point. En revanche, l'efficacité des codes s'appuyant sur les accès directs peut se retrouver impactée par ce type de solution.

3.4.3.c Accès concurrents

Dans la partie précédente, nous avons vu qu'il est nécessaire de verrouiller les zones mémoire accessibles en mémoire partagée pour pouvoir y accéder sans coopération du processus cible. Ce verrou pose deux principaux problèmes : tout d'abord, il n'est pas possible de demander à le verrouiller uniquement si aucun autre processus n'en a demandé l'exclusivité, comme il est possible de le faire avec les verrous de la bibliothèque pthread, avec l'appel à la fonction *pthread_ mutex_ trylock*.

De plus il n'est pas possible de verrouiller seulement une partie de la zone mémoire sans la découper en plusieurs zones MPI. Cependant, certaines structures de données sont difficilement découpables en plusieurs parties distinctes, comme par exemple une liste chaînée.

En revanche, contrairement aux mutex de pthread, il est possible de verrouiller les zones mémoire en accès partagé. D'autres processus peuvent alors "verrouiller" cette même zone, à la condition que le mode de verrouillage soit lui aussi en accès partagé.

Il est alors possible d'accéder à la zone mémoire en lecture de manière concurrente, ou d'accéder en écriture à plusieurs parties distinctes de la zone mémoire. De cette façon il est possible d'implémenter un système de lecteurs/écrivains où un seul processus peut accéder à une zone mémoire en écriture, ou plusieurs en lecture.

Une autre possibilité pour gérer des accès concurrents est d'utiliser des accès atomiques, qui sont proposés dans le standard MPI. Ce type d'accès permet d'effectuer des opérations en lecture ou en écriture en garantissant qu'un autre processus ne puisse pas interférer en accédant à la même partie de la zone mémoire.

En pratique, et dans MPI, il est possible d'utiliser toutes les opérandes de *MPI_ Reduce*, ainsi que l'opération *test_ and_ set*. Cette dernière permet de ne changer la valeur d'une variable qu'à la condition que cette variable soit égale à

une valeur donnée, et d'être informé ou non du succès de l'opération. Cette opération permet donc d'implémenter un système de verrous, qui peut par exemple être récursif ou supporter un verrouillage conditionnel.

Il est également possible, à l'aide de cette implémentation, de créer plusieurs verrous dans une même zone mémoire, ce qui permet de créer des structures de données accessibles en écriture en utilisant des opérations non atomiques.

Nous pouvons enfin noter que ces opérations atomiques ne sont pas compatibles avec les opérations atomiques classiquement utilisées en multithreading au sein d'un noeud (e.g. avec des *intrinsics* du compilateur, ou via les supports exécutifs de pthreads ou OpenMP). En effet la norme MPI autorise les implémentations à effectuer les mises à jour en plusieurs écritures, à condition de garantir que les autres accès atomiques ne puissent accéder à un état intermédiaire. La norme MPI autorise cependant explicitement de mélanger les opérations MPI et des opérations atomiques locales à condition qu'il s'agisse d'opérations sur un seul et unique octet.

3.5 Composition de paradigmes : MPI + X

Dans la Partie 3.4.2, nous avons vu que la bibliothèque MPI permet de créer plusieurs processus communiquant entre eux, pour créer du parallélisme au sein des calculs. Ces processus peuvent notamment se situer sur le même nœud de calcul, permettant d'exploiter les différents threads des machines.

Cependant, nous pouvons noter que lorsqu'un grand nombre de processus MPI sont instanciés, les communications perdent en efficacité, réduisant l'intérêt de cette approche. Il est alors intéressant de limiter le nombre de processus MPI en utilisant des techniques hybrides exploitant efficacement la mémoire partagée au sein d'un nœud, et d'utiliser MPI pour les communications inter-nœuds.

Comme nous l'avons vu, plusieurs méthodes de parallélisme en mémoire partagée existent. La plupart peuvent être combinées avec des bibliothèques de communication telles que MPI. Par exemple nous pouvons citer la bibliothèque OpenMP, que nous avons présentée dans la Partie 3.2.1.b, ou encore l'utilisation de supports exécutifs, que nous avons présentés dans la Partie 3.2.2.c.

La combinaison de MPI avec une méthode de parallélisation en threads permet de réduire le nombre de processus nécessaires à l'occupation de ressources de calcul, et donc d'augmenter l'efficacité des communications entre les processus restants. Nous verrons cependant que ces deux modèles présentent une incompatibilité intrinsèque, qui doit être prise en compte pour les utiliser conjointement de manière efficace.

Notre bibliothèque, comme d'autres runtimes, permet également d'instancier des tâches, et donc de disposer d'un parallélisme efficace. Nous verrons que cette méthode peut également être utilisée dans un contexte distribué, et est utile dans la gestion des communications asynchrone

3.5.1 Les difficultés posées par le mélange de deux modèles qui n'ont pas été conçus pour fonctionner ensemble

Comme nous l'avons vu, la bibliothèque OpenMP permet de paralléliser les algorithmes en utilisant seulement quelques directives. Cependant, cette paral-

lélisation n'utilise pas automatiquement MPI, et ne peut donc être exploitée qu'au sein d'un nœud de calcul, ce qui impose de concevoir également une parallélisation à l'aide de MPI.

Ces deux paradigmes étant intrinsèquement différents, nous pouvons noter une première difficulté à concevoir des algorithmes parallèles exploitant ces deux méthodes en même temps. Une solution généralement retenue consiste à effectuer une première parallélisation à l'aide de MPI sur les différents processus, puis à paralléliser une seconde fois à l'aide d'OpenMP entre les threads de chaque processus.

De plus, si la norme MPI prévoit bien l'existence de processus MPI comportant plusieurs threads, quatre niveaux de compatibilité existent, et les différentes implémentations MPI ne sont pas contraintes de tous les implémenter. La sélection du niveau de compatibilité se fait au moment de l'initialisation de la bibliothèque, via l'appel *MPI_Init_thread* qui remplace l'appel *MPI_Init* utilisé dans des applications mono-threadées.

Le niveau de compatibilité le plus restrictif est *MPI_THREAD_SINGLE*. L'utilisation de ce mode indique à MPI que les processus n'instancieront qu'un seul thread. Plusieurs optimisations sont alors possibles, comme par exemple l'utilisation de fonctions de la bibliothèque standard non thread-safe sans risquer une corruption de la mémoire.

Le second niveau de compatibilité est *MPI_THREAD_FUNNELED*. Dans ce cas, plusieurs threads peuvent exister au sein d'un processus mais seul celui ayant appelé la fonction d'initialisation peut réaliser des communications.

Le troisième niveau est *MPI_THREAD_SERIALIZED*. Dans ce mode, tous les threads peuvent effectuer des appels à la bibliothèque MPI, à condition qu'il n'y ait jamais plus d'un thread faisant un appel à la bibliothèque MPI en même temps.

Enfin, le niveau le plus permissif est *MPI_THREAD_MULTIPLE*. Dans ce mode, aucune restriction n'est placée sur les appels à la bibliothèque MPI par les différents threads. Cependant ce dernier n'est pas toujours implémenté de manière efficace par les différentes implémentations. Pour des raisons de compatibilité, nous n'avons donc pas utilisé ce dernier niveau, et nous sommes donc restreints à l'utilisation des trois premiers.

De plus nous pouvons noter que même en la présence d'une bonne implémentation de *MPI_THREAD_MULTIPLE*, l'usage d'un grand nombre de communications simultanées peut facilement surcharger le réseau de communication. De plus, il est généralement possible de sérialiser plusieurs communications, permettant de réduire le nombre de threads impliqués dans les appels MPI, et donc la contention sur leurs algorithmes internes. Pour ces raisons les appels MPI sont rarement effectués dans des régions parallèles.

3.5.2 Le problème des tâches de communication

De manière logique, une communication doit débuter après que les données à envoyer aient été calculées, et doit se terminer avant le début des calculs nécessitant ces informations. Il est donc naturel d'intégrer les échanges de données dans le modèle STF. Pour cela, des tâches de communication sont mises en place, qui ne contiennent pas de calculs mais uniquement les appels à MPI pour les échanges de données.

De manière générale, les tâches de calcul s'exécutent sans interruptions, si l'on exclut l'attente d'éventuelles dépendances. Cependant, comme nous l'avions expliqué précédemment, il est important que les communications effectuées soient des communications de type non bloquantes. Ce type de communications étant difficilement compatible avec la non interruption des tâches, un nouveau mécanisme a été nécessaire, permettant d'interrompre les tâches de communication au cours de leur exécution, à la manière des coroutines.

Les tâches interrompues sont réinsérées dans la liste des tâches exécutables, pour pouvoir reprendre leur exécution ultérieurement. Pour que la tâche puisse reprendre son exécution sans devoir ré-exécuter toutes les actions précédentes, un système d'étape a été mis en place.

Les tâches sont découpées en plusieurs parties logiques, chacune identifiée par un unique entier. Lorsqu'une tâche finit d'exécuter l'une d'entre elles, elle passe à la suivante, et conserve cette information dans un compteur interne.

De cette façon quand une tâche est redémarrée, elle peut passer les étapes déjà effectuées avec succès, et reprendre son exécution là où elle s'était arrêtée.

```
1  int task_recv(task* t){
2    switch(t->step){
3    case 1: //étape 1 : la communication est soumise
4      MPI_Irecv(...);
5      t->step = 2;
6    case 2: //étape 2 : la complétion de la communication est testée
7      int completion = MPI_Test(...);
8      if(!completion)
9       {
10       // la communication n'est pas terminée,
11       // la tâche sera ré-exécuté plus tard
12       return TASK_YIELD;
13     }
14     else
15     {
16       // la communication est terminé, la tâche peut se couper définitivement,
17       // et libérer ses dépendences.
18       return TASK_EXIT;
19     }
20   }
21 }
```

FIGURE 3.6 – Code simplifié d'une tâche de communication

Dans la Figure 3.6, nous pouvons voir un exemple simplifié d'une tâche de communication s'interrompant si la communication n'est pas terminée.

Enfin, nous pouvons noter que pour des raisons de performances de certaines implémentations, il peut être intéressant de limiter l'exécution de ces tâches de communication sur un seul thread, qui sera appelé thread de communication. De plus, il a été observé que l'exécution de tâches de calcul, potentiellement longues, pouvait réduire le nombre de tâches de communication exécutées dans une période de temps donnée, augmentant la latence et donc dégradant les performances des communications. Pour résoudre ce problème, une solution est de contraindre le thread de communication à n'exécuter que des tâches de communication.

3.6 Rééquilibrage de charge dans les environnements de programmation distribués

De nombreux langages ou environnements de programmation permettent d'effectuer des calculs sur des machines à mémoire distribuée. Nous considérons ici en particulier les systèmes qui offrent un support pour de l'équilibrage de charge.

Charm++ [57] est un environnement de programmation qui permet d'exprimer une application distribuée à l'aide d'un paradigme de passage de messages. Contrairement à MPI où les échanges se font entre des processus a priori fixes, Charm++ propose une abstraction appelée *chare* similaire à un processus, et qui peut communiquer par passage de messages avec d'autres *chares*. Un programme écrit avec Charm++ se compose ainsi d'un nombre de *chares* généralement bien supérieur au nombre de noeuds de calculs. L'intérêt majeur de Charm++ est qu'il est possible de migrer les *chares* d'un noeud de calcul à l'autre afin d'équilibrer la charge de manière dynamique. Ce concept de *chares* est donc analogue à l'approche de *processus virtuels* que nous développons dans cette thèse dans le cadre particulier de notre solveur pour des matrices compressibles, puisqu'il s'agit d'un regroupement logique d'un ensemble de calculs. Si de nombreuses extensions de Charm++ existent et permettent par exemple de générer des graphes de tâches [56], la granularité d'un *chare* est a priori bien plus grande qu'une tâche de notre modèle de programmation. Des techniques similaires à notre approche qui consiste à répartir les données et donc la charge en fonction de modèles de coûts ont été utilisées dans l'application ChaNGa [55] qui utilise Charm++.

Le vol de tâche a été popularisé par le langage Cilk [49], dont le *work-first principle* assure l'efficacité pour des algorithmes hiérarchiques. Le mécanisme de vol de tâche s'adapte naturellement aux systèmes distribués, même si leur mise en oeuvre s'avère délicate d'un point de vue pratique [43]. Nous montrons dans cette thèse comment ce type de mécanisme peut s'insérer dans un environnement de programmation avec des tâches interdépendantes.

Xkaapi [50] est une bibliothèque basée sur des templates C++ qui permet de décrire des graphes de tâches dans un environnement en mémoire distribuée. La programmation est facilitée par une sémantique séquentielle qui permet au support exécutif sous-jacent de générer automatiquement les mouvements de données et d'effectuer les synchronisations nécessaires implicitement. L'intérêt majeur de Xkaapi est de permettre un équilibrage de charge dynamique basé sur un mécanisme de vol de tâche. A l'instar de notre approche, ces mécanismes de vols sont implémentés à l'aide d'active messages, qui sont des communications permettent de déclencher automatiquement l'exécution d'une fonction lors de leur réception [88]. Contrairement à notre environnement de programmation, Xkaapi se concentre sur un paradigme de type fork-join. L'utilisation d'une synchronisation explicite (*join*) du parallélisme issu des sous-tâches évite cependant d'avoir à détecter automatiquement la terminaison de l'algorithme distribué.

StarPU-MPI [10] est une extension de StarPU qui permet d'exécuter une application écrite à l'aide de StarPU en laissant StarPU-MPI introduire automatiquement des échanges de données entre les processus. Chaque donnée enregistrée auprès de StarPU-MPI est associée à un processus qui en est responsable. Le responsable d'une donnée est chargé d'exécuter toutes les tâches

qui modifient cette donnée, et de la transmettre aux autres processus qui auraient besoin de lire ces données. Cela permet donc de distinguer la mise en oeuvre d'une application distribuée avec StarPU-MPI, et le mapping des données à proprement parler. StarPU-MPI ne permet cependant pas de rééquilibrer les calculs avec des mécanismes inter-noeuds tels que du vol de tâches puisque seul le processus responsable d'une donnée est autorisé à la modifier.

Legion [16] est une bibliothèque qui permet d'exprimer des applications hiérarchiques sur des machines à mémoire distribuée à grande échelle. Le concept le plus novateur de Legion est la structure de *mapper* qui permet de répartir une donnée sur une machine de manière complètement indépendante de l'algorithme, ce qui permet d'exécuter la même application sur une machine à grande échelle ou sur une machine de bureau par exemple. La flexibilité apportée par le *mapper* qui est indépendant de l'algorithme parallèle est à rapprocher de notre notion de processus virtuels. Legion offre des fonctionnalités telles que le vol de tâche, et propose également des mécanismes similaires à l'algorithme de consensus que nous utilisons pour déterminer la terminaison des calculs (voir Section 5.2.4). Bien que Legion s'intéresse en particulier à des applications hiérarchiques, les algorithmes considérés sont généralement bien plus réguliers que notre solveur direct qui implique des données temporaires irrégulières, des synchronisations à grain fin, et une variation des tailles des données.

Enfin, on dénombre un certain nombre de langages ou d'environnements de programmation qui automatisent la parallélisation des calculs sur une machine à mémoire distribuée à partir d'une distribution des données. Il peut s'agir de combiner MPI avec des modèles comme OpenMP dans le cas de OmpSs [46, 27] qui utilise la bibliothèque TAMPI [80] similaire à nos tâches interruptibles pour intégrer les communications *bloquantes* au sein d'un graphe de tâches dont l'exécution est asynchrone. HPF [59] permettait également de distribuer automatiquement un code Fortran annoté en générant automatiquement les communications nécessaires. D'autres approches sont possibles, comme les langages UPC [29], X10 [33], Chapel [31] ou Fortress [84] qui reposent quant à eux sur un environnement de type PGAS [93] dont Gasnet [22] ou OpenSHMEM [32] sont des implémentations. On pourra ici remarquer que le fait de permettre d'assigner les données de manière flexible, voire indépendante du programme lui-même, ne permet souvent pas de rééquilibrer la charge en déplaçant les données si les échanges de données ont été introduits statiquement par un environnement de compilation, par exemple.

Chapitre 4

Etat de l'art

Dans ce chapitre, nous rappelons quels sont les principaux solveurs d'algèbre linéaire, en particulier directs, et les éventuelles techniques que ces solveurs mettent en oeuvre pour rééquilibrer la charge de calcul.

Nombreuses sont les applications numériques qui nécessitent de résoudre des systèmes linéaires. On distingue en particulier deux classes de solveurs : les solveurs directs qui nécessitent de factoriser la matrice associée au système afin de traiter avec grande précision un nombre quelconque de seconds-membres, et les solveurs itératifs où l'on cherche une solution de plus en plus proche de la solution réelle, généralement second-membre par second-membre.

Les matrices qui décrivent les systèmes à résoudre sont soit denses, soit creuses lorsque la majorité des entrées sont nulles. Cette structure influence grandement la façon de stocker les données, et la mise en oeuvre des algorithmes de résolution.

Naturellement, ces structures impactent également la manière de répartir les données dans un contexte d'exécution parallèle à mémoire partagée et à mémoire distribuée. Un solveur direct dense peut voir sa charge de travail répartie grâce à une simple décomposition de la matrice par blocs de même taille, distribués sur des unités de calcul différentes. Les solveurs creux, directs ou itératifs, doivent par contre mettre en oeuvre une stratégie de décomposition plus complexe de par le caractère irrégulier des structures de données en jeu pour le stockage des matrices creuses (renumérotation, agrégation, calcul du remplissage, *proportional mapping*, ...).

Le solveur avec compression utilisé pour la simulation des problèmes de SER au coeur de la problématique de cette thèse est implémenté sur la base d'un solveur dense direct parallèle par bloc. Similairement aux solveurs creux, la compression hiérarchique, en introduisant du déséquilibre de charge entre les blocs matriciels, rend les structures de données plus irrégulières. Il apparaît alors opportun de s'intéresser aux stratégies d'équilibrage des solveurs denses et creux pour architectures à mémoire partagée et à mémoire distribuée, en plus de dresser l'état des lieux des stratégies de rééquilibrage pour les solveurs avec compression.

4.1 Solveurs denses

Lorsque l'on considère des systèmes denses, on se restreint en général à des méthodes directes. La Section 2.2 présente le fonctionnement d'un solveur direct que nous ne rappelons que brièvement ici. L'opération principale consiste à factoriser une matrice dense A sous la forme d'un produit LU avec L et U des matrices triangulaires. L est dite inférieure (tous les éléments au dessus de la diagonale sont nuls), et U est dite supérieure (tous les éléments en dessous de la diagonale sont nuls). Dans le cas particulier d'une matrice symétrique, la

méthode de Cholesky (ou décomposition LL^T) cherche à factoriser la matrice sous la forme d'un produit avec $A = LL^T$. D'autres méthodes de factorisation existent, telle que la décomposition sous une forme QR avec Q une matrice orthonormale $Q^TQ = I$, et R une matrice triangulaire supérieure. Cette décomposition QR est notamment utile pour résoudre des systèmes de type moindre-carrés et offre de bonnes propriétés en termes de stabilité numérique.

Pour des raisons de stabilité numérique, et pour éviter des divisions par des nombres de très petite taille, on associe généralement ces factorisations à une permutation des inconnues du système qui a lieu lors de la phase de factorisation. On parle alors de pivotage. Les systèmes mis en oeuvre dans le cadre applicatif de cette thèse sont spécifiques et ne nécessitent pas de pivotage, et nous ne considérerons donc pas cet aspect, y compris lorsque nous introduirons des techniques de compression.

4.1.1 En mémoire partagée

Il existe un nombre considérable de solveurs denses que nous ne pouvons énumérer ici. En mémoire partagée, les implémentations de l'interface standard LAPACK [9] sont souvent utilisées pour la résolution de $AX = B$. Ces implémentations s'appuient généralement sur une implémentation de l'interface standard pour les opérations d'algèbre linéaire de base BLAS [63]. En guise d'exemple, Netlib et OpenBLAS offrent des implémentations historiques en Fortran et en C des interfaces BLAS et LAPACK. Pour des implémentations plus modernes, les bibliothèques Armadillo et Eigen peuvent également implémenter les interfaces BLAS et LAPACK en plus de proposer leurs propres interfaces C++.

Les concepteurs de processeurs généralistes fournissent *systématiquement* une implémentation optimisée des noyaux BLAS afin d'assurer une exécution efficace de ce type de solveurs. En pratique, ces bibliothèques ne se limitent pas aux noyaux BLAS, et fournissent également tout ou partie des noyaux LAPACK, eux-mêmes spécifiquement optimisés. C'est le cas de la bibliothèque MKL (Math Kernel Library) de Intel, de BLIS et libFLAME pour AMD, ArmPL (Arm Performance Library) pour ARM.

Ces bibliothèques peuvent proposer des implémentations séquentielles et parallèles pour les architectures à mémoire partagée. La MKL, OpenBLAS et Armadillo reposent par exemple sur le standard OpenMP (voir la Section 3.2.1.b) pour implémenter l'interface BLAS en parallèle afin de pouvoir exploiter plusieurs coeurs au sein des noyaux de calcul. De même, une attention toute particulière est souvent portée sur la vectorisation efficace des noyaux de calculs BLAS, afin de tirer le meilleur parti des processeurs généralistes multicoeurs modernes, dont la taille des instructions vectorielles ne cesse d'être étendue.

Les fabricants d'accélérateurs ont également tendance à mettre à disposition des implémentations dédiées pour exploiter au mieux les architectures de calcul proposées. Pour ses solutions à base de GPU, NVIDIA propose les bibliothèques CuBLAS et CuSOLVER. Intel propose également une implémentation spécialisée pour son architecture KNL (Knights Landing).

Des implémentations plus récentes, comme Chameleon [2] ou PLASMA [44], ont pris le parti de reposer sur des modèles de programmation par tâches (voir la Section 3.2.2) pour exploiter les architectures à mémoire partagée. Un moyen pour décomposer une décomposition LL^T consiste à écrire l'algorithme par

bloc 2 avec des appels BLAS, puis à soumettre une tâche par opération BLAS à effectuer en indiquant le mode d'accès aux données pour chaque tâche. Les dépendances entre les tâches sont ensuite automatiquement inférées du mode d'accès aux données par le support d'exécution. Ces implémentations permettent généralement d'exploiter plus efficacement les architectures de calcul multicoeurs de par le caractère plus asynchrone des algorithmes ainsi produits. En outre, les déséquilibres de charge sont souvent atténués par cette approche, sous réserve d'avoir des tâches de coûts calculatoires comparables en nombre suffisant pour occuper pleinement toutes les unités de calcul.

4.1.2 En mémoire distribuée

Lorsque le système à résoudre est réparti entre plusieurs processus, il est nécessaire d'utiliser des méthodes de factorisation adaptées. On retrouve toujours les mêmes algorithmes de factorisation (LU, LL^T et QR), mais des échanges de données sont nécessaires au sein de la phase de factorisation. Il s'agit typiquement d'échanges de données collectifs tels que des diffusions (ou *Broadcast* en anglais), ou des opérations de réduction.

La bibliothèque SCALAPACK [20] offre ainsi une interface similaire aux BLAS pour des systèmes en mémoire distribuée. SCALAPACK repose sur les implémentations de BLACS [35] qui fournit les primitives pour les échanges de données. Au-delà de primitives de communication telles qu'on en trouve aujourd'hui dans MPI, BLACS traite également la question de la répartition des données entre les différents processus.

Un des aspects les plus notables de SCALAPACK est de proposer de distribuer les données selon un motif dit "2D-bloc cyclique", où les données sont organisées en blocs. Les différents blocs attribués à un processus sont entrelacés de telle sorte que les noyaux SCALAPACK tels que PZPOTRF (LL^T) ou PZGETRF (LU) consistent en une succession d'appels tels que PZGEMM (produit matriciel) ou PZTRSM (résolution d'un système triangulaire) qui opère non pas sur des blocs mais sur la totalité de la matrice (ou une sous-partie englobant plusieurs blocs).

Cette distribution 2D-bloc cyclique distribue les blocs de manière régulière, ce qui assure un bon équilibrage en termes de consommation mémoire. Encore plus intéressant, cette distribution permet une répartition optimale des communications et des calculs entre les différents processus [45].

À l'image des algorithmes tuilés qui permettent de mieux exploiter les architectures multicores potentiellement équipées d'accélérateurs [28], combiner ces algorithmes avec des supports exécutifs permet d'exploiter pleinement des machines à mémoire distribuée, avec un parallélisme agressif [2, 24], y compris avec des accélérateurs [25, 10].

Comme pour SCALAPACK, les algorithmes tuilés distribués reposent généralement sur une distribution 2D-bloc cyclique des données qui offre une excellente distribution de la charge de calcul et des communications. Notre solveur direct opérant des blocs compressés (hiérarchiquement ou non) adopte donc naturellement une approche similaire. Il s'agit cependant d'une heuristique car les blocs ne sont pas tous exactement de la même taille, et car la mise à jour d'un bloc ne nécessite pas toujours la même quantité de calcul.

4.2 Solveurs creux

Les systèmes que l'on résout ont souvent une structure issue du problème sous-jacent. En particulier, les problèmes issus de la méthode des éléments finis se traduisent généralement par un système dit *creux*, avec une écrasante majorité des entrées de la matrice de valeur nulle.

Il est tentant de faire des analogies entre les solveurs creux et les matrices compressées, et nous présentons donc brièvement les techniques utilisées pour résoudre des systèmes creux ainsi que les méthodes utilisées pour équilibrer la charge de calcul.

4.2.1 Solveurs creux directs

La conception d'un solveur direct pour des matrices creuses est un problème délicat, en particulier en mémoire distribuée, et il existe plusieurs approches. Le code MUMPS est un exemple d'approche dites multi-frontale, et offre une très grande robustesse numérique [8]. PaStiX [52, 53] et SuperLU [65] adoptent une approche dite super-nodale, qui nécessite une analyse symbolique initiale complexe, mais qui permet d'obtenir une parallélisation particulièrement efficace.

L'équilibrage de charge de ces solveurs, qui traitent des matrices dont la structure est généralement liée à l'application, reposent souvent sur une approche de partitionnement de maillage à l'aide de logiciels comme METIS [58] ou SCOTCH [71]. Ces partitionneurs permettent de répartir les inconnues de manière homogène tout en réduisant autant que possible la taille des *séparateurs* entre les domaines. L'heuristique dite de proportional mapping est également très utilisée pour répartir récursivement les blocs de manière équitable [74]. Les solveurs SymPACK [54] et SuperLU [65] utilisent quant à eux des distributions 1D et 2D bloc-cycliques.

4.2.2 Solveurs creux itératifs

Si les méthodes directes sont robustes, elles sont néanmoins coûteuses et gourmandes en mémoire. Plutôt que d'inverser ou de factoriser la matrice associée au système, une technique consiste à chercher une solution de manière itérative.

Cela signifie qu'à partir d'une solution initiale plus ou moins bonne, on raffine peu à peu la solution jusqu'à obtenir une erreur *satisfaisante*.

Il existe de très nombreuses méthodes itératives [36], parmi lesquelles on trouve classiquement des algorithmes simples comme Jacobi [77], Gauss-Seidel [60] ou les méthodes de relaxation (SOR) [86, 78]. Les méthodes de type Krylov [62] sont également particulièrement utilisées. Parmi celles-ci, on trouve par exemple la méthode du gradient conjugué, ou les algorithmes GMRES [79].

Pour résoudre un système avec une méthode de type Krylov, il suffit généralement de disposer d'opérateurs efficaces pour effectuer le produit de la matrice avec un vecteur, et pour calculer un produit scalaire. Cela permet ainsi de mettre en oeuvre des algorithmes complexes comme un GMRES à partir de ces opérateurs relativement simples [48, 14]. Si une matrice se trouve dans une machine à mémoire distribuée, il suffit alors d'utiliser des opérateurs traitant les matrices à l'aide de MPI par exemple. On peut d'ores et déjà noter que cette méthodologie reste adaptée dans le cas de matrices denses compressibles.

En pratique, il ne suffit pas de disposer d'un simple produit matrice vecteur, car il faut généralement appliquer un opérateur appelé préconditionneur sur le problème afin de réduire le nombre d'itérations nécessaires pour la méthode itérative. Là encore, de nombreuses techniques de préconditionnement existent, mais leur étude est hors du cadre de cette thèse.

Dès lors que la matrice est préconditionnée, la majeure partie du temps de calcul est alors passée dans le produit matrice vecteur et dans les produits scalaires. Pour obtenir un bon équilibrage de charge, il est donc souhaitable que ces opérations soient bien réparties. On pourra donc par exemple répartir les valeurs non nulles de manière homogène entre les processus afin d'obtenir approximativement le même nombre d'opérations sur chacun de ces processus. La qualité de cet équilibrage et la bonne répartition des communications entre les processus dépend cependant fortement de la structure de la matrice creuse.

4.3 Solveurs avec compression

L'apport des techniques de compression est aujourd'hui bien établi au sein des applications [4, 66], mais aussi au coeur de solveurs denses [30, 61] ou creux [73, 7, 6, 34]. Le nombre d'implémentations de solveurs directs utilisant des techniques de compression reste cependant limité.

Leur mise en œuvre sur un système distribué se révèle encore plus complexe, et on observe que l'ensemble des implémentations reposent sur des moteurs d'exécution de tâches. Lizé et al. [66] utilisent la surcouche MPI de la bibliothèque StarPU [13] pour exploiter quelques nœuds de calcul. Une distribution de type round-robin permet de répartir les blocs de la H-matrice de manière uniforme, sans utiliser d'équilibrage de charge. L'utilisation de ces techniques à grande échelle nécessite en revanche des techniques plus évoluées [11]. Les problématiques de déséquilibre de charge apparaissent lorsque l'on cherche à passer à l'échelle. Pei et al. étudient le comportement de différents modèles de programmation vis-à-vis du déséquilibre de charge dans le cadre des matrices low-rank (non hiérarchiques) [70]. Ils étudient en particulier l'apport de CHARM++, dont les processeurs migrables (chares) sont en quelque sens analogues à nos processeurs virtuels [57].

Kriemann propose une méthode [61] permettant une bonne scalabilité en mémoire partagée mais qui nécessite de dérouler le graphe de dépendence. Bien que cela améliore grandement la possibilité de passage à l'échelle, il n'y a pas, à notre connaissance, d'autres efforts que ceux présentés dans cette thèse pour rééquilibrer la charge de solveurs directs hiérarchiques. La stratégie d'ordonnancement rudimentaire mise en œuvre en guise de preuve de concept peut être améliorée à l'aide des nombreuses heuristiques ou stratégies d'ordonnancement disponibles dans la littérature [17, 67]. Des modèles de performance plus élaborés [3, 83], et basés sur un calibrage automatique au sein du moteur d'exécution de tâches [12], permettront de mieux prédire le comportement de l'application.

Chapitre 5

Contributions

Nous pouvons tout d'abord classer les différentes méthodes de redistribution en deux grandes catégories : le rééquilibrage statique et le rééquilibrage dynamique. Si ce dernier permet en général de mieux répartir la charge de calcul, son implémentation est souvent plus complexe et son exécution peut induire des coûts supplémentaires en temps d'exécution. À l'inverse, l'équilibrage statique est souvent plus simple à mettre en place, mais ne permet pas toujours de rééquilibrer complètement la charge de calcul.

Malgré les coûts de développement supplémentaires il peut être intéressant d'utiliser consécutivement ces deux méthodes, en effectuant d'abord un premier équilibrage statique, puis en corrigeant les déséquilibres restant à l'aide d'équilibrage dynamique par la suite.

Nous allons donc d'abord présenter dans la Partie 5.1 des méthodes de rééquilibrage statique, en les appliquant sur la phase de factorisation de notre solveur. Nous évoquerons ensuite dans la Partie 5.2 différentes méthodes d'équilibrage dynamique.

Dans ces parties, nous utiliserons plusieurs métriques pour quantifier le déséquilibre de charge entre les différents processus. La première sera le temps d'exécution, qui correspondra, sauf mention contraire, à l'exécution du processus ayant mis le plus de temps à exécuter ses calculs. Une seconde métrique sera le nombre de FLOPs. Le terme FLOPs, est l'acronyme de FLoating point OPeration, qui peut se traduire par opération en point flottant en français. Cette quantité correspond au nombre de calculs sur des nombres à virgule nécessaires pour effectuer un calcul donné. Nous pouvons également noter l'utilisation des flops par secondes qui représente donc logiquement le nombre opérations exécutées par seconde. Enfin, nous utiliserons la quantité de mémoire consommée par processus, un déséquilibre important de cette métrique pouvant indiquer un déséquilibre dans les deux métriques précédentes, et peut également empêcher l'exécution du calcul, si le processus le plus défavorisé a besoin de plus de mémoire que ne peuvent en fournir les machines de calcul.

Le déséquilibre de charge peut quant à lui être défini de plusieurs façons. Une première méthode est de le définir comme l'écart entre le processus le plus chargé, et celui le moins chargé. Cependant, il peut être plus intéressant de le définir comme l'écart entre le processus le plus chargé et la moyenne de charge des processus prenant part au calcul. En effet, si certains processus reçoivent moins de charge que d'autres, du déséquilibre va être créé, mais ce sont ceux qui sont plus chargés que la moyenne qui posent réellement problème. C'est donc cette dernière définition que nous utiliserons dans cette partie.

Sommaire

5.1 Équilibrage statique 61
5.1.1 Influence du mapping sur les performances 61

5.1.2 Processus virtuels 65
5.1.3 Éviter les cas pathologiques à l'aide des processus virtuels 72
5.1.4 Rééquilibrage mémoire 76
5.1.5 Rééquilibrage guidé par une connaissance a priori des temps de calcul 80
5.1.6 Prédire les temps de calcul à l'aide de modèles de performance 87
5.1.7 Expériences avec une simulation du problème à N-corps (Barnes-Hut) 90
5.1.8 Extraction des métriques 93
5.2 Équilibrage dynamique 96
5.2.1 Délégation de tâches 97
5.2.2 Tâches distribuées 99
5.2.3 Principe du vol de travail 104
5.2.4 Détection de terminaison 110
5.2.5 Valorisation dans un code d'aérodynamique hypersonique 114
5.2.6 Vol de tâche coopératif 116
5.2.7 Vol de tâche one-sided 121
5.2.8 Vol de tâche par rendez-vous 124
5.2.9 Support pour le langage Fortran 126
5.2.10 Bilan 127

5.1 Équilibrage statique

Le rééquilibrage est dit statique quand il s'effectue au début du calcul, souvent sans prendre en compte l'occupation des différentes machines. La répartition se fait alors en estimant le coût des différentes parties des calculs, et en répartissant ce coût sur les processus, en général lors du lancement de ces derniers.

Pour effectuer cette distribution, les matrices calculées sont découpées sous forme de blocs, qui seront eux-mêmes distribués sur les différents processus.

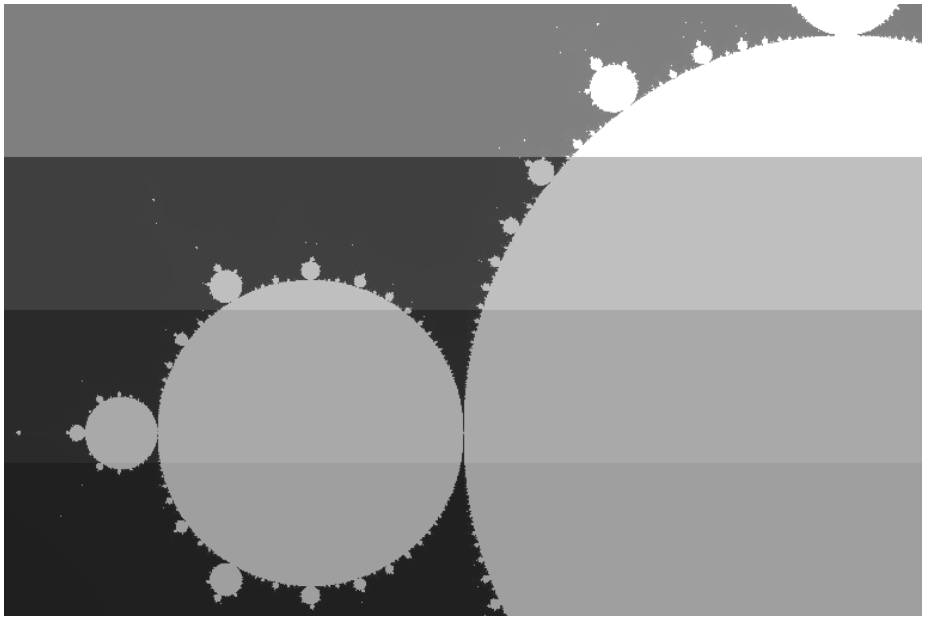


FIGURE 5.1 – Exemple d'équilibrage statique du calcul de la fractale de Mandelbrot

La Figure 5.1 illustre ce type de distribution, sur le calcul de la fractale de Mandelbrot. Cette figure a été découpée en blocs de forme carrée, qui ont été distribués parmi 4 processus, chaque processus utilisant un niveau de gris différent.

En pratique, dans le cadre des calculs sur les matrices, la distribution se fait le plus souvent en utilisant une distribution 2D bloc cyclique. Dans cette partie, nous verrons que cette distribution peut rencontrer des limites quand elle est appliquée sur des matrices hiérarchiques.

Nous présenterons ensuite plusieurs méthodes pour estimer le coût des différentes parties de ces matrices hiérarchiques, et pour exploiter ce coût pour redistribuer la charge de calcul plus efficacement.

5.1.1 Influence du mapping sur les performances

Dans cette partie nous allons présenter plusieurs méthodes de distribution existantes et nous verrons lesquelles sont pertinentes dans le cadre de calculs sur des matrices hiérarchiques.

Nous allons tout d'abord présenter l'intérêt des distributions 2D bloc cyclique par rapport à des distributions 1D bloc cyclique en lignes ou en colonnes. Puis, nous montrerons que dans le cadre de matrices compressées, certaines de ces distributions 2D bloc cyclique peuvent entraîner des déséquilibres en mémoire et en temps de calcul, réduisant la scalabilité des codes associés. Enfin, nous détaillerons une heuristique permettant d'éliminer ces distributions problématiques.

5.1.1.a Expériences préliminaires

distribution	durée
1×16	245s
2×8	250s
4×4	260s
8×2	304s
16×1	397s

TABLEAU 5.1 – Comparaison des temps d'exécution des différentes distributions 2D bloc cycliques pour 16 processus sur TERA1000-1

distribution	durée
1×12	283s
2×6	290s
3×4	263s
4×3	288s
6×2	343s
12×1	431s

TABLEAU 5.2 – Comparaison des temps d'exécution des différentes distributions 2D bloc cycliques pour 12 processus sur TERA1000-1

Les Tableaux 5.1 et 5.2 présentent les temps de factorisation du cas de l'UAV présenté dans la Partie 2.4 sur la machine TERA1000-1 présentée dans la Partie 2.5. Ces cas ont été exécutés sur respectivement 12 et 16 nœuds de calculs, pour l'ensemble des distributions 1D et 2D blocs cycliques possibles. Les distributions 1D sont ici les distributions comportant 1 processus par ligne ou 1 processus par colonne. Nous pouvons tout d'abord noter que les différentes distributions 2D bloc cycliques ont différents temps d'exécution. Ces différences montrent que le choix de la forme de la distribution influe grandement sur les temps d'exécution.

Nous pouvons également voir que le temps d'exécution d'une distribution $n \times p$ n'est pas le même que celui d'une distribution $p \times n$, ce qui montre qu'il n'est pas suffisant de sélectionner une distribution 2D bloc cyclique pour avoir un équilibrage optimal, notamment dans le cas où il n'est pas possible d'utiliser des distributions de type $n \times n$, comme dans le cas du Tableau 5.2 où 12 processus sont utilisés.

Distribution	Volumes de communication		Durée
	Total (Nombre)	Max. par processus (Nombre)	
1×20	1 788 GB (199 614)	52,9 GB (6 046)	559,2 s
20×1	1 937 GB (203 528)	58,3 GB (6 152)	576,7 s
4×5	688 GB (74 366)	14,7 GB (1 654)	439,8 s

TABLEAU 5.3 – Comparaison de la quantité de transferts de données et des performances en utilisant des distributions 1D et 2D blocs cycliques sur 20 nœuds de TERA1000-1 (cas test d'une sphère de 837 000 inconnues)

Distribution	Consommation mémoire par processus	
	Min.	Max.
1×20	3,176 MB	4,526 MB
20×1	3,225 MB	4,488 MB
4×5	3,572 MB	4,054 MB

TABLEAU 5.4 – Comparaison de la consommation mémoire en utilisant des distributions 1D et 2D blocs cycliques sur 20 nœuds de TERA1000-1 (cas test d'une sphère de 837 000 inconnues)

Ensuite, les Tableaux 5.3 et 5.4 montrent respectivement les avantages des distributions 2D bloc cycliques sur les distributions 1D en termes de volume de communications et de consommation mémoire. Le Tableau 5.3 montre que les quantités de communication sont réduites d'un facteur 4 en utilisant une distribution de 4×5 par rapport aux distributions 1D.

En effet, les communications dans le cadre de la factorisation d'une matrice se font sur l'ensemble des processus d'une même ligne ou d'une même colonne. Dans le cadre des distributions 1D, il s'agit donc de l'ensemble des processus, tandis que dans le cas des distributions 2D bloc cyclique utilisant des grilles carrées, il ne s'agit que de $\sqrt{p}$, pour p processus participant au calcul.

Ce déséquilibre en termes de communications explique, au moins en partie, le déséquilibre de temps de calcul, qui est ici de 25%. En effet, une trop grande sollicitation des réseaux peut entraîner une réduction de l'efficacité parallèle des codes de calcul.

Enfin, le Tableau 5.4 montre une réduction de 11% en termes de consommation maximale de mémoire. Cette réduction met en évidence un second type de déséquilibre des distributions 1D. Ce déséquilibre peut empêcher d'exécuter certains cas fortement consommateurs de mémoire. Il est donc également important de le corriger quand cela est possible.

5.1.1.b Des cas pathologiques

Distribution	Temps
1×36	199s
2×18	180s
4×9	173s
6×6	189s
9×4	184s
18×2	249s
36×1	365s

TABLEAU 5.5 – Comparaison des durées de calcul pour 36 processus

En plus des distributions 1D cycliques que nous avons vues dans la Partie 5.1.1.a, d'autres distributions peuvent s'avérer problématiques. Par exemple, dans le Tableau 5.5, nous pouvons voir que la distribution reposant sur la grille carrée 6×6 est plus lente d'environ 8% par rapport à la grille 4×9, qui est la plus rapide.

En effet, contrairement aux matrices denses pour lesquelles toutes les tuiles vont avoir un coût de calcul équivalent, les matrices hiérarchiques vont avoir tendance à concentrer les calculs dans les tuiles proches de la diagonale. Cela est dû au taux de compression qui n'est pas le même sur toute la matrice. Par exemple, de manière générale, nous pouvons observer que les blocs diagonaux sont généralement moins compressés que d'autres blocs plus éloignés de la diagonale, et sont donc plus coûteux en mémoire . Nous pouvons également voir que ces blocs diagonaux sont ceux ayant coûté le plus de temps pour leur factorisation.

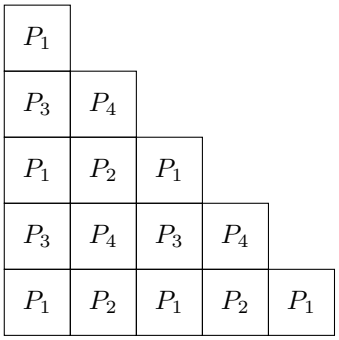


FIGURE 5.2 – Distribution 2D bloc cyclique pour 4 processus

Si, de plus, les grilles des distributions 2D bloc cyclique utilisées comportent autant de processus disposés sur les lignes que sur les colonnes, alors nous pouvons observer que seuls certains processus devront traiter ces blocs diagonaux. Par exemple, la Figure 5.2, représente une distribution 2D bloc cyclique de 4 processus, sur une matrice triangulaire inférieure. Dans ce cas, seuls les processus 1 et 4 vont effectuer des calculs sur les blocs diagonaux. Ces processus devront alors effectuer plus de calculs que les autres.

5.1.1.c Une heuristique pour éviter les cas pathologiques

Afin d'éviter ces différents cas pathologiques, nous pouvons faire varier la forme des grilles 2D cycliques, lorsque cela est possible.

Nous avons vu dans la Partie 5.1.1.b que les distributions 2D bloc cyclique utilisant des grilles carrées équilibrent mal les calculs. En prenant des grilles telles que le nombre de lignes est premier avec le nombre de colonnes, alors tous les processus se partagent les blocs diagonaux, ce qui conduit à un meilleur équilibrage.

Par exemple, la Figure 5.3, représente une distribution 2D bloc cyclique de 6 processus, pour une matrice triangulaire inférieure. Dans ce cas, les 5 blocs diagonaux sont tous attribués à un processus différent, permettant une meilleure distribution des calculs.

Dans la Partie 5.1.1.a, nous avons également vu que plus la forme d'une grille est allongée, plus le volume de communications devient important, augmentant la durée de calcul et diminuant l'efficacité parallèle. En effet, le nombre de processus susceptibles de recevoir un bloc correspond généralement à la somme du nombre de lignes et du nombre de colonnes dans la grille de processeurs $p+q$.

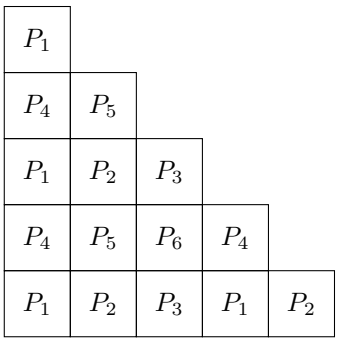
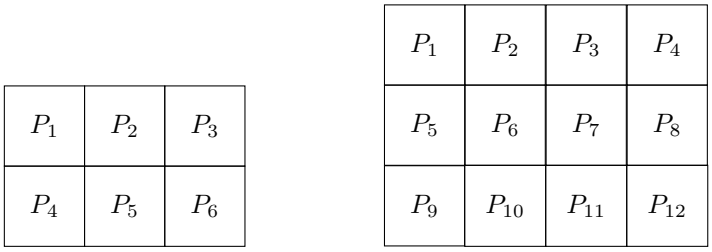


FIGURE 5.3 – Distribution 2D bloc cyclique pour 6 processus



(a) Distribution de forme 2×3 (b) Distribution de forme 3×4

FIGURE 5.4 – Distribution de forme $n \times (n+1)$ pour 6 et 12 processus

Il faut, pour contrer cet effet, sélectionner des grilles les plus carrées possible.

Pour combiner ces deux contraintes, une solution existante est de choisir $p > 2$, le nombre de lignes et q, le nombre de colonnes tels que $q = p + 1$. La Figure 5.4 illustre cette configuration pour 6 et 12 processus. La première grille comporte 2 lignes et 3 colonnes, tandis que la seconde comporte 3 lignes et 4 colonnes.

De cette façon p et q seront premiers entre eux et de valeurs très proches, ce qui permet d'obtenir une répartition des blocs diagonaux sur tous les processus et une faible quantité de communications de par la propriété presque carrée de la grille résultante.

Cependant, cette solution contraint l'utilisateur final à choisir la quantité de processus participant au calcul parmi un ensemble plus réduit.

Pour permettre d'utiliser un nombre arbitraire de processus, sans être impacté par certaines distributions pathologiques, nous voudrions pouvoir dissocier la quantité de processus alloués au calcul de la forme des grilles des distributions. Une solution serait de les décorréler en introduisant des processus "virtuels", et en effectuant la distribution par rapport à ces derniers. Nous décrivons cette notion dans la prochaine partie.

5.1.2 Processus virtuels

Dans cette partie, nous allons présenter la notion de processus virtuels, qui permettent de modifier la distribution des blocs parmi les processus MPI, tout

en gardant le contrôle de la quantité de communications.

En effet, pour un nombre de processus fixé, nous avons vu dans la Partie 5.1.1.b que la distribution des données influe sur la durée des calculs et donc qu'un choix approprié de celle-ci peut permettre de les diminuer.

Dans le cas particulier d'une distribution 2D bloc cyclique sur n processus, une méthode pour faire varier la manière de répartir les données consiste à changer le nombre de processus p situés sur les lignes et le nombre de processus q situés sur les colonnes tels que la relation $p \times q = n$ soit toujours vérifiée. Par exemple une distribution sur 12 processus peut s'exprimer par une grille de 4×3 processus ou de 2×6 processus pour ne citer que deux possibilités.

Cependant pour d'autres quantités de processus, le nombre de possibilités d'organisation en grille 2D bloc cyclique peut être plus restreint : pour 31 processus les deux seules grilles possibles sont 31×1 processus et 1×31 processus, qui sont toutes les deux équivalentes à des distributions 1D, ces dernières provoquant de grands volumes de communications.

Dans la suite de cette section, nous allons donc présenter les différentes propriétés nécessaires à une distribution de données efficace pour un calcul sur des matrices hiérarchiques, avant de présenter une méthode de distribution les satisfaisant. Enfin nous présenterons comment passer d'une distribution à une autre. Ceci permettra en pratique de changer de grille plusieurs fois pendant le calcul, ce qui peut s'avérer crucial lorsque le déséquilibre de charge évolue au cours du temps.

5.1.2.a Présentation des besoins pour une distribution de processus

Pour des raisons de compatibilité avec le code industriel existant, on se limite tout d'abord aux méthodes de distribution de données qui s'effectuent au niveau des tuiles des matrices, de manière similaire à la distribution 2D bloc cyclique. Pour les mêmes raisons, nous nous limitons aux méthodes permettant l'implémentation des différentes méthodes d'accès des distributions 2D bloc cyclique. Ceci nous permettra de nous concentrer sur la création de nouvelles techniques de distribution sans avoir à adapter le code de calcul.

Ces deux contraintes permettent d'utiliser de nouvelles distributions avec la même logique qu'avec des distributions 2D bloc cyclique, et donc avec la même interface, permettant ainsi de réutiliser des algorithmes ayant été conçus pour ces dernières.

S'il n'est pas indispensable que le nombre et le volume de communication soient optimaux, il est cependant préférable que le nombre de processus situés sur une même ligne ou une même colonne soit limité, permettant ainsi de restreindre les communications réseaux qui pourraient sinon limiter la vitesse de calcul.

De plus, les temps de calcul pour mettre à jour une partie de la matrice compressée hiérarchiquement n'étant *a priori* pas répartis de manière égale sur l'ensemble de la matrice, il faut que la méthode de distribution proposée soit assez souple pour prendre en compte ces variations. Cela n'est par exemple pas possible avec la distribution canonique pour les matrices denses, 2D-bloc cyclique.

Enfin, il faut que le surcoût nécessaire pour connaître le propriétaire d'un bloc soit court, pour qu'il reste négligeable devant la durée des calculs opérant sur les blocs. Cela suppose donc de disposer d'une structure de données efficace

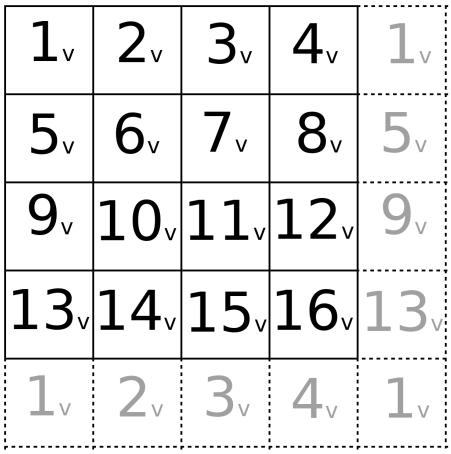


FIGURE 5.5 – 16 processus virtuels disposés sur une grille de 4×4

pour retrouver le propriétaire d'un bloc sans parcourir la totalité des blocs, par exemple.

5.1.2.b Présentation des processus virtuels

Pour résoudre ces problématiques, nous proposons d'étendre la notion de distribution *2D bloc cyclique* en nous appuyant sur des processus dits virtuels.

Ces processus sont dits virtuels car ils ne sont pas liés à des processus MPI, et ne sont utilisés que pour déterminer quel est le détenteur d'un des blocs de la matrice. Cela permet aussi de connaître les participants aux différentes phases de communication lors de la diffusion de ces blocs quand cela est nécessaire.

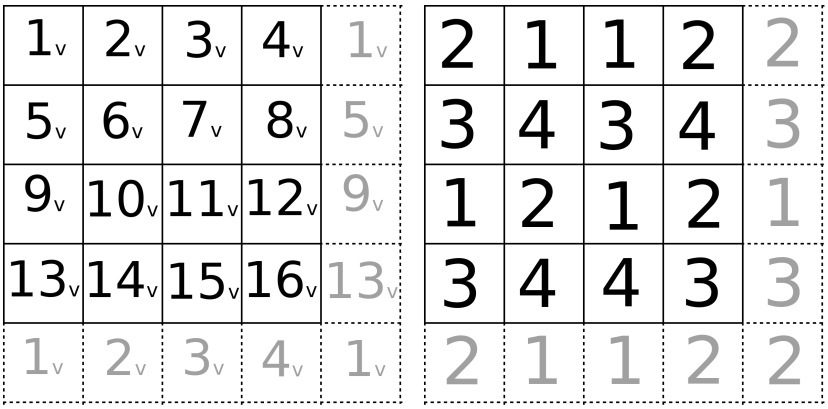
Dans la suite de ce document, nous nommerons *processus physiques* les processus MPI instanciés au début du programme, pour les différencier des *processus virtuels* décrits ici.

Algorithme 3 : Association des blocs d'une matrice avec les processus virtuels

Entrées : *p',q'*, représentant la taille de la grille de processus virtuels
Entrées : *x_ max, y_ max*, représentant le nombre de blocs de la matrice selon l'axe x
Résultat : affectation des processus sur la grille étendue (p',q')

```
1 pour chaque x in [0, x_ max[ faire
2 |   pour chaque y in [0, y_ max[ faire
3 |   |   processus virtuel[(x%p') + ((y%q') × p')] ← bloc(x, y)
4 |   fin
5 fin
```

Comme nous pouvons le voir dans l'Algorithme 3, les données vont être associées selon une distribution *2D bloc cyclique* à des processus virtuels, dont le nombre ne sera pas le même que celui des processus physiquement alloués au calcul. Un exemple de cette distribution est présenté dans la Figure 5.5.



(a) 16 processus virtuels disposés sur une grille de 4×4

(b) Exemple d'une distribution utilisant des processus virtuels

FIGURE 5.6 – 2 représentations de grilles de processus virtuels

Puis, chacun de ces processus virtuels est affecté à un processus physique, comme nous pouvons le voir dans la Figure 5.6b. Par exemple, le processus virtuel 6_v de la Figure 5.6a est affecté au processus physique 4 sur la Figure 5.6b. La correspondance entre les blocs et les processus virtuels étant connue, nous pouvons déduire simplement quels blocs sont associés à un processus en particulier, et quel processus est chargé de calculer un bloc précis. Nous présenterons dans les sections 5.1.4 et 5.1.5 plusieurs méthodes permettant de décider de cette affectation.

Dans la Section 5.1.2.a nous avons vu qu'il est nécessaire d'obtenir la liste des processus situés sur une même ligne ou colonne lors des calculs sur des matrices. D'une manière similaire à la méthode d'obtention des processus physiques impliqués dans les différentes communications dans le cadre d'une distribution 2D bloc cyclique, il est possible d'obtenir la liste des processus virtuels dans le cadre des distributions décrites ici. L'association entre les processus virtuels et les processus physiques étant connue, il est ensuite possible d'en déduire la liste des processus physiques impliqués.

De cette façon, nous pouvons implémenter une interface similaire à celle des distributions 2D bloc cyclique pour les processus virtuels et donc conserver les mêmes appels de fonction dans le code de calcul sans tenir compte du mapping effectif, ce qui facilite grandement l'adoption de ce nouveau type de distribution.

Nous pouvons également noter que si le nombre de processus virtuels est égal au nombre de processus physiques et qu'à chaque processus physique est associé un unique processus virtuel, alors cette méthode de distribution permet de conserver une distribution 2D bloc cyclique.

Cette nouvelle méthode de distribution permet donc d'associer des blocs d'une matrice aux processus physiques à l'aide de l'introduction de la notion de processus virtuels. Elle possède également l'avantage de ne pas devoir assigner de manière individuelle les blocs aux processus physiques grâce à la conservation du principe de cycle. De plus, sa similarité avec les distributions 2D bloc cyclique permet de l'utiliser dans des codes de calcul existants sans les mettre à jour pour prendre en compte une nouvelle méthode. Elle octroie en outre la pos-

sibilité de garder le contrôle sur le volume de communications nécessaires dans le cadre d'une décomposition LU ou LL^T sur architecture à mémoire distribuée, en imposant par exemple une limite sur le nombre de processus physiques par ligne et/ou par colonne lors de l'étape d'association des processus virtuels aux processus physiques. Ainsi, elle respecte tous les pré-requis énoncés dans le paragraphe précédent.

En utilisant une grille de processus virtuels plus grande que celle des processus physiques, comme par exemple dans la Figure 5.6, nous pouvons créer des schémas moins réguliers qu'une distribution 2D bloc cyclique. Par la suite, à l'aide de méthodes de rééquilibrage de charge, nous allons pouvoir déterminer une association entre les processus virtuels et physiques permettant de réduire la consommation de mémoire ou le temps de calcul.

Enfin, nous pouvons noter que cette méthode nécessite premièrement de pouvoir transférer la gestion des blocs d'un processus à un autre. Nous présenterons cette méthode dans la Partie 5.1.2.c. Il est également nécessaire de disposer d'algorithmes permettant de déterminer une distribution optimisant certains critères, comme le temps de calcul ou la mémoire nécessaire par exemple. Nous présenterons ces méthodes dans les Parties 5.1.4 et 5.1.5.

5.1.2.c Passage d'une distribution à l'autre

Nous pouvons tout d'abord noter que tous les critères ne sont pas utilisables à n'importe quelle partie du calcul. Par exemple, une solution se basant sur l'empreinte mémoire des blocs de la matrice est difficilement applicable avant d'avoir effectué l'étape d'assemblage.

Il n'est donc pas toujours possible de partir d'une distribution optimisée, ce qui signifie qu'il faut pouvoir passer d'une distribution à une autre dynamiquement.

Nous allons donc présenter dans cette section un algorithme permettant de passer d'une distribution se basant sur les processus virtuels à une autre.

Nous pouvons voir dans l'Algorithme 4 une première méthode de redistribution. Dans celui-ci, chaque processus va, pour chacun des blocs, vérifier s'il a la charge de ce dernier dans une distribution mais pas dans l'autre. Si c'est le cas alors il va vérifier si c'était la distribution d'origine ou la nouvelle et va respectivement envoyer ou recevoir le bloc.

Une première amélioration de cet algorithme va consister à utiliser des communications non bloquantes. Il s'organisera alors en trois parties :

1. Chaque processus alloue de la place pour les nouveaux blocs,
2. Chaque processus initie les communications nécessaires,
3. Chaque processus achève les communications et libère la mémoire utilisée par les anciens blocs.

La mémoire nécessaire pour ce nouvel algorithme de redistribution est donc, pour un processus donné, de la somme de l'espace nécessaire aux blocs lui appartenant uniquement dans la première distribution, plus la somme de l'espace nécessaire aux blocs lui appartenant uniquement dans la seconde distribution, plus la somme de l'espace nécessaire aux blocs lui appartenant dans les deux distributions.

Dans le pire des cas, si tous les blocs lui appartenant dans la première distribution ne lui appartiennent plus dans la seconde, et vice-versa, la quantité de

Algorithme 4 : Algorithme de redistribution naïf

```
   Entrées : map_old, distribution d'origine
   Entrées : map_new, nouvelle distribution
   Entrées : B, ensemble de blocs à redistribuer
   Entrées : rang, rang du processus courant dans le calcul
 1 pour chaque b ∈ B faire
 2     si map_old(b) = rang ⊕ map_new(b) = rang alors
 3         si map_old(b) = rang alors
 4             owner ← map_new(b)
 5             Envoyer b à owner
 6             Désallouer b ;
 7         sinon
 8             owner ← map_old(b)
 9             Allouer b
10             Recevoir b de owner
11         fin
12     fin
13 fin
```

mémoire nécessaire lors de l'envoi sera la somme de la mémoire nécessaire pour chacune des deux distributions. Cependant, nous avons vu précédemment que la mémoire peut être un facteur limitant pour traiter certains calculs. Dans ce cas, le surplus de mémoire nécessaire à la redistribution peut être supérieur à la mémoire disponible, rendant impossible l'exécution de l'algorithme.

Une seconde amélioration consiste alors à séparer la soumission des envois de celle des réceptions, comme l'illustre l'Algorithme 5. Dans celui-ci chaque processus commence par faire la liste des blocs à envoyer et celle des blocs à recevoir. Puis, il va soumettre l'ensemble des envois, avant de commencer à soumettre les réceptions.

Pour ces dernières, si la mémoire est suffisante, il va allouer suffisamment de mémoire pour recevoir le bloc, sinon il va attendre la complétion d'un ou plusieurs envois et libérer la mémoire associée aux blocs envoyés. Une fois que suffisamment de mémoire a été libérée, il peut reprendre les soumissions.

Enfin, les processus vont s'assurer que tous les envois ont été complétés et la mémoire associée libérée, avant de compléter les réceptions.

De cette façon, la mémoire disponible est immédiatement exploitée, permettant de soumettre les réceptions au plus tôt, et de les délayer lorsque la mémoire du processus est surchargée.

La quantité de mémoire nécessaire pour exécuter l'Algorithme 5 est plus faible que pour l'Algorithme 4. En effet, contrairement à celui-ci, il n'est pas nécessaire d'allouer l'ensemble des blocs dès le début de l'algorithme. Si un des processus peut allouer suffisamment de mémoire pour recevoir le bloc suivant, alors nous pouvons observer que l'algorithme pourra le transférer. Ainsi, si chaque processus possède suffisamment de mémoire pour être capable de recevoir au moins un bloc à la fois, l'algorithme pourra transférer l'ensemble des données sans situations de blocage dûes à un manque de mémoire. Nous pouvons donc en déduire que la quantité de mémoire nécessaire est inférieure à $max(D, F) + B$, D représentant la quantité de mémoire nécessaire à la distribu-

Algorithme 5 : Algorithme de redistribution asynchrone prenant en compte la mémoire

```
   Entrées : map_old, distribution d'origine
   Entrées : map_new, nouvelle distribution
   Entrées : B, ensemble de blocs à redistribuer
   Entrées : rang, rang du processus courant dans le calcul
 1 liste_envoi ← lister_envois(map_old,map_new)
 2 liste_receptions ← lister_réceptions(map_old,map_new)
   /* On soumet les envois                                          */
 3 pour chaque out_com ∈ liste_envoi faire
 4 |   soumettre_communication(out_com)
 5 fin
   /* On soumet les réceptions, en s'assurant d'avoir assez de
      place                                                         */
 6 pour chaque in_com ∈ liste_receptions faire
 7 |   tant que ¬ espace_disponible(in_com) faire
 8 |   |   ended_com ← attendre_une_com(liste_envoi)
 9 |   |   libérer_mémoire(ended_com)
10 |   |   liste_envois ← liste_envois \ ended_com
11 |   fin
12 |   allouer_mémoire(in_com)
13 |   soumettre_communication(in_com)
14 fin
   /* On s'assure que tous les envois sont complétés               */
15 pour chaque out_com ∈ liste_envois faire
16 |   attendre_complétion(out_com)
17 |   libérer_mémoire(out_com)
18 fin
   /* On s'assure que toutes les réceptions ont été complétées
      */
19 pour chaque in_com ∈ liste_receptions faire
20 |   attendre_complétion(in_com)
21 fin
```

tion de départ, F la quantité de mémoire nécessaire à la distribution finale et B la quantité de mémoire nécessaire au bloc le plus coûteux. Cette dernière sera inférieure, dans le pire des cas, à un bloc complet non compressible.

Enfin, une autre solution possible pour régler ce problème de mémoire est d'implémenter les envois et réceptions en utilisant des tâches de communication. Il est alors possible d'insérer des dépendances entre les tâches de réception et celles de calculs sur les mêmes blocs.

Les tâches de communication pouvant être interrompues, l'étape préliminaire de ces tâches peut consister à s'assurer qu'il reste suffisamment de mémoire disponible, ou à s'arrêter en attendant qu'il y en ait assez. Dès lors que la quantité de mémoire disponible est suffisante, la tâche se déroulera de manière habituelle. Dans le cas contraire son exécution est interrompue, ce qui permet d'attendre la terminaison des envois, et donc la libération de la mémoire associée.

5.1.3 Éviter les cas pathologiques à l'aide des processus virtuels

Dans la Section 5.1.1.c nous avons montré qu'il est parfois souhaitable d'adapter le nombre de processus physiques pour éviter certaines situations pathologiques.

En effet, nous avons vu que les blocs diagonaux sont typiquement plus coûteux à mettre à jour que les autres blocs. Si un sous-ensemble de processus doit factoriser tout ces blocs, alors la quantité de calculs qu'ils auront à effectuer sera probablement plus importante que celle des autres processus, induisant donc un potentiel déséquilibre de charge de calculs.

À l'aide des processus virtuels que nous avons présentés dans la Section 5.1.2.b, nous pouvons redistribuer les blocs de la matrice parmi les différents processus virtuels, ainsi que les processus virtuels parmi les processus physiques.

Nous allons maintenant présenter un exemple simplifié d'un calcul déséquilibré, avant de détailler l'utilisation successive des processus virtuels sur celui-ci en vue de rééquilibrer les temps de calcul. Enfin nous examinerons l'efficacité de cette méthode sur des cas réels.

3s				
1s	3s			
1s	1s	3s		
1s	1s	1s	3s	
1s	1s	1s	1s	3s

(a) Durée de calcul nécessaire aux blocs d'une matrice fictive

P_1				
P_3	P_4			
P_1	P_2	P_1		
P_3	P_4	P_3	P_4	
P_1	P_2	P_1	P_2	P_1

(b) Distribution 2D bloc cyclique pour 4 processus

FIGURE 5.7 – Durée de calcul et distribution des blocs d'une matrice fictive

Dans la Figure 5.7a nous pouvons voir une matrice triangulaire inférieure fictive dont les blocs diagonaux sont trois fois plus coûteux en temps de calcul que les autres blocs. Bien que simplificateur, le choix de ce modèle permet d'illustrer correctement les problèmes induits par les distributions 2D blocs cycliques. Dans le reste de cette Section 5.1.3, nous nous appuierons sur cet exemple pour présenter la méthode de redistribution à l'aide de processus virtuels.

processus 1	12s
processus 2	3s
processus 3	3s
processus 4	7s

TABLEAU 5.6 – Durée de calcul pour l'exemple présenté dans la Figure 5.7a, distribué sur une grille 2D bloc cyclique de 2×2 processus, tel que présenté dans la Figure 5.7b

Le Tableau 5.6 présente les temps de calcul de 4 processus, en ayant réparti la matrice de notre exemple en suivant une distribution 2D bloc cyclique, et en ignorant les dépendances entre les différents blocs. Le processus numéro 1 est celui auquel le plus de calcul a été assigné. Il a en effet 12 secondes de calculs à effectuer, tandis que les processus en ayant le moins n'ont que 3 secondes de calculs. On peut également calculer la moyenne des temps de calcul, qui correspondrait à la durée totale de l'exécution si le calcul était parfaitement équilibré. Celle-ci est de 6,25 secondes, ce qui est presque deux fois plus court que ce que doit effectuer le processus 1.

processus 1	6s
processus 2	5s
processus 3	4s
processus 4	5s
processus 5	4s
processus 6	1s

TABLEAU 5.7 – Durée de calcul pour l'exemple de la Figure 5.7a pour un mapping 2D bloc cyclique de 3×2 processus

Dans le Tableau 5.7, nous pouvons voir la durée de calcul pour 6 processus distribués en 2D bloc cyclique selon une grille de 3×2 processus. Ici, la charge est mieux équilibrée que pour 4 processus. En effet, le processus 1 qui est toujours le processus le plus chargé a 6 secondes de calcul pour 4 secondes de calcul en moyenne, soit une différence de 2 secondes.

Nous proposons maintenant d'appliquer l'Algorithme 6 pour distribuer les processus virtuels parmi les processus physiques en utilisant une distribution 2D cyclique correspondant au nombre de processus physiques.

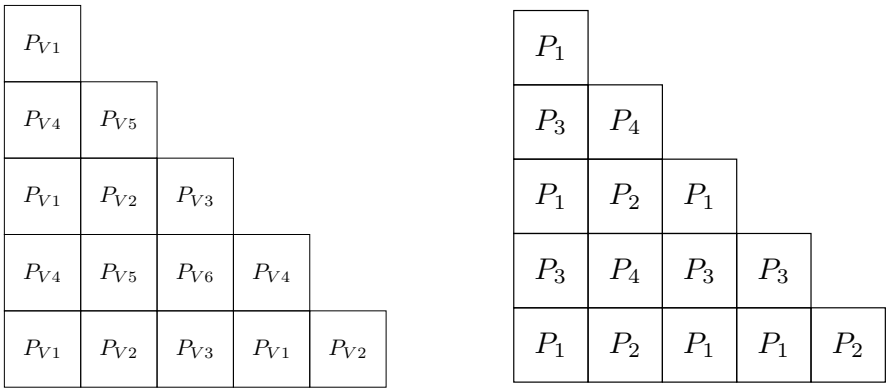
Comme nous pouvons le voir, cette distribution se fait sans faire intervenir de critères, ce qui la rend inapte à prendre en compte d'éventuels déséquilibres de charge. En revanche elle permet d'obtenir une première distribution, qui pourra être rééquilibrée par la suite si besoin.

Dans la Figure 5.8a, nous pouvons voir la distribution des blocs d'une matrice triangulaire inférieure sur 6 processus virtuels en suivant l'Algorithme 3 présenté

Algorithme 6 : Attribution initiale des processus virtuels aux processus physiques

Entrées : *p',q'*, représentant la taille de la grille de processus virtuels
Entrées : *p,q*, représentant la taille de la grille de processus physiques

```
pour chaque x in [0, p'[ faire
    pour chaque y in [0, q'[ faire
        processus physique[(x%p) × ((y%q) × p)] ←
          processus virtuel(x, y)
    fin
fin
```



(a) Distribution des blocs parmi 6 processus virtuels

(b) Distribution des blocs parmi 4 processus en utilisant 6 processus virtuels

FIGURE 5.8 – Distribution des blocs parmi des processus virtuels et physiques

dans la Partie 5.1.2.b.

$P_{V1} \rightarrow P_1$	$P_{V2} \rightarrow P_2$	$P_{V3} \rightarrow P_1$
$P_{V4} \rightarrow P_3$	$P_{V5} \rightarrow P_4$	$P_{V6} \rightarrow P_3$

TABLEAU 5.8 – Résultat de l'Algorithme 6 permettant d'associer des processus virtuels aux processus physiques.

Dans le Tableau 5.8, nous pouvons voir un exemple de l'Algorithme 6, où 6 processus virtuels sont disposés sur une grille de 2×3. Ces processus virtuels sont ensuite distribués sur une grille de 2×2 processus physiques. La Figure 5.8b présente quant à elle la distribution des blocs parmi les processus physiques en suivant cette correspondance entre les processus virtuels et physiques.

Enfin, dans le Tableau 5.9, nous pouvons voir la quantité de calcul assignée à ces quatre processus physiques, lorsque les blocs sont assignés aux processus physiques via des processus virtuels tels que décrit dans le Tableau 5.8. Le processus ayant le plus de temps de calcul assigné est le processus numéro 1, mais

processus 1	processus virtuels 1 et 3	10s
processus 2	processus virtuels 2	5s
processus 3	processus virtuels 4 et 6	6s
processus 4	processus virtuels 5	4s

TABLEAU 5.9 – Durée de calcul des processus en utilisant des processus virtuels

on voit qu'il n'a que 10 secondes de calcul, ce qui correspond à une diminution d'environ 15% par rapport à la distribution 2D bloc cyclique de notre exemple, présenté dans le Tableau 5.6.

Nous pouvons enfin noter qu'une distribution 1D en ligne ou en colonne aurait fait aussi bien dans ce cas. Cependant, comme nous l'avons présenté précédemment, ce type de distribution augmente grandement la quantité de communications et n'est donc pas adapté pour du calcul distribué sur un grand nombre de processus.

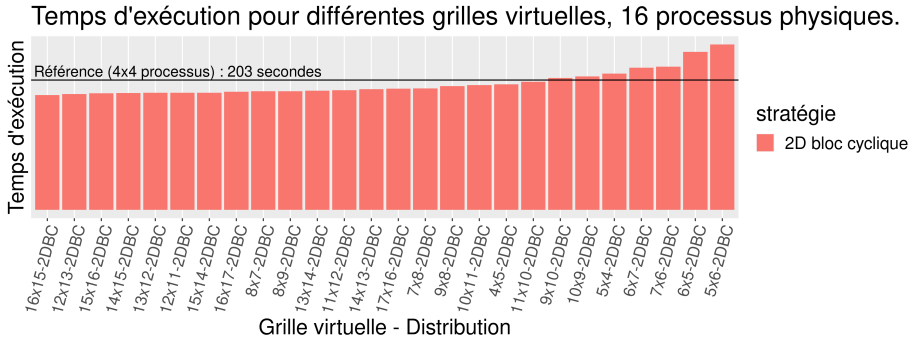


FIGURE 5.9 – temps d'exécution pour différentes grilles virtuelles

Dans la Figure 5.9 nous pouvons tout d'abord observer le temps de différentes exécutions du cas de l'UAV (présenté dans la Partie 2.4) sur la machine TERA1000-1 (présentée dans la Partie 3.1). Ces différentes exécutions utilisent chacune une grille différente de processus virtuels. Pour limiter le nombre de communications induites par la nouvelle distribution, nous n'avons utilisé que des grilles de forme $n \times (n+1)$. Nous pouvons également noter le temps d'exécution en utilisant une grille de 4×4 processus physiques. Ce dernier sera notre temps de référence et est représenté par une ligne noire horizontale.

Nous pouvons voir que le passage à des processus virtuels permet effectivement de faire varier le temps d'exécution. Si pour plusieurs de ces distributions le temps d'exécution subit une augmentation par rapport au temps de référence, nous pouvons voir qu'il peut également être diminué, jusqu'à 13%. Cette diminution confirme l'intérêt des processus virtuels si l'on dispose d'une heuristique pour choisir une distribution appropriée.

Dans la prochaine partie, nous proposerons une méthode pour redistribuer les processus virtuels, permettant ainsi d'améliorer leur distribution parmi les processus physiques et donc de diminuer le temps de calcul.

5.1.4 Rééquilibrage mémoire

La première utilisation de ces processus virtuels consiste à réduire le déséquilibre en mémoire. En effet, un grand nombre de modèles de performances existant dans l'état de l'art se basent sur celui-ci pour réduire le déséquilibre en temps.

On peut également noter qu'un déséquilibre mémoire trop important peut conduire certains processus à dépasser leur limitation mémoire, empêchant d'effectuer le calcul. Une bonne répartition mémoire permet d'empêcher ce dernier phénomène.

Pour corriger ce déséquilibre mémoire, il est *a priori* nécessaire de connaître la quantité de mémoire utilisée par les différents blocs composant la matrice.

Les rangs de ces blocs, et donc leur empreinte mémoire n'étant pas connus avant l'assemblage, il n'est pas possible de rééquilibrer la phase d'assemblage en fonction de la consommation mémoire sans connaissance préalable des données.

En revanche nous allons utiliser cette méthode pour rééquilibrer la phase de factorisation, et nous pourrions également utiliser cette approche pour la phase de résolution, quitte à rééquilibrer une nouvelle fois après la factorisation dans l'éventualité où les rangs des blocs auraient trop changé.

Enfin, il faut noter que les expériences présentées dans la suite de cette partie s'appuieront sur une trace post-mortem, en se limitant aux informations disponibles à la fin de l'étape d'assemblage. De cette façon nous pouvons développer dans un premier temps les différents modèles de rééquilibrage dans un environnement simplifié avant d'implémenter les modèles finaux dans le solveur.

Cette méthodologie basée sur des traces post-mortem permet de se convaincre de la validité de notre approches et de nos modèles avec la facilité que permet de travailler *offline*, c'est à dire en dehors du code plutôt que directement au sein du programme en cours d'exécution parallèle. Une fois les techniques étudiées avec cette approche post-mortem bien validées, nous pourrons mettre en oeuvre ces mêmes algorithmes directement au sein du code parallèle, par exemple entre les phases d'assemblage et de factorisation.

5.1.4.a Méthodologie de rééquilibrage

Nous allons maintenant voir une méthode permettant de distribuer les processus virtuels pour rééquilibrer les calculs lors de la factorisation d'une matrice hiérarchique, en se basant sur la consommation de mémoire.

Pour cela, nous allons d'abord créer une grille de processus virtuels avec un nombre de processus virtuels plus important que le nombre de processus réels.

Afin de garder de bonnes propriétés en ce qui concerne le volume de communications, et d'éviter une explosion combinatoire, nous choisissons de nous limiter à des grilles de processus virtuels avec un ratio entre nombre de lignes et nombre de colonnes limité. Cela permet d'éviter les grilles trop allongées (avec par exemple 2x15 processus plutôt que 5x6).

La première étape consiste donc à générer un ensemble de grilles virtuelles, à partir desquelles nous allons calculer, pour chaque grille, une meilleure redistribution en fonction de critères mémoire.

Puis, comme nous l'avons présenté dans la Section 5.1.3, pour chaque grille virtuelle de taille $p \times q$, nous allons dans un premier temps distribuer les blocs de la matrice parmi les processus virtuels, selon une distribution 2D cyclique.

Puis nous redistribuerons les processus virtuels parmi les processus physiques, en suivant le même schéma qu'une distribution 2D bloc cyclique utilisée en temps normal pour la répartition des blocs parmi les processus physiques.

Algorithme 7 : utilisation d'un algorithme glouton pour rééquilibrer la charge des processus virtuels

Entrées : C, une fonction permettant d'évaluer la somme des coûts des blocs associés à un processus virtuel ou physique
Entrées : P, ensemble de processus définis comme un ensemble de processus virtuels

1 $P_{min} \leftarrow argmin(C(P))$
2 $P_{max} \leftarrow argmax(C(P))$
3 **tant que** $\exists P_V \in P_{max} \mid C(P_V) < C(P_{max}) - C(P_{min})$ **faire**
4 $\quad P_{V\ éligibles} \leftarrow \{P_V \in P_{max} \mid C(P_V) < C(P_{max}) - C(P_{min})\}$
5 $\quad P_{V\ sélectionné} \leftarrow max(P_{V\ éligibles})$
6 $\quad P_{max} \leftarrow P_{max} \setminus \{P_{V\ sélectionné}\}$
7 $\quad P_{min} \leftarrow P_{min} \cup \{P_{V\ sélectionné}\}$
8 $\quad P_{min} \leftarrow min(P)$
9 $\quad P_{max} \leftarrow max(P)$
10 **fin**

À l'aide de l'Algorithme 7, nous allons maintenant déplacer certains processus virtuels, en utilisant un algorithme glouton, pour permettre un rééquilibrage de la mémoire utilisée par les différents processus. La quantité de mémoire associée à un processus virtuel est définie comme la somme de la mémoire nécessaire au stockage de tous les blocs affectés à ce processus virtuel. De la même manière, la quantité de mémoire allouée à un processus physique est définie comme la somme de la mémoire allouée à l'ensemble de ses processus virtuels. Nous pouvons noter d'une part que la quantité de mémoire assignée à un processus virtuel restera fixe, les blocs assignés ne changeant pas. D'autre part, et en revanche, la quantité de mémoire assignée à un processus physique est quant à elle dépendante des processus virtuels assignés, ceux-ci pouvant changer au cours de la redistribution des données.

Jusqu'à ce que cette opération ne soit plus possible (première ligne de l'Algorithme 7), on va migrer un des processus virtuels depuis le processus physique ayant le plus de mémoire allouée, vers celui en ayant le moins.

	Processus virtuels affectés	Mémoire consommée
processus 1	processus virtuels 1 et 3	10Mo
processus 2	processus virtuels 2	5Mo
processus 3	processus virtuels 4 et 6	6Mo
processus 4	processus virtuels 5	4Mo

TABLEAU 5.10 – Quantité de mémoire allouée pour 4 processus en utilisant des processus virtuels

À l'aide du Tableau 5.11 nous allons maintenant illustrer le rééquilibrage décrit ci-dessus avec un cas similaire à celui donné en exemple dans la Section 5.1.3. Cependant, au lieu de considérer le temps de calcul nous considérerons la quan-

tité de mémoire allouée, en Mo. Le Tableau 5.10 indique pour chaque processus quelle est la quantité de mémoire consommée et les processus virtuels qui leur sont affectés au début de l'algorithme.

Au départ, le processus n°1 gère les processus virtuels 1 et 3, le processus n°2 gère le processus virtuel 2, le processus n°3 gère les processus virtuels 4 et 6 et le processus n°4 gère le processus virtuel 5.

1-2 Nous commençons par déterminer le plus petit et le plus grand processus, en termes de quantité de mémoire. Ce sont respectivement les processus 4 et 1.

3 La différence de mémoire entre ces deux processus est de 6 Mo. Il existe au moins un processus virtuel qui coûte moins de mémoire et qui est associé au processus 1 (il s'agit du processus virtuel 3 qui coûte 4 Mo). La condition de la ligne 3 étant vérifiée, on rentre dans la boucle.

4 La liste des processus virtuels éligibles sont les processus virtuels du processus 1 qui gèrent moins de 6 Mo. Ici la liste ne contient que le processus virtuel 3

5 Le plus grand processus virtuel de cette liste est donc le processus virtuel 3

6-7 On transfère le processus virtuel 3 du processus 1 vers le processus 4

8-9 On détermine que les plus petits et plus grands processus sont maintenant respectivement le processus 2 et le processus 4.

10 La différence de mémoire entre ces deux processus est de 3 Mo. Aucun processus virtuel ne gère moins de mémoire, on ne rentre donc pas dans la boucle et on termine l'algorithme.

Étape	ligne	P_1	P_2	P_3	P_4	P_{min}	P_{max}	$P_{V\ éligibles}$	$P_{V\ sélectionné}$
initiale		$\{P_{V1}, P_{V3}\}$	$\{P_{V2}\}$	$\{P_{V4}, P_{V6}\}$	$\{P_{V5}\}$				
1.	1					P_4			
2.	2						P_1		
3.	3								
4.	4							$\{P_{V3}\}$	
5.	5								P_{V3}
6.	6	$\{P_{V1}\}$							
7.	7				$\{P_{V3}, P_{V5}\}$				
8.	8					P_2			
9.	9						P_4		
10.	3								

TABLEAU 5.11 – Évolution des variables dans l'Algorithme 7 en utilisant notre exemple comme entrée

5.1.4.b Efficacité du rééquilibrage mémoire

La Figure 5.10 présente le temps nécessaire à la factorisation du cas de l'UAV, toujours sur la machine TERA1000-1, présentés respectivements dans les sections 2.4 et 3.1. Dans celle-ci nous présentons plusieurs formes de grilles pour l'affectation des processus virtuels. Pour limiter le nombre de grilles, nous nous sommes restreint à des grilles de la forme $n \times (n+1)$, n étant un entier quelconque.

De plus, comme nous l'avons vu dans la Section 5.1.1.c, ce type de forme permet en général une bonne répartition en 2D bloc cyclique, avant même de

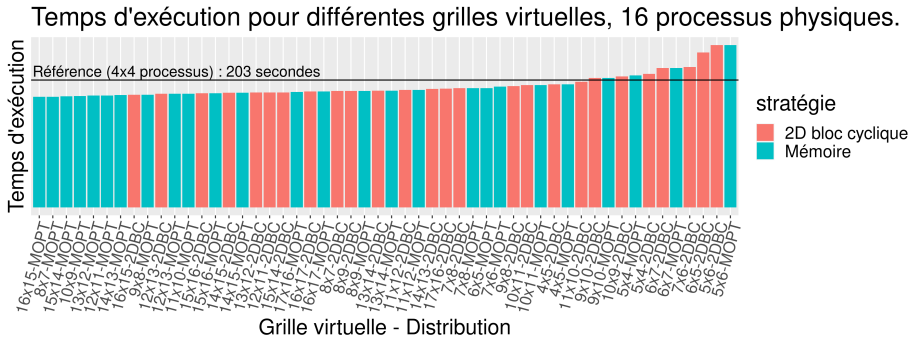


FIGURE 5.10 – Comparaison du temps d'exécution entre des distributions 2D bloc cyclique et des distributions rééquilibrées en mémoire

répartir les processus virtuels parmi les processus physiques. Cette bonne répartition permettra par la suite un meilleur gain de performance lors de l'application de l'algorithme glouton.

Nous pouvons d'abord voir que pour certaines des distributions proposées, l'équilibrage du temps de calcul est moins bon que pour la distribution 2D bloc cyclique de base. En effet, même si la mémoire est correctement équilibrée, il peut exister certains noyaux de calculs plus coûteux que d'autres qui peuvent occasionner un déséquilibre. De plus nous ne prenons pas en compte les coûts occasionnés par le réseau dans notre rééquilibrage, qui peuvent cependant être notables.

Nous avons donc montré que cette méthode d'équilibrage permet de diminuer le temps de calcul dans la plupart des cas, ainsi que de limiter les déséquilibres mémoire. Cette diminution est ici, dans la meilleur des distributions, de 16%. De plus les techniques présentées ne nécessitent pas de modifications importantes des codes de calcul, ce qui permet de faciliter leur implémentation.

Cependant, nous pouvons noter plusieurs limites à cette métrique. Tout d'abord le temps de calcul d'une tuile n'est pas toujours proportionnel à la mémoire consommée. En effet, le calcul d'un bloc faisant intervenir plusieurs noyaux algébriques aux efficacités différentes, et dans des proportions variées, un bloc consommant moins de mémoire qu'un autre ne sera pas nécessairement plus rapide qu'un autre ayant une consommation mémoire supérieure.

De plus l'étape de factorisation des matrices hiérarchiques se présente sous la forme d'un graphe de tâches pouvant avoir des dépendances les unes avec les autres. Comme la méthode que nous venons de présenter se base sur la consommation totale de mémoire des processus virtuels, nous ne pouvons pas les prendre en compte, ce qui nous empêche de détecter des famines, qui peuvent faire baisser l'efficacité parallèle.

Dans la Partie 5.1.5, nous présenterons de nouvelles méthodes d'équilibrage permettant de prendre en compte ces deux limitations.

5.1.5 Rééquilibrage guidé par une connaissance a priori des temps de calcul

Une seconde utilisation des processus virtuels est d'équilibrer directement le temps d'exécution des différentes parties du calcul, et non pas une métrique indirecte comme l'occupation mémoire.

En effet, nous pouvons observer que toutes les tâches n'ont pas le même débit de traitement de données. Il n'est donc pas possible d'estimer correctement la durée de calcul en se basant uniquement sur la quantité de mémoire traitée.

Contrairement à cette dernière, le temps n'est pas une métrique directement disponible *a priori*. Nous verrons dans la Section 5.1.6 comment obtenir cette donnée. Dans cette partie, nous allons à nouveau nous appuyer sur une trace *post-mortem*, nous permettant de séparer la problématique de l'exploitation des temps d'exécution et celle de leur obtention.

Dans cette partie nous commencerons par étudier la répétabilité des performances de calcul, celle-ci étant une condition nécessaire mais non suffisante pour que le temps d'exécution puisse être prédit à partir de la mémoire consommée, cette dernière étant identique entre toutes les exécutions d'un même cas.

Puis nous étudierons une méthode de rééquilibrage en temps ne prenant pas en compte les dépendances entre les différents calculs. Cette simplification permet de réutiliser les algorithmes développés pour l'équilibrage en fonction de la mémoire mais empêche cependant de détecter d'éventuelles zones d'inactivité dans l'ordonnancement.

Nous verrons ensuite d'autres méthodes permettant de prendre en compte ces dépendances, ce qui permettra d'avoir des prédictions du temps total d'exécution plus précises et prenant en compte les zones d'inactivité au prix d'une plus grande complexité de traitement.

5.1.5.a Répétabilité des performances

Dans cette partie nous allons examiner des méthodes permettant de rééquilibrer les charges de calcul en nous basant sur le temps que met chaque mise à jour d'une partie de la matrice hiérarchique lors de l'étape de factorisation.

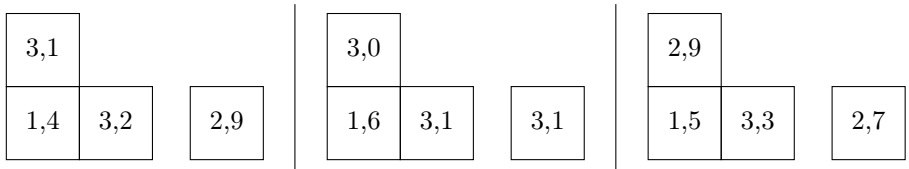
Ces méthodes ne sont pertinentes que si ce temps de calcul nécessaire pour effectuer un calcul à plusieurs reprises reste sensiblement le même.

En particulier, si le temps nécessaire à la factorisation d'un même bloc de la matrice et pour un même cas varie fortement entre plusieurs exécutions, alors nous pouvons en déduire que la durée de calcul dépend d'autres paramètres que ceux disponibles au début du calcul, rendant peu probable l'existence d'un modèle permettant de prévoir les différents temps de calcul.

Nous avons donc exécuté plusieurs fois les mêmes cas, avec les mêmes paramètres. Pour chaque bloc, et à chaque itération nous avons ensuite comparé les temps de calcul entre ces différentes exécutions, et calculé leur moyenne. Enfin nous avons calculé, pour chaque bloc, à chaque itération et pour chaque exécution l'écart entre cette moyenne et les blocs correspondants.

Par exemple, la Figure 5.11 montre les temps de calcul de trois factorisations d'une matrice fictive de 2×2 blocs. À chaque itération nous pouvons observer une variation de ce temps, autour d'une valeur fixe pour chaque bloc.

Le tableau 5.12 montre quant à lui les temps de calcul moyens des différents blocs, ainsi que l'écart à cette moyenne pour chaque exécution. On peut voir,



(a) Exécution n°1 (b) Exécution n°2 (c) Exécution n°3

FIGURE 5.11 – 3 exécutions de la factorisation d'une matrice fictive de 2×2 blocs

Position et itération du bloc	Temps moyen	Variation du temps d'exécution		
Bloc 0,0, itération 1	3,0	+3,3%	0%	-3,3%
Bloc 1,0, itération 1	1,5	-6,6%	+6,6%	0%
Bloc 1,1, itération 1	3,2	0%	-3,1%	+3,1%
Bloc 1,1, itération 2	2,9	0%	+6,9%	-6,9%

TABLEAU 5.12 – Illustration du concept d'écart au temps moyen de factorisation des blocs

par exemple, que le temps de factorisation du bloc 0,0 , à l'itération n°1 varie de $\pm 3,3$s.

La Figure 5.12 représente la distribution de ces variations pour le cas du cône IEEE et de l'UAV sur les machines TERA1000-1 et TERA1000-2. Ces deux cas ont été présentés dans la Partie 2.4, tandis que les machines ont été présentées dans la Partie 2.5. Nous pouvons noter que les graphiques ont été zoomés sur l'axe des abscisses afin de voir plus précisément les données se regroupant autour de la valeur 0, les valeurs ayant été exclues du graphique n'étant pas discernables.

Nous pouvons voir que pour les quatre expériences, les écarts à la moyenne des temps d'exécution restent inférieur à 1, avec un regroupement très net autour de 0. Ceci indique que le temps d'exécution d'une tâche se situera très probablement entre 50% et 200% de son temps moyen, et sera plus probablement à une valeur proche de 100% que d'une extrémité de l'intervalle.

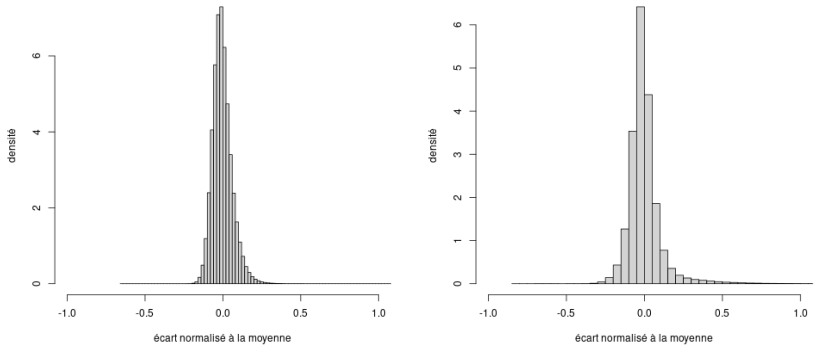
Dans cette partie, nous avons vu que le temps de calcul d'un même bloc, à la même itération est stable d'une exécution à l'autre. Il ne peut donc pas dépendre d'une variable aléatoire ou d'une métrique évoluant d'une exécution à une autre.

Dans les parties 5.1.5.b et 5.1.5.c, nous allons donc supposer qu'il existe des méthodes permettant de prédire les différents temps de calcul, pour développer des méthodes de rééquilibrage s'appuyant directement sur le temps de calcul. Dans ces parties nous nous appuierons sur des traces post-mortem, ce qui nous permettra de séparer la problématique du rééquilibrage de celle de la prédiction du temps de calcul nécessaire à la factorisation des différents blocs de la matrice.

Puis, dans la Partie 5.1.6, nous présenterons plusieurs méthodes permettant de prédire ce temps de calcul.

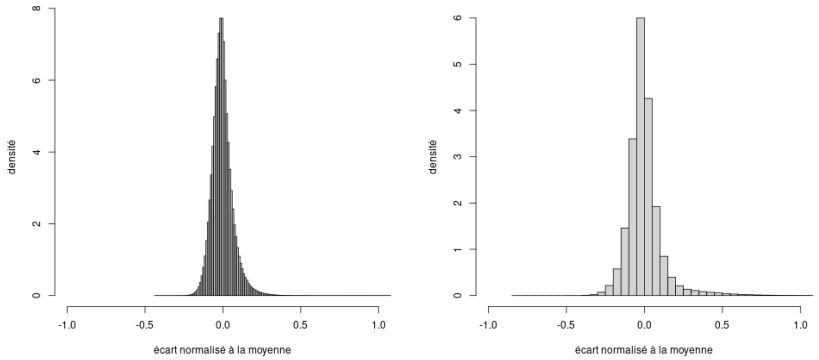
5.1.5.b Rééquilibrage sans prendre en compte les dépendances

La première méthode pour rééquilibrer la charge en fonction des temps de calcul consiste donc à ne pas prendre en compte les dépendances entre les diffé-



(a) Cas du cône IEEE sur la machine Tera1000-1

(b) Cas du cône IEEE sur la machine Tera1000-2



(c) Cas de l'UAV sur la machine Tera1000-1

(d) Cas de l'UAV sur la machine Tera1000-2

FIGURE 5.12 – Distribution des écarts aux temps moyens de factorisation des blocs

rentes parties du calcul. En effet, cette simplification du problème nous permet de réutiliser l'équilibrage mémoire présenté dans la partie précédente, sans avoir à simuler le graphe de dépendance des différentes tâches de calcul, cette simulation étant plus complexe à mettre en place et, comme nous le verrons dans la Section 5.1.5.c, également plus coûteuse à calculer.

Pour réutiliser le rééquilibrage via un algorithme glouton nous allons attribuer un poids à chaque bloc de la matrice. Cependant, au lieu d'utiliser la quantité de mémoire comme métrique, nous utiliserons ici la somme des temps de calcul nécessaire à leur mise à jour.

Ensuite, de manière similaire à l'algorithme présenté dans la Partie 5.1.4.a, chaque bloc va être attribué à un processus virtuel selon une distribution 2D cyclique. Puis ces processus virtuels vont être attribués à un processus physique, selon une distribution 2D cyclique une fois encore.

Enfin, tant que cela peut réduire le déséquilibre de charge, les processus

virtuels sont transférés depuis le processus physique ayant le plus de calcul à exécuter vers celui en ayant le moins.

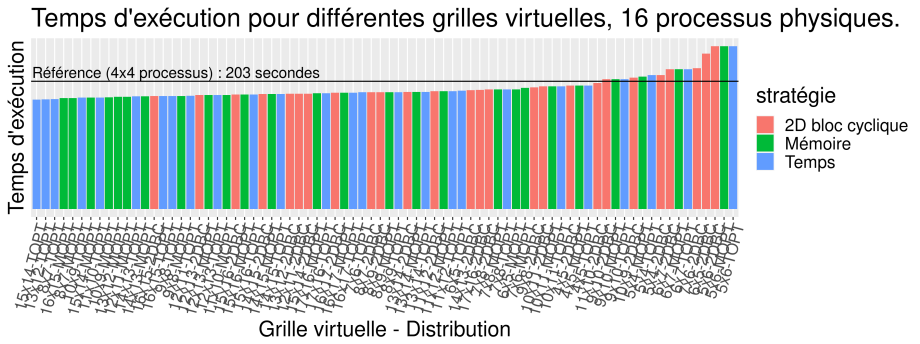
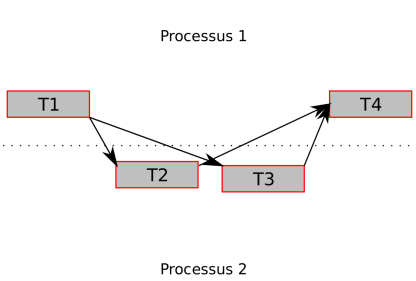


FIGURE 5.13 – Comparaison du temps d'exécution entre des distributions 2D bloc cyclique et des distributions rééquilibrées en temps

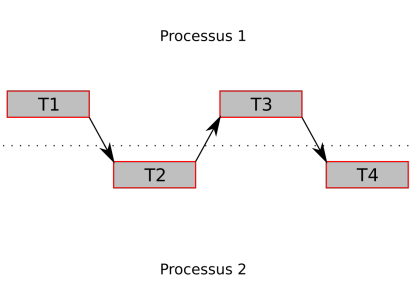
La Figure 5.13 présente une comparaison entre un rééquilibrage glouton de processus virtuels basé sur la durée d'exécution des blocs, un rééquilibrage glouton de processus virtuels basé sur l'empreinte mémoire et un rééquilibrage naïf se basant sur les processus virtuels.

Nous pouvons observer que s'il reste plusieurs cas plus lents que le temps de référence ou qu'avec la redistribution en fonction de la quantité de mémoire, pour une même taille de grille, cette nouvelle méthode de redistribution permet généralement d'obtenir de meilleurs temps que les deux précédentes, la diminution par rapport à la distribution 2D bloc cyclique étant maintenant de 16%.

En effet, cette méthode nous permet de disposer d'une métrique mieux corrélée aux temps de calcul des différents processus qu'avec l'occupation mémoire, et donc de mieux rééquilibrer leur charge. En revanche, les dépendances n'étant toujours pas prises en compte, il peut exister des distributions de processus distribuant la charge de manière égale aux différents processus mais induisant des périodes d'attente dans l'ordonnancement des différentes parties.



(a) Quatre tâches dont la distribution contraint le parallélisme



(b) Quatre tâches formant un calcul séquentiel

FIGURE 5.14 – Deux exemples montrant un mauvais parallélisme malgré un bon équilibrage

Nous pouvons voir sur la Figure 5.14 deux exemples : sur la Figure 5.14a,

nous pouvons voir 4 tâches de longueur égale réparties sur 2 processus monothreadés. Chacun d'entre eux a la charge de 2 tâches, la répartition est donc ici optimale. Nous pouvons cependant voir que les tâches 2 et 3 ont été affectées toutes les deux au processus 2. Celui-ci ne gérant qu'une seule unité de calcul, elles devront être traitées l'une après l'autre, empêchant le calcul d'être effectivement parallèle, alors qu'il aurait pu l'être si elles avaient été distribuées parmi les deux processus.

Sur la Figure 5.14b nous pouvons voir un second exemple, où chacune des tâches dépend de la précédente, empêchant une parallélisation malgré un bon équilibrage.

5.1.5.c Rééquilibrage en prenant en compte les dépendances

Dans la Partie 5.1.5.b, nous avons vu qu'il est possible d'estimer le temps de calcul en se basant sur la durée des différentes tâches. Nous avons également vu que la non prise en compte des dépendances peut conduire à des situations où les temps de calcul sont correctement équilibrés mais sans que suffisamment de parallélisme puisse être exprimé pour exploiter efficacement le calculateur.

Nous pouvons prendre par exemple les dernières étapes de la factorisation, où peu de calculs sont disponibles et où il est particulièrement important de s'assurer que toutes les unités de calcul puissent exécuter des calculs en les distribuant de manière équitable.

Dans cette partie, nous allons étudier la faisabilité de simuler l'exécution des tâches d'un solveur avec leurs dépendances.

Comme le montre l'Algorithme 8, la première étape va consister à obtenir le graphe de dépendance des tâches, ainsi que le temps nécessaire à l'exécution de ces tâches. Ces données pourront être obtenues soit de manière algorithmique, soit en s'appuyant sur la trace d'une exécution post-mortem, de manière similaire à la Partie 5.1.5.b. C'est cette dernière solution que nous avons privilégiée, ce qui nous permet de séparer le traitement des dépendances de leur génération. La racine de ce graphe de dépendances est ensuite insérée dans la liste des tâches prêtes à être exécutées.

Puis, chaque processus le pouvant sélectionne une tâche à exécuter. Les différents processus déterminent quelle est la tâche la plus courte, pour déduire sa durée du temps d'exécution des tâches sélectionnées. Les tâches terminées sont ensuite retirées de la liste de tâches à exécuter.

Enfin, les tâches dont toutes les dépendances ont été entièrement simulées sont insérées dans la liste des tâches prêtes à être exécutées.

Ces opérations sont exécutées jusqu'à ce que plus aucun processus ne possède de tâches prêtes à être exécutées.

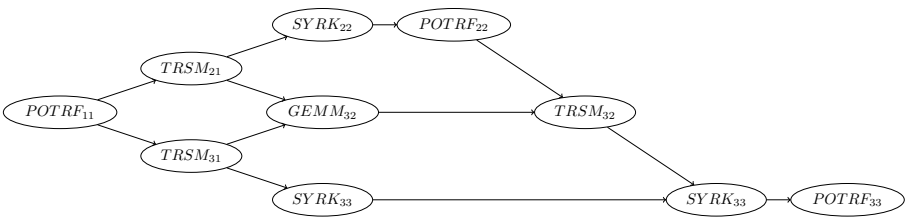


FIGURE 5.15 – DAG des tâches d'une factorisation d'une matrice de 3×3 blocs

Algorithme 8 : simulation du temps d'exécution en prenant en compte les dépendances

Entrées : $liste_{proc}$, la liste des processus prenant part au calcul
Entrées : *dag*, le graphe de dépendances des tâches à simuler

```
1  temps_execution ← 0
2  taches_pretes ← {racine(dag)}
3  tant que taches_pretes ≠ ∅ faire
4  |  taches_exec ← ∅
5  |  pour chaque p ∈ liste_proc faire
6  |  |  si ∃t ∈ taches_pretes | processus(t) = p alors
7  |  |  |  taches_exec ← taches_exec ∪ {select_task(p)}
8  |  |  fin
9  |  fin
10 |  tache_minimale ← min(taches_exec)
11 |  temps_execution ← tache_minimale_temps
12 |  pour chaque t ∈ taches_exec faire
13 |  |  t_temps ← t_temps − tache_minimale_temps
14 |  |  si t_temps = 0 alors
15 |  |  |  pour chaque f ∈ fils(t) faire
16 |  |  |  |  si ndeps(f) = 0 alors
17 |  |  |  |  |  taches_pretes ← taches_pretes ∪ {f}
18 |  |  |  |  fin
19 |  |  |  fin
20 |  |  fin
21 |  fin
22 fin
```

Par exemple, nous pouvons considérer la factorisation d'une matrice de 3×3 blocs. La Figure 5.15 présente le graphe de dépendance des différentes tâches d'un tel calcul, en supposant qu'une seule tâche est nécessaire par bloc.

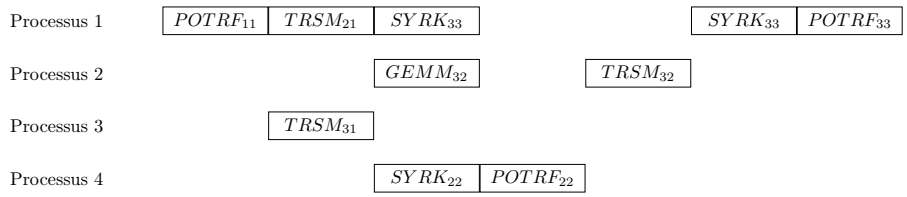


FIGURE 5.16 – Diagramme de gantt de la factorisation d'une matrice de 3×3 blocs par 4 processus disposés sur une distribution 2D bloc cyclique

Comme l'illustre la Figure 5.16, dans le cas d'une distribution de 2×2 processus, l'algorithme de simulation va d'abord sélectionner la tâche $POTRF_{11}$ pour l'exécuter sur le processus 1. Cette tâche va ensuite libérer l'exécution des tâches $TRSM_{21}$ et $TRSM_{31}$, qui vont respectivement être simulées sur les processus 1 et 3. Ces tâches vont à leur tour libérer d'autres tâches, jusqu'à ce que l'ensemble de l'algorithme ait été simulé. Ainsi, en supposant que toutes les tâches durent 1 seconde, nous pouvons déterminer que la durée totale du calcul

est de 7 secondes.

Nous pouvons noter que cet algorithme simule les tâches par partie, comme nous pouvons le voir à la ligne 13. De plus, dans le cas où une tâche sélectionnée n'est simulée qu'en partie, l'algorithme ne garanti pas qu'elle sera sélectionnée à l'étape suivante, et donc permet d'insérer une ou plusieurs tâches avant de terminer sa simulation. Ce comportement n'est pas rencontré pas dans une exécution réelle du solveur mais permet de simplifier l'implémentation du simulateur.

De plus, ni les communications, ni les priorité des différentes tâches ne sont ici prises en compte. Si les premières auraient augmenté le temps de calcul estimé, les secondes l'auraient quand à elles grandement diminué. En pratique, nous observons que le temps estimé est généralement supérieur au temps réel des simulation.

Nombre de processus	Temps réel	Temps estimé	Temps nécessaire à la simulation
12	173s	199,8s	1109s
16	143s	168,8s	1336s
20	112s	139,1s	1566s

TABLEAU 5.13 – Comparaison des temps d'exécution de la factorisation pour 12, 16 et 20 processus avec le temps estimé et le temps nécessaire au calcul de cette estimation

Le tableau 5.13 compare les temps obtenus avec cette méthode, comparés avec les temps nécessaires à la phase de factorisation des cas "cônes IEEE" présentés dans la Section 2.4, pour 12, 16 et 20 processus. Les distributions utilisées sont respectivement de 4×3, 4×4 et 5×4. Nous pouvons voir que le temps d'exécution est prédit avec un ordre de grandeur correct mais que le temps nécessaire au calcul de cette prédiction est un ordre de grandeur de plus que la durée de la factorisation.

En effet, la complexité de cet algorithme est linéaire avec le produit du nombre de processus simulés avec le nombre tâche considérées. En revanche, dans le cadre d'une exécution réelle, la durée d'une exécution idéalement parallélisée est inversement proportionelle au nombre de processus impliqué. Ainsi, plus le nombre de processus considéré sera important, plus la différence entre le temps nécessaire à la simulation par rapport au temps de calcul réel sera important.

Plusieurs améliorations sont possibles pour augmenter la précision, mais qui nécessiteraient une complexification de l'algorithme. Par exemple la prise en compte de la priorité des tâches peut permettre de simuler un ordonnancement des tâches plus fidèle à la réalité, en réduisant le surplus de temps estimé. Cependant cette prise en compte nécessiterait de trier les listes de tâches, et donc d'augmenter à nouveau le temps nécessaire à la simulation.

Enfin, la méthode que nous avons présentée permet uniquement d'établir une estimation du temps nécessaire à une exécution, mais pas de l'améliorer. Pour cela il est nécessaire d'effectuer de nouvelles opérations potentiellement coûteuses, réduisant à nouveau l'intérêt de cette méthode pour du rééquilibrage.

Pour ces raisons, nous pouvons déterminer que cette méthode est en pratique peu intéressante dans le cadre du rééquilibrage en comparaison avec les méthodes que nous avons présentées dans la Partie 5.1.5. En revanche cette mé-

thode peut être intéressante, par exemple lors du développement de méthodes d'ordonnancement, pour estimer leur efficacité sans devoir réserver les ressources nécessaires à une exécution réelle.

D'un point de vue pratique, on peut cependant remarquer qu'il serait intéressant de pouvoir ainsi prédire la durée d'une simulation. En effet, lorsqu'un utilisateur du code a besoin de réserver des ressources de calculs (nombre de processeurs, nombre d'heures de calcul, ...), il est souhaitable de ne pas demander un temps trop important sous peine de ne pas obtenir de temps sur la machine. A l'inverse, si le temps demandé est trop faible, et que le calcul est encore en cours à la fin du temps alloué, la totalité du calcul est alors perdue car la simulation ne peut pas être arrêtée et reprise en plein milieu de la phase de factorisation par exemple. Prédire grossièrement ce temps de calcul est relativement simple avec un solveur dense sans compression, puisqu'il suffit d'évaluer la quantité de calcul avec la relation cubique entre le nombre d'inconnues et le nombre d'opérations, et de considérer que l'on obtient une certaine fraction des performances crêtes de la machine (ou de la vitesse mesurée sur des noyaux tels qu'un produit matriciel par exemple). Ce type d'estimation est en revanche impossible sur un solveur compressé, sauf à utiliser des modèles de performances ou des simulations telles que celles proposées ici. Le dimensionnement rapide des ressources nécessaires, même de manière très grossière, aurait donc un intérêt industriel important, en supposant bien entendu que les ressources nécessaires au dimensionnement soient négligeables devant le calcul lui-même.

5.1.6 Prédire les temps de calcul à l'aide de modèles de performance

Dans la Partie 5.1.5, nous avons étudié plusieurs méthodes permettant d'équilibrer la charge de calcul, en se basant sur une connaissance *a priori* des temps de calcul des différentes parties de la matrice. Pour obtenir ces temps nous avons utilisé une exécution antérieure, les différents temps d'exécution variant peu. Dans cette partie nous allons présenter une méthode s'appuyant sur des modèles de performance pour prédire les temps de calcul en s'appuyant sur la taille des données d'entrées.

Un premier modèle consiste à supposer que le temps d'exécution d'une tâche est proportionnel au nombre d'opérations nécessaires à son exécution. Un tel modèle est simple à implémenter, mais est peu précis dans le cas où les tâches exécutent des calculs différents ayant des performances différentes, comme le montre la Figure 5.17. Cette figure indique, pour chaque tâche de calcul de la factorisation d'un cas de 103000 inconnues, le temps prédit par un tel modèle en fonction du temps réellement nécessaire à son exécution. Les différents types de tâches sont représentés de différentes couleurs : les calculs des GEMMs sont représentés en vert, les TRSMs en orange, les SYRKs en bleu et les POTRFs en rouge. Enfin, une ligne noire indique l'axe $x = y$, permettant d'indiquer la prédiction qu'un modèle théorique ne faisant pas d'erreur effectuerait.

Nous pouvons tout d'abord observer que ce modèle permet une première estimation du temps de calcul : l'ordre de grandeur des temps d'exécution est correctement prédit par le modèle, ce qui permet une première approximation. Nous pouvons également observer que les différentes tâches composent des nuages de points distincts. Par exemple, trois nuages de points séparés ont un temps d'exécution d'environ 0,1 ms. En revanche, les temps prédits sont très différents d'un

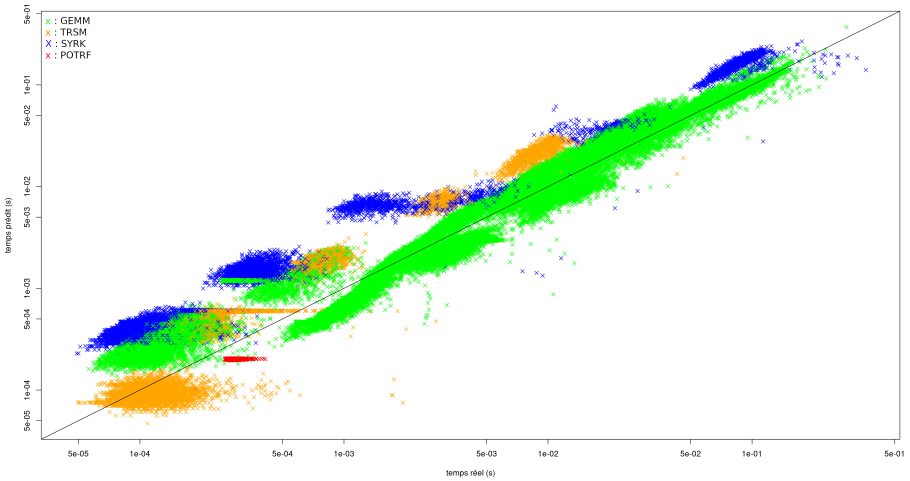


FIGURE 5.17 – Prédiction du temps d'exécution des tâches en supposant un modèle linéaire.

nuage à l'autre.

En effet, les différents noyaux algébriques n'ont pas tous la même efficacité, c'est à dire que le nombre d'opérations exécutées par seconde peut être différent d'un noyau algébrique à un autre. Une première amélioration peut être d'utiliser un noyau par type de tâche. Cette méthode peut permettre de mieux prédire leur temps, cependant nous pouvons voir, par exemple sur les tâches exécutant des GEMM, que nous avons plusieurs groupes de points à l'intérieur d'un même type de tâche, ce qui limite les performances de cette approche.

En effet, pour un même type de tâche, les opérations effectuées ne seront pas les mêmes selon si les matrices manipulées sont compressées ou non. Celles-ci pouvant utiliser une compression mono-niveau, hiérarchique, où ne pas être compressées, le nombre de combinaisons possibles s'avère grand, et la mise en œuvre d'un tel modèle peut donc s'avérer très complexe. Considérons par exemple le produit matriciel (noyau BLAS3 GEMM) qui opère sur 3 matrices ($C = \alpha C + \beta op_a(A) op_b(B)$) : avec A, B et C qui peuvent respectivement être des blocs pleins, des blocs compressés, ou des blocs hiérarchiques, et en considérant les opérateurs op_a et op_b qui peuvent transposer ou non les matrices A et B, on obtient ainsi $3 * 3 * 3 * 2 * 2 = 108$ combinaisons possibles. Toutes ne sont évidemment pas utilisées, et de nombreuses combinaisons sont mises en oeuvre de manière similaire, mais il est donc très délicat de construire un modèle capable de prendre en compte toutes ces situations dont le comportement s'avère souvent totalement différent.

Une seconde amélioration consiste à traiter différemment les temps des différents types de noyaux utilisés à l'intérieur des tâches. Dans celle-ci, chaque noyau est associé à un modèle différent, quel que soit le type de tâche qui l'exécute. De cette façon, chaque modèle peut supposer que le temps d'exécution est proportionnel au nombre d'opérations nécessaires à son exécution, tout en offrant une meilleure précision que le modèle linéaire présenté dans la Figure 5.17.

La Figure 5.18 compare le temps prédit par un tel modèle avec le temps

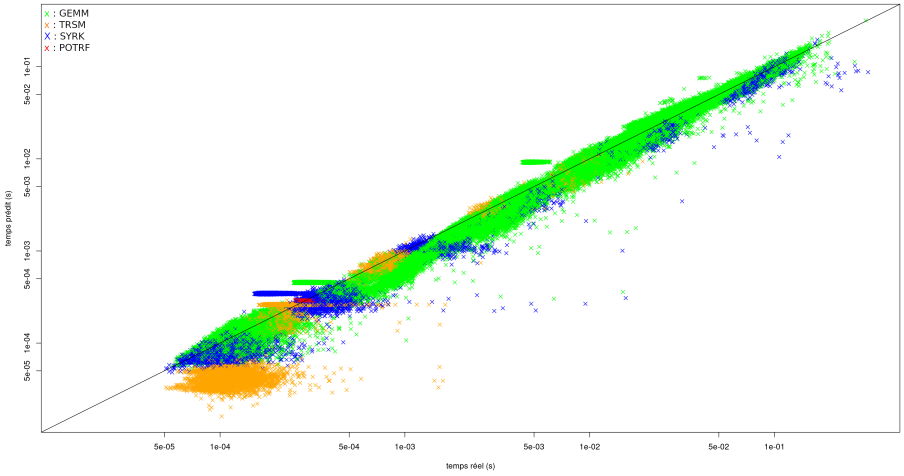


FIGURE 5.18 – Prédiction du temps d'exécution des tâches en supposant un modèle linéaire par BLAS.

d'exécution réel des tâches, toujours pour le cas de l'UAV sur le calculateur TERA1000-1. Nous pouvons tout d'abord noter que les différentes tâches se regroupent maintenant davantage autour de la ligne $x = y$. Nous pouvons également voir que plusieurs groupes de données sont toujours désaxés, indiquant une mauvaise prédiction du modèle pour les tâches de ces groupes.

Une des causes possible à ces imprécisions réside dans l'utilisation de mémoire à accès non uniforme : ceux-ci sont pris en compte lors de l'implémentation des bibliothèques de BLAS, mais pour des matrices de plus petite taille, ou de forme très déséquilibrée, l'efficacité des noyaux peut quand même être plus faible.

Prendre en compte cette baisse d'efficacité peut donc permettre d'améliorer la précision des modèles de performances.

La Figure 5.19 montre la précision d'un modèle de performances différenciant les calculs sur des matrices carrées ou non en plus du type de BLAS, pour les noyaux les plus utilisés dans notre solveur. Nous pouvons voir que la prédiction du temps des tâches s'est à nouveau rapprochée du temps réel, confirmant l'intérêt de cette approche.

Pour pouvoir utiliser ces modèles dans les méthodes de rééquilibrage présentées dans la Partie 5.1.5, il sera nécessaire de pouvoir prédire quelles tâches seront exécutées, ainsi que les noyaux BLAS appelés par ces dernières. La méthode la plus simple pour cela est d'exécuter le solveur, sans effectuer les appels aux différents BLAS, qui constituent la majorité du temps de calcul. Comme il n'est pas nécessaire de simuler différents processus et différents threads, cette exécution est beaucoup plus rapide que celle décrite dans la Partie 5.1.5.c, et peut être exécutée en temps raisonnable.

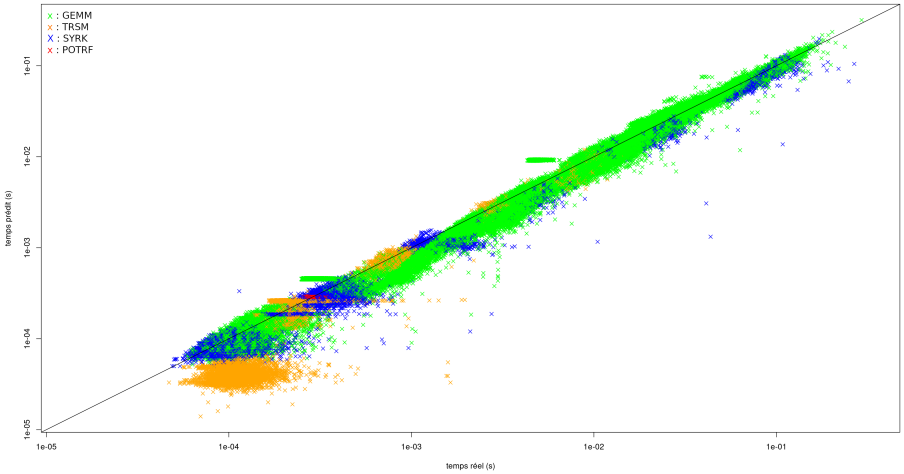


FIGURE 5.19 – Prédiction du temps d'exécution des tâches en séparant les matrices rectangulaires et carrées

5.1.7 Expériences avec une simulation du problème à N-corps (Barnes-Hut)

Pour montrer que notre approche ne se limite pas au problème des solveurs directs compressés hiérarchiquement, nous étudions brièvement le problème du rééquilibrage d'une simulation de type n-corps. Il s'agit d'un problème très différent de notre solveur, mais qui peut bénéficier des techniques de rééquilibrage guidées par des modèles de performances que nous venons d'étudier.

L'implémentation *naive* du problème des n-corps nécessite de calculer $\mathcal{O}(n^2)$ interactions par pas de temps, c'est à dire qu'il faut calculer la force entre chaque paire de particules. Nous résolvons quant à nous ce problème à l'aide d'un algorithme appelé méthode de Barnes-Hut [15]. Cette méthode est basée sur un principe physique qui consiste à ne pas considérer l'interaction entre chaque paire de corps, mais plutôt entre des corps et des groupes de corps suffisamment éloignés. Par exemple, si l'on simule les interactions gravitationnelles entre des étoiles, il sera probablement judicieux de considérer l'interaction entre une étoile et le corps fictif équivalent à l'ensemble des étoiles qui composent une galaxie éloignée. Ce corps fictif sera positionné au centre des masses de la galaxie, et aura une masse équivalente à la somme des masses des étoiles.

Les corps sont donc regroupés spatialement dans une structure hiérarchique de type octree, ce qui permet de déterminer facilement si un groupe de corps est suffisamment éloigné d'un autre corps, tout en stockant des informations comme le centre de masse et la masse de chaque noeud interne de l'octree. Comme l'illustre la Figure 5.20 qui montre le calcul de la collision entre la Voie lactée, et la galaxie d'Andromède, cela permet de traiter des problème de grande taille avec une complexité de $\mathcal{O}(nlog(n))$ au lieu de $\mathcal{O}(n^2)$ pour n corps. Il s'agit ici d'une simulation sur 80000 corps [89], mais les résultats présentés dans ce document ont également été reproduits sur des distributions aléatoires de type PLUMMER pour différentes tailles de problème.

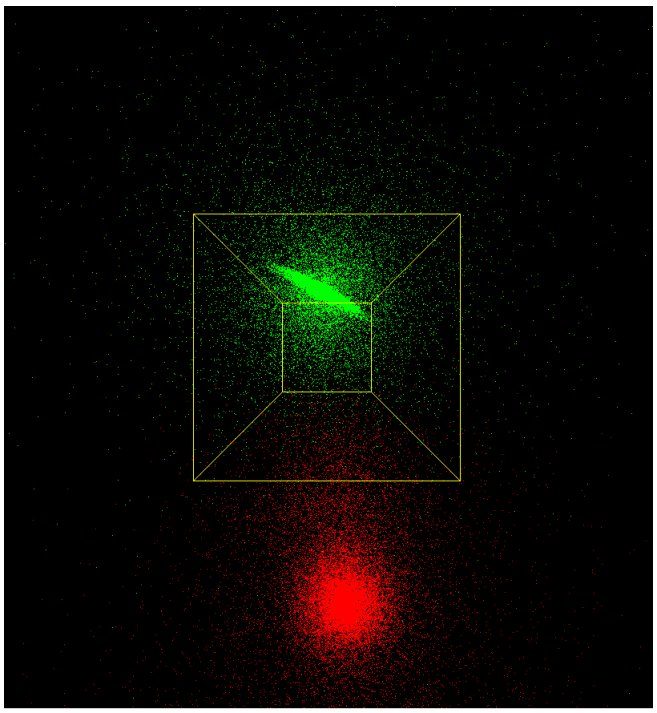


FIGURE 5.20 – Exemple de simulation de la collision entre la Voie lactée et la galaxie d'Andromède

Une étape du calcul des forces consiste à calculer, pour chaque corps C_i, l'interaction entre C_i et tous les autres corps. Plutôt que d'itérer sur l'ensemble des corps, on parcourt l'octree. Si C_i est suffisamment éloigné du corps équivalent à un noeud interne de l'octree, on se contente de calculer l'interaction entre C_i et ce noeud, sans inspecter les sous-parties de l'octree. Il s'agit donc d'un algorithme hiérarchique qui, comme pour notre solveur direct avec compression hiérarchique, nécessite un rééquilibrage rendu d'autant plus délicat que les corps sont en mouvement, ce qui ne permet pas de partir d'un bon équilibrage initial et de s'y tenir. Nous considérons donc le problème qui consiste à rééquilibrer la charge dynamiquement entre les différentes étapes (ou après un certain nombres d'étapes).

Le problème consiste à répartir les corps C_i, indexés selon un Z-parcours de l'octree, entre les différents processus. Le code utilisé dans cette partie se base sur une version écrite par Cédric Augonnet, et a été parallélisé dans le cadre de cette thèse. Chaque processus possède une copie de tous les corps simulés, et calcule les forces appliqués sur ceux dont il a la charge. À chaque itérations, les différents processus commencent par reconstruire l'octree, avant répartir la gestion des différentes particules. Puis les forces d'attractions entre les différents corps et les octrees sont calculés, avant de mettre à jour leur vitesses et leur positions. Enfin, les nouvelles positions des objets sont envoyés aux autres processus pour les utiliser lors des itérations suivantes.

La Figure 5.21 montre ainsi la comparaison entre la durée minimale et maximale sur les différents processus dans les zones hachurées, pour les différentes

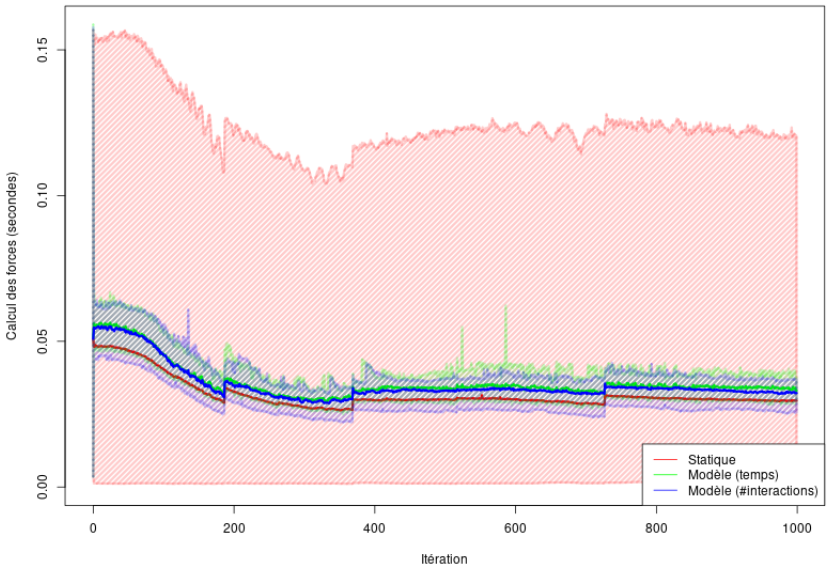


FIGURE 5.21 – Déséquilibre de charge lors de la simulation de la collision de la Voie lactée et Andromède selon la méthode de rééquilibrage

stratégie de rééquilibrage. Il s'agit de 1000 itérations de la simulation de la collision de la Voie lactée et d'Andromède sur 20 processus de 12 threads avec la partition SKYLAKE de la machine INTI (voir Section 2.5). La courbe pleine indique la moyenne des temps de calcul sur l'ensemble des processus, et correspond à un équilibrage parfait.

La couleur rouge indique les performances avec un équilibrage statique avec un nombre équivalent de corps par processus. On constate un écart important entre le processus le plus chargé et celui le moins chargé (avec un écart d'un facteur 3x), et le processus le plus chargé nécessite significativement plus de temps que la moyenne.

En vert, on cherche à répartir les corps pour avoir un temps d'exécution équivalent. Comme pour les modèles de performances utilisés dans le solveur direct hiérarchique, cela suppose que l'on a réussi à capturer le temps nécessaire au calcul de l'interaction d'une particule donnée avec l'ensemble de l'octree. On constate alors un équilibrage particulièrement efficace puisque les courbes de la durée minimale, maximale, et moyenne d'une itération sont presque confondues. L'ensemble de la simulation est donc plus rapide puisque la durée d'une itération correspond *in fine* à la durée maximale.

En bleu, on répartit les corps pour avoir un nombre d'interactions calculées équivalentes. Cela ne suppose donc pas d'avoir mesuré préalablement la durée du calcul pour une particule lors d'une itération précédente. Là encore, on constate une amélioration très significative de la qualité de l'équilibrage de charge. Cette approche illustre ce qu'il est possible de faire si l'on peut émuler une exécution avec une bonne prédiction du nombre d'interactions à calculer, à l'instar de ce que l'on pourrait faire pour prédire le nombre d'opérations nécessaires

pour mettre à jour un bloc de matrice compressé hiérarchiquement dans notre solveur. On pourra par ailleurs remarquer qu'il est parfois difficile de mesurer effectivement le temps de calcul lié à une opération, soit à cause du surcoût induit, soit parce que cette information n'est tout simplement pas accessible. Si l'on considère par exemple des calculs concurrents sur un même GPU, la durée réelle de ces noyaux n'est plus une information pertinente.

On remarquera un léger écart entre les moyennes des temps de mise à jour avec les différentes stratégies qui s'explique par le surcoût nécessaire au rééquilibrage. Ce surcoût moyen se justifie ici par le gain très important dans le temps total d'exécution qui correspond au temps du processus le plus long. Il serait possible de le réduire en ne rééquilibrant pas les calculs à chaque itérations, ou en effectuant ces opérations de rééquilibrage de manière plus asynchrone.

Ce problème des n-corps simulé à l'aide d'une méthode de Barnes-Hut est donc une application pour laquelle les techniques discutées dans ce document peuvent avoir un intérêt particulier. On notera que cela permet également de considérer des problèmes tels que la granularité de prédiction des performances avec des blocs de corps plutôt que des corps pris individuellement pour réduire le surcoût de modélisation. On pourra concevoir des modèles de performances sur des noyaux d'algèbre linéaires sur des noeuds intermédiaires de la matrice hiérarchique plutôt que sur les feuilles. On peut également s'interroger sur la nécessiter de rééquilibrer le calcul à chaque étape, ou de manière régulière, ainsi que sur la conception de mécanismes pour détecter le déséquilibre de manière distribuée.

5.1.8 Extraction des métriques

Dans les Parties 5.1.5.b et 5.1.5.c, nous nous sommes appuyés sur des traces post-mortem pour développer les méthodes présentées. Ces traces nécessitent de mesurer la quantité de calcul nécessaire à l'exécution des tâches calculant la factorisation des blocs de la matrice.

Une première méthode est d'indiquer, lors de la soumission des tâches, une variable servant de compteur, et dans laquelle le runtime va renseigner le temps d'exécution ou la quantité d'opérations qui ont été nécessaires à l'exécution de la tâche.

Cette variable peut ensuite être partagée sur plusieurs tâches, et indique donc le coût de l'ensemble des tâches auquelle elle a été associée, et non plus une seule tâche.

Dans les systèmes STF récursifs, il est possible de déterminer, lors de la soumission d'une tâche traitant un bloc entier, quelle est la position de bloc et de l'indice de l'itération actuelle. Le runtime peut quant à lui associer automatiquement le compteur d'une tâche fille avec celui de la tâche parente, permettant donc d'associer un compteur par tâche exécutée. Il est alors aisé d'associer un compteur par bloc et par itération, comme le montre l'Algorithme 9, ce qui permet d'examiner les coûts d'exécution des différentes parties du solveur.

Pour des raisons d'efficacité, il peut être intéressant de soumettre le calcul des blocs de manière asynchrone. Dans ce cas, il est plus complexe de déterminer lors de la soumission des tâches à quels blocs leur calcul va contribuer. Il est donc difficilement envisageable d'indiquer à ce moment-là quels sont les compteurs affectés à un bloc en particulier.

Algorithme 9 : Exemple d'utilisation de compteurs de performances dans un code STF

Entrées : *tiles*, un tableau de dimension $nb\_iter \times nb\_iter$ contenant les tuiles de la matrice
Entrées : compteurs, un tableau de dimension $nb\_iter \times nb\_iter \times nb\_iter$, contenant les compteurs de performances

```
1 pour i ∈ [0; nb_iter] faire
2 |  pour x ∈ [i; nb_iter] faire
3 |  |  pour y ∈ [x; nb_iter] faire
4 |  |  |  submit_bloc(blocs, x, y, i, compteurs[x + y * nb_iter + i *
  |  |  |   nb_iter^2]
5 |  |  fin
6 |  fin
7 fin
```

Nous proposons donc de mettre en place une interface permettant à l'utilisateur du runtime d'indiquer quels sont les compteurs de performances utilisés à un instant donné. Le runtime met alors en place un contexte, permettant d'associer toutes les tâches soumises dans celui-ci avec le compteur indiqué.

Algorithme 10 : Exemple d'utilisation de compteurs de performances dans un code asynchrone

Entrées : *tiles*, un tableau de dimension $nb\_iter \times nb\_iter$ contenant les tuiles de la matrice
Entrées : compteurs, un tableau de dimension $nb\_iter \times nb\_iter \times nb\_iter$, contenant les compteurs de performances

```
1 pour i ∈ [0; nb_iter] faire
2 |  pour x ∈ [i; nb_iter] faire
3 |  |  pour y ∈ [x; nb_iter] faire
4 |  |  |  set_async_timing_counter(compteurs[x + y * nb_iter + i *
  |  |  |   nb_iter^2])
5 |  |  |  async_submit_bloc(blocs, x, y, i)
6 |  |  |  unset_async_timing_counter()
7 |  |  fin
8 |  fin
9 fin
```

En pratique, comme le montre l'Algorithme 10, l'interface prend la forme de deux fonctions, indiquant l'entrée et la sortie du contexte, ainsi que le compteur manipulé. De cette façon il est possible de dissocier la mise en place des compteurs de performances avec la soumission des tâches, ce qui permet d'examiner le temps d'exécution des tâches soumises de manière asynchrone.

Enfin, nous pouvons noter que ces compteurs peuvent être utilisés dans d'autres cas que pour la génération des traces post-mortem. Par exemple, ils peuvent être utilisée pour connaître le temps total d'exécution, et donc pour

déterminer *a posteriori* l'efficacité de l'exécution.

Pour cela il suffit de positionner un compteur de performances au début de la soumission, et de le retirer à la fin de celle-ci, ce qui lui permet donc d'être associé à l'ensemble des tâches de calcul. Cependant, nous pouvons remarquer que l'utilisation d'un unique compteur ne permet pas de mesurer le temps d'exécution de plusieurs niveaux simultanément.

```
1 struct performance_counter
2 {
3     double *counter;
4     struct performance_counter *parent;
5     int nb_refs;
6 };
```

FIGURE 5.22 – Structure de donnée utilisée pour représenter les compteurs asynchrones récursifs

Pour cela, il est nécessaire de créer une hiérarchie de compteurs. Au lieu d'utiliser une unique variable en tant que compteur, nous avons mis en place une structure de donnée présentée dans la Figure 5.22. Cette structure se compose de trois membres : un pointeur vers la variable indiquée par l'utilisateur, un pointeur vers la structure parente et enfin le nombre d'objets qui utilisent ce compteur de performances. Ces objets sont soit les tâches utilisant directement ce compteur de performances, soit d'autres compteurs qui sont rattachés à un ensemble plus restreint de tâches.

Quand plus aucun objet n'utilise un compteur de performances p, la valeur contenue dans le compteur peut être ajoutée au compteur père et la mémoire utilisée par le compteur p peut être libérée. De cette façon il est possible de garder la même interface que pour les compteur asynchrones, tout en examinant les performances de la matrice à plusieurs niveaux de profondeur simultanément.

5.2 Équilibrage dynamique

Dans la Partie 5.1, nous avons présenté plusieurs méthodes statiques permettant d'effectuer un équilibrage des calculs et de la mémoire. Dans cette partie, nous allons maintenant présenter des méthodes d'équilibrage dynamique. Contrairement à l'équilibrage statique, un équilibrage dynamique ne va pas effectuer une distribution *a priori* des différents calculs à effectuer parmis les processus.

Un équilibrage dynamique quand à lui va rééquilibrer les processus au fur et à mesure des calculs, souvent en s'appuyant sur la charge de ces processus. Si ce type d'équilibrage est en général plus complexe à implémenter que le rééquilibrage statique, il permet de prendre en compte des petites variations de performances difficilement prédictibles.

En effet, le rééquilibrage statique impose de mettre en place des modèles permettant de prédire le coût des différentes parties du calcul, ce qui peut être complexe sur certains types d'opérations, par exemple sur les méthodes itératives. De plus, les différents threads participant aux calculs peuvent interférer les uns avec les autres, dégradant les performances de manière difficilement prédictible, et réduisant donc la précision des modèles de performance.

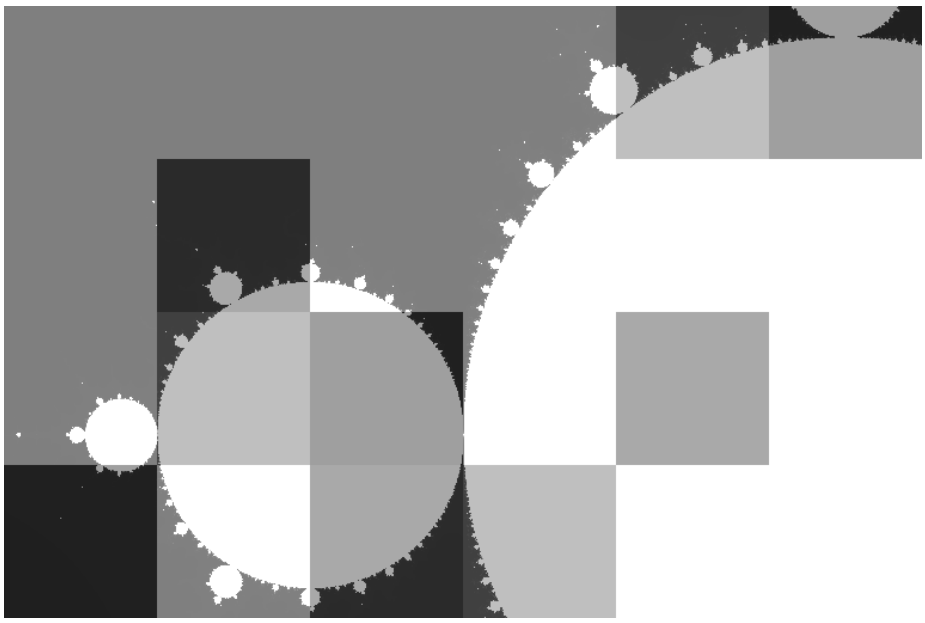


FIGURE 5.23 – Exemple d'équilibrage dynamique du calcul de la fractale de Mandelbrot

En revanche, un équilibrage dynamique va agir de manière plus opportuniste, et donc privilégier les transferts de calculs quand les processus seront surchargés ou au contraire en situation de famine. Par exemple, la Figure 5.23 présente le même calcul de fractale que nous avons présenté au début de la Partie 5.1, en ayant soumis l'ensemble des calculs sur le processus 1, et en ayant trois autres processus exécutant certains de ces calculs.

Dans cet exemple nous pouvons voir que le processus 1, qui correspond aux

blocs les plus clairs, a calculé la majorité des blocs de la fractale. En effet, les équilibrages dynamiques peuvent nécessiter un surcoût pour transférer les calculs, ce qui entraîne ici un déséquilibre de charge, la quantité de calcul nécessaire aux différents blocs étant faible devant ces coûts de communication, du fait de la faible profondeur utilisé dans le calcul de cette fractale.

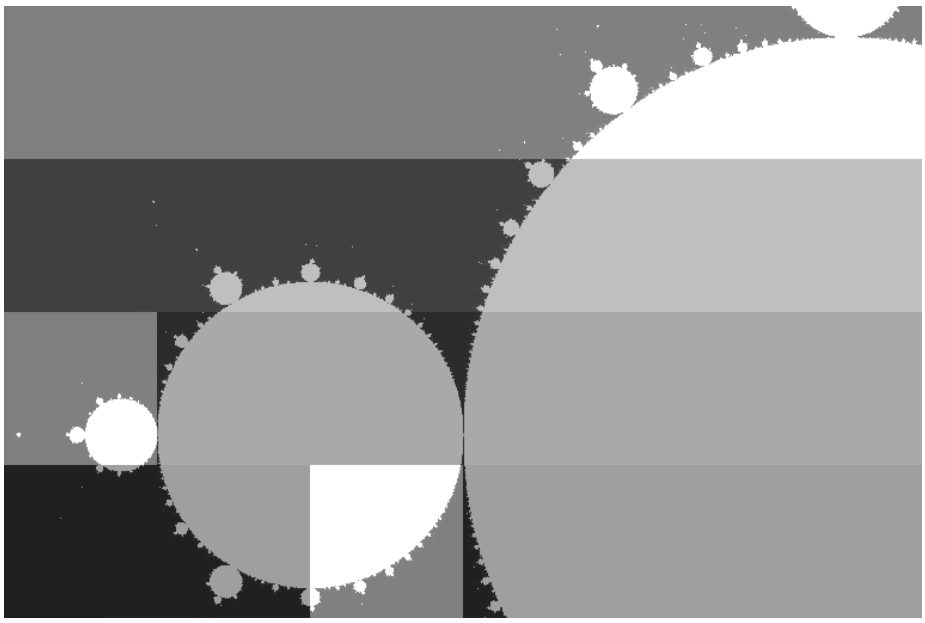


FIGURE 5.24 – Exemple de rééquilibrage hybride du calcul de la fractale de Mandelbrot

Comme le montre la Figure 5.24, il peut en revanche être intéressant de partir d'une première distribution statique, même simpliste, pour ensuite effectuer un rééquilibrage dynamique. Dans ce cas, chaque processus a un ensemble de blocs à calculer, et n'exécute les calculs soumis par les autres processus qu'en cas de famine.

Dans les Parties 5.2.1 et 5.2.2 nous allons maintenant évoquer deux méthodes d'équilibrage dynamique, que nous n'avons pas développées durant cette thèse. Leur présentation permettra cependant d'introduire des concepts utilisés dans la suite de ce document.

Nous présenterons ensuite plusieurs prérequis pour effectuer un rééquilibrage dynamique, ainsi que des méthodes permettant de satisfaire ces prérequis. Enfin nous explorerons plusieurs types de vol de tâches, qui est l'une des techniques possibles permettant d'effectuer des rééquilibrages dynamiques.

5.2.1 Délégation de tâches

Lorsqu'un processus est responsable de trop de calculs, une approche possible consiste à déléguer la responsabilité de certains calculs à d'autres processus a priori moins occupés.

On peut supposer dans un premier temps que le concepteur de code est capable de déterminer facilement que le processus sera trop chargé, alors que

d'autres ne le sont pas. Si cette hypothèse est envisageable, par exemple lors d'une phase d'initialisation qui peut être difficile à programmer de manière distribuée, cette méthode n'est probablement pas adaptée à l'utilisation d'un grand nombre de processus. De plus elle ne correspond pas au cas où le déséquilibre n'est pas prévisible, par exemple dans le cas de variation de performance des noyaux.

Algorithme 11 : Exécution ou délégation d'une tâche locale

```
1 si is_eligible(t) alors
2 |   w ← select_worker(t);
3 |   send_task(t,w);
4 |   send_args(t,w);
5 |   receive_result(t,w);
6 |   release_deps(t);
7 sinon
8 |   exec(t);
9 fin
```

Algorithme 12 : Réception et exécution d'une tâche déléguée

```
  /* attend de recevoir une tâche t d'un processus w        */
1 w,t ← wait_incoming_task();
2 receive_args(t,w);
3 exec(t);
4 send_result(t,w);
```

Cela s'intègre cependant de manière naturelle dans un modèle de tâches avec des dépendances de données implicites, y compris entre des nœuds distants (STF), et permettant donc de gérer simplement les communications MPI. Dans ce cas l'exécution distante sera synchronisée avec le reste des calculs par l'intermédiaire des échanges de données qui induisent des dépendances implicites.

L'Algorithme 11 illustre de manière simplifié l'utilisation de ces tâches déléguées par les processus. Avant l'exécution d'une tâche, les processus vont déterminer si cette tâche doit être déléguée ou non. Si c'est le cas, alors un processus sera sélectionné pour l'exécuter, et la tâche, ainsi que ses arguments vont lui être envoyés. Le processus local va ensuite attendre de recevoir le résultat de la tâche, avant de libérer ses dépendances pour permettre d'exécuter les tâches suivantes. Nous pouvons noter que ces transferts de données peuvent nécessiter d'être implémentés par l'utilisateur si leur complexité ne permet pas d'être générés automatiquement. Enfin, si la tâche n'est pas sélectionnée pour être déléguée, alors le processus local l'exécute normalement.

L'algorithme 12 illustre quant à lui la gestion de ces tâches par travailleur. Après avoir reçu la tâche, et ses arguments, il va l'exécuter avant de renvoyer le résultat au processus d'origine. Nous pouvons enfin noter que cette suite d'instruction doit être ensuite réexécutée depuis le début, pour traiter de nouvelles tâches, jusqu'à ce que tous les processus aient terminé l'exécution de leurs calculs.

Nous pouvons enfin noter que les communications décrites dans les Algorithmes 11 et 12 peuvent être asynchrones, ce qui permet aux différents processus de ne pas être mis en attente lors des échanges.

Cependant, s'il est aisé de concevoir le mécanisme pour déléguer une tâche, il l'est beaucoup moins de déterminer quelle tâche sera une bonne candidate pour une exécution distante, et quel processus est à même d'être suffisamment disponible pour exécuter cette tâche sans devenir lui-même déséquilibré.

La délégation de tâche est à rapprocher du mécanisme dit de RPC(acronyme de *Remote Procedure Call*) [19], permettant d'exécuter des fonctions sur des processus distants. Dans ce cas, le travailleur va attendre qu'un autre processus lui demande d'exécuter une de ces fonctions, cette attente pouvant être recouverte par l'exécution de tâches locales.

La mise en œuvre des RPC peut se faire de diverses manières : par exemple par l'intermédiaire d'un mécanisme de type *Active Message*, par l'intermédiaire du système de communication (si l'on utilise Gasnet par exemple), ou encore de manière plus rudimentaire au dessus de MPI avec un système d'attente active périodique (par exemple avec `MPI_iprobe`) sur un communicateur ou un tag dédié. Une fois un message de contrôle échangé entre le processus qui souhaite déléguer du travail et le processus qui peut éventuellement l'assister, il suffit alors de reprendre le protocole décrit précédemment, où le processus originel se contente d'échanger des données et où le processus cible reçoit les données, effectue le calcul, et retourne les résultats.

Cette première méthode d'équilibrage dynamique pose également le problème de la détection de la terminaison de l'algorithme. En effet s'il n'est pas souhaitable qu'un processus se termine dès la fin de l'exécution de ses tâches, ce qui empêcherait la délégation de tâche vers lui, il est important qu'il détecte le moment où plus aucune tâches ne lui sera envoyé pour pouvoir passer à l'étape suivante ou pour terminer son exécution et libérer les ressources de calcul. Nous présenterons plus en détail ce problème dans la Partie 5.2.4.

Enfin, nous pouvons noter qu'une mise en œuvre statique peut également être possible : la décision de déléguer ou non serait alors prise avant la soumission des tâches, ainsi que le choix du travailleur. Les communications seraient alors implicitement mises en place par le runtime, et la détection de la terminaison pourrait se faire de la même manière que pour les algorithmes classiques, l'ensemble des communications étant connues dès l'étape de soumission.

Quelle que soit la méthode sélectionnée, il est nécessaire de déterminer si une tâche locale doit être déportée, et de choisir quel processus peut s'en occuper. Les modèles de performance que nous avons présentés dans la Partie 5.1.6 peuvent offrir un moyen efficace de prendre ce type de décision.

5.2.2 Tâches distribuées

Une deuxième approche possible pour rééquilibrer la charge de calcul en utilisant des méthodes dynamiques consiste à distribuer les tâches les plus coûteuses sur plusieurs nœuds de calculs.

En effet, certaines tâches nécessitent significativement plus de calculs que d'autres, ce qui limite le passage à l'échelle si le processus qui en est responsable est déjà fortement chargé ou si ces tâches se situent sur le chemin critique.

En supposant que ces tâches puissent être identifiées, et qu'elles permettent d'exprimer suffisamment de parallélisme, il est en fait possible de scinder ces

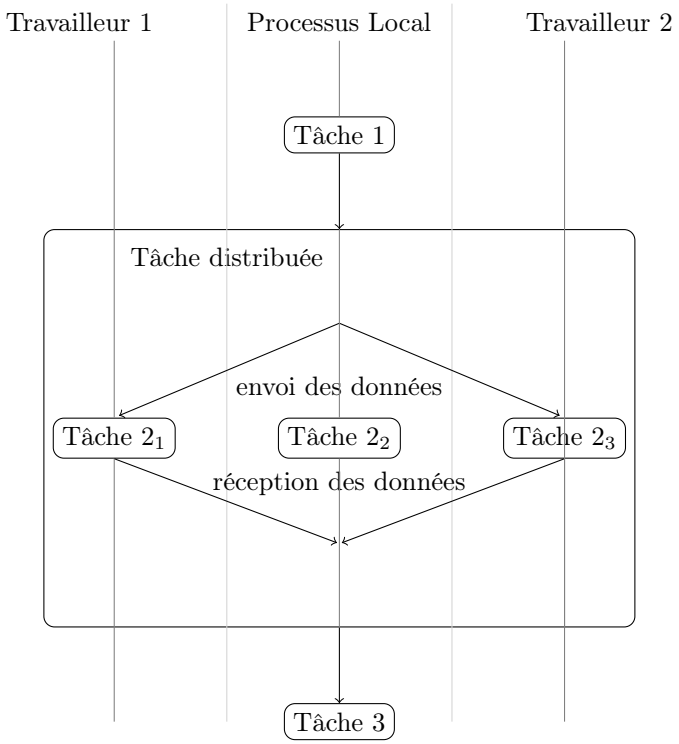


FIGURE 5.25 – Exemple d'implémentation des tâches distribuées

calculs entre plusieurs processus afin de les exécuter en parallèle.

D'une certaine façon, cela permet d'adapter la granularité de tâches trop coûteuses en les redistribuant entre plusieurs processus à l'aide d'une parallélisation à un grain plus fin. Dans le cas d'un solveur direct, on peut par exemple imaginer que les blocs les plus coûteux à mettre à jour seraient subdivisés en plusieurs sous-blocs moins coûteux, et de redistribuer ces sous-blocs pour répartir les tâches de calcul coûteuses.

Dans le cadre de notre solveur direct, une approche possible consisterait à adapter l'application elle-même pour gérer des blocs de tailles variables afin d'éviter de traiter des blocs particulièrement couteux. Cette approche se révèle cependant difficile à mettre en oeuvre en pratique, car au delà de la connaissance nécessaire *a priori* pour choisir une distribution des données appropriée, la parallélisation de notre solveur deviendrait un problème bien plus complexe puisqu'il faudrait gérer explicitement des dépendances significativement plus complexes. Par ailleurs, si le choix d'une granularité de calcul variable selon les blocs hiérarchiques peut s'avérer pertinente à un moment, il n'est pas certain que ce type de choix reste optimal pendant la totalité de l'exécution du code. Une solution dynamique est donc probablement plus simple à mettre en oeuvre, et plus pertinente pour s'adapter au comportement de l'algorithme pendant la totalité de l'exécution.

Plusieurs implémentations sont possibles pour mettre en oeuvre un mécanisme qui permet de distribuer des tâches entre plusieurs processus. La Figure 5.25 illustre l'une d'entre elles sur trois processus. On remarque en parti-

culier que l'on suppose ici que les processus se sont préalablement entendus pour traiter cette tache distribuée en parallèle, et que la tâche 2 est divisible en 3 parties de complexité équivalente. Dans cet exemple, trois processus exécutent une liste de tâches, dont la tâche 2 qui sera exécutée par les trois processus à l'aide d'une tâche distribuée. Cette dernière est composée de trois étapes principales :

1. Le processus local envoie les données d'entrée aux travailleurs.
2. Les différents processus exécutent leur partie de la tâche 2.
3. Les travailleurs renvoient les données calculées au processus local.

Dans cette proposition, les processus participant au calcul doivent insérer la tâche distribuée dans leur liste de tâches lors de l'étape de soumission, ce qui permet l'implémentation de cette méthode dans un modèle STF sans devoir changer grandement ce dernier.

Nous pouvons enfin noter que les étapes 1 et 3 sont des étapes de communication et peuvent donc être recouvertes par le calcul d'autres tâches pour augmenter l'efficacité de l'algorithme. Ce recouvrement n'a pas été représenté sur la Figure 5.25 par souci de clarté, mais peut être implémenté de manière simple à l'aide d'un modèle de tâche gérant les communications asynchrones.

Une seconde possibilité d'implémentation pourrait être de se reposer sur un mécanisme plus dynamique, tel que les RPC. Dans ce cas, le processus local sélectionnerait les différents travailleurs prenant part au calcul, et lancerait l'exécution des calculs via des appels de fonctions distantes. Ce mécanisme permettrait de déterminer au plus tard quels processus participent au calcul ou non. Cependant ce mécanisme est plus complexe et ne permet pas à un processus de déterminer simplement si d'autres processus sont plus ou moins chargés que lui, rendant toujours délicate la prise de décision.

Enfin, dans le cas d'un solveur direct de type H-matrices, une méthode statique pour détecter les calculs concernés peut être de distribuer tous les calculs traitant des données situées sur la diagonale de la matrice, ceux-ci étant généralement les plus coûteux.

D'une manière générale, il faut toutefois veiller à ce que les latences induites par la parallélisation de ces calculs ne nuisent pas au chemin critique. On veillera donc par exemple à redistribuer des tâches relativement éloignées du chemin critique, mais dont on sait qu'elles sont coûteuses.

Il est cependant plus délicat de sélectionner les processus ayant le moins de calcul à exécuter. De plus, les processus prenant part au calcul devant tous être désignés au moment de la soumission de la tâche, cette solution n'est que peu dynamique et ne permet pas forcément de prendre en compte quels sont les processus les plus à même de participer au calcul.

En revanche, il est possible de considérer statiquement qu'un ensemble de processus va participer à une tâche distribuée, mais d'introduire un préambule à la tâche distribuée afin de déterminer de manière collective quels processus sont effectivement impliqués. Si les processus auxiliaires considèrent qu'ils sont déjà trop chargés, ils peuvent annuler l'exécution de la tâche distribuée, et laisser un sous-ensemble des processus identifiés statiquement en charge de l'exécution.

Nous pouvons cependant noter qu'avec cette méthode une augmentation du volume de communication est inévitable. Les données devant être transférées entre les processus concernés avant et après le calcul, il est alors nécessaire de recouvrir ces communications par d'autres tâches pour permettre une bonne

efficacité de cette méthode de rééquilibrage. Il est donc important que d'autres tâches soient disponibles pendant l'exécution d'une tâche distribuée.

Algorithme 13 : Exemple d'une tâche distribuée permettant de calculer un GEMM

```
1  liste_proc ← participants(t)
2  si is_proc_orig(t) alors
3  |  local_part_A ← gemm_scatter(A, liste_proc)
4  |  local_part_B ← gemm_scatter(B, liste_proc)
5  |  local_part_C ← gemm_scatter(C, liste_proc)
6  sinon
7  |  local_part_A ← gemm_scatter(NULL, liste_proc)
8  |  local_part_B ← gemm_scatter(NULL, liste_proc)
9  |  local_part_C ← gemm_scatter(NULL, liste_proc)
10 fin
11 C_part ← C_part + A_part * B_part
12 si is_proc_orig(t) alors
13 |  C ← gemm_gather(local_part_C, liste_proc)
14 sinon
15 |  gemm_gather(local_part_C, liste_proc)
16 fin
```

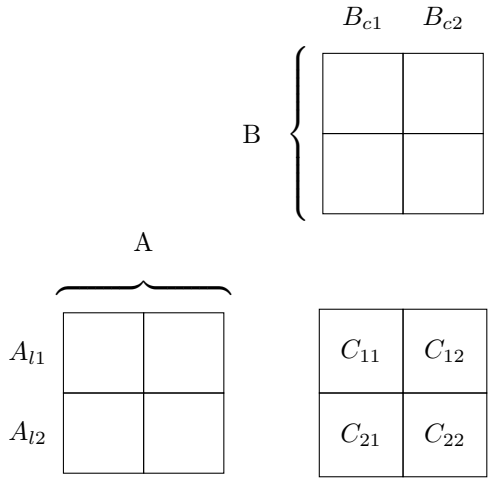


FIGURE 5.26 – Matrices A, B, et C, utilisées pour calculer C=C+AB

L'Algorithme 13 montre une implémentation possible d'une multiplication de matrices, à l'aide d'une tâche distribuée. Ce calcul utilise trois matrices, nommées A, B et C comme l'illustre la Figure 5.26. À la suite de son exécution, la matrice C aura pris la valeur de $C+A*B$, tandis que les deux autres n'auront pas changé de valeur. Au début de l'exécution ces trois matrices sont accessibles uniquement par un seul processus, celui qui aurait dû effectuer le calcul en temps normal.

Après avoir déterminé les processus prenant part à la tâche distribuée, les

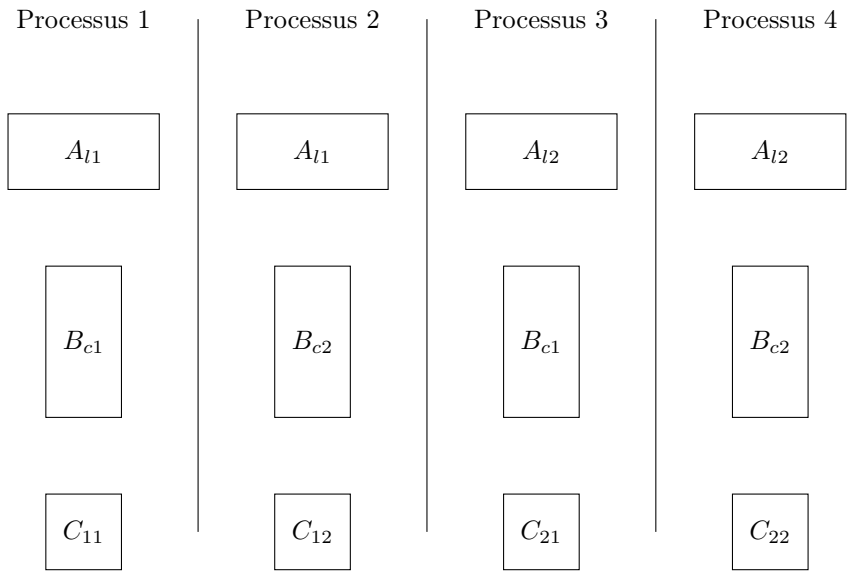


FIGURE 5.27 – Distribution des matrices A, B, et C sur 4 processus

trois matrices sont distribuées parmi ceux-ci. La Figure 5.27 montre cette distribution sur 4 processus. On peut noter que chaque bloc de la matrice C n'a été envoyé qu'à un unique processus, tandis que les blocs des matrices A et B ont été dupliqués sur plusieurs participants. Les envois de ces blocs peuvent donc être optimisés en utilisant des communications collectives plutôt que des communications point à point.

Enfin les processus effectuent leur partie du calcul, avant de renvoyer le résultat au processus de départ.

Ainsi l'implémentation du produit matriciel reproduit exactement ce que l'on aurait fait pour concevoir cet algorithme à l'aide de MPI, sans tenir compte du fait que l'on est dans une tâche distribuée. Comme c'est le cas avec le reste des échanges MPI, on utilise cependant des routines utilitaires afin d'effectuer les échanges MPI de manière asynchrone au sein de tâches qui sont effectuées par le thread principal afin de contourner les limitations de la couche MPI vis à vis du support des threads.

Une fois les calculs terminés, le résultat est reconstruit sur le processus principal, et la terminaison de cette tâche distribuée débloque les dépendances induites par la tâche distribuée. La tâche ainsi distribuée s'insère donc naturellement dans le graphe de tâches du processus principal, et s'entrelace avec les calculs des autres processus qui participent à la tache distribuée.

En reprenant les exemples donnés précédemment dans la Figure 5.14, nous pouvons par exemple noter qu'une distribution des tâches présentées permettrait d'augmenter à nouveau le parallélisme, et donc de réduire les phases où les processus sont bloqués par les dépendances. Nous pouvons donc voir que cette méthode peut être utile, même quand le calcul est déjà globalement équilibré.

5.2.3 Principe du vol de travail

Enfin, une troisième méthode de rééquilibrage dynamique est le vol de travail. Elle permet d'inverser la prise de décision : ce sont les processus les moins chargés en calcul qui vont prendre la décision de déclencher le transfert des tâches. De cette façon, il est facile de garantir que ce transfert ne risque pas de rajouter du temps de calcul au processus.

Il existe différentes façons de concevoir un système basé sur du vol de travail que nous allons présenter dans cette Section. En particulier, on distingue les approches qui nécessitent une collaboration explicite entre les processus victimes et les processus voleurs. Des considérations techniques ont également une influence certaine sur la conception d'un protocole de rééquilibrage basé sur le vol de tâche. Par exemple, le type de mémoire accessible par les mécanismes mis en oeuvre lors du vol de tâches (e.g. RDMA) aura un impact sur la façon de déplacer les données nécessaires à la réalisation de tâches volées à un autre processus.

5.2.3.a Vol coopératif ou non coopératif

Tout d'abord nous pouvons lister deux gestions différentes du vol de travail : le vol non coopératif et le vol coopératif. Lors d'un vol coopératif, la victime et le voleur vont déterminer conjointement si un vol est possible, par l'intermédiaire du système de tâches. Les tâches pouvant être volées ayant été annotées, celui-ci peut déterminer automatiquement si des tâches peuvent êtres transférées ou non.

Le cas échéant, les processus vont établir une communication point à point pour transférer les calculs. Ce transfert pouvant être complexe, il est parfois nécessaire que les méthodes de communication soient implémentées par l'utilisateur, pour assister le système de tâches.

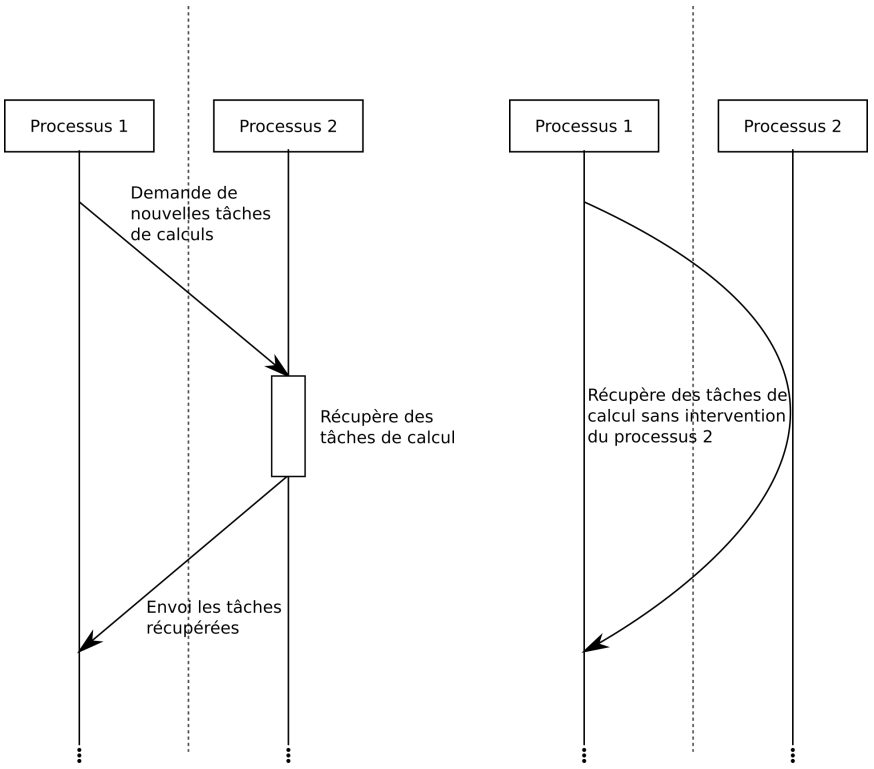
À l'inverse, lors d'un vol non coopératif, le voleur va déterminer seul si un vol est possible et transférera sans assistance le calcul depuis la victime.

Comme le montre la Figure 5.28a, lors d'un vol coopératif, le voleur va envoyer un message à la victime pour indiquer son intention de transférer des calculs depuis cette dernière. Puis, celle-ci va consulter ses listes de tâches internes avant d'envoyer des tâches de calculs si elle en possède en nombre suffisant, ou indiquer que le transfert n'est pas possible.

Cette méthode peut être rapprochée de celle des tâches déléguées, où la décision de déléguer ou non une tâche serait décidée non pas au moment de leur soumission mais quand un processus ne posséderait plus aucune tâche. De cette façon la problématique de trouver des processus n'ayant pas ou peu de travail en cours est résolue, ces derniers se signalant d'eux-mêmes, et est remplacée par la problématique de trouver des processus ayant suffisamment de travail pour supporter un vol.

Le vol coopératif permet également une grande liberté sur les structures de données utilisées pour lister les différentes tâches en attente d'exécution. En effet, l'accès étant effectué par un thread local, les contraintes sont les mêmes que pour les accès en mémoire partagée, et les algorithmes peuvent être implémentés à l'aide de solutions telles qu'*OpenMP*.

Cependant, cette méthode demande une coopération active de la victime, ce qui peut occasionner des coûts supplémentaires lors des tentatives de vols.



(a) Schéma d'un vol coopératif (b) Schéma d'un vol non coopératif

FIGURE 5.28 – Différences entre un vol coopératif et non coopératif

À l'inverse, lors d'un vol non coopératif, seul le voleur va manipuler les structures de données des victimes comme le montre la Figure 5.28b.

De cette façon, seul le voleur subit un surcoût lors d'une tentative de vol, la victime n'ayant pas besoin d'effectuer de calculs.

Cependant, cette méthode est plus complexe à mettre en œuvre que pour les vols coopératifs : il est nécessaire que le processus voleur puisse accéder directement à la mémoire de la victime en lecture et écriture pour manipuler lui-même les structures ordonnant les calculs. Ce type d'accès n'est pas toujours possible quand les deux processus se trouvent sur deux nœuds différents. En effet, suivant le type de réseau mis en place sur le calculateur, ces accès directs peuvent être rendus possibles ou non. Par exemple, les réseaux ethernet classiques ne prévoient pas ce type d'accès tandis qu'ils sont rendus possibles par des réseaux InfiniBand [72].

Enfin, les systèmes de synchronisation en mémoire distante ne sont généralement pas compatibles avec ceux en mémoire distribuée. Par exemple les standard OpenMP et MPI, qui sont respectivement les standards couramment utilisés en mémoire partagée et distribuée, ne prévoient pas d'interfaçage l'une avec l'autre. Il est alors nécessaire d'adapter les structures de données utilisées pour accéder aux listes de tâches pour qu'elles puissent être utilisées aussi bien via les accès mémoire distants qu'avec les accès mémoire locaux.

5.2.3.b Structure de données de la liste de tâches

Il existe un grand nombre de structures de données pour stocker les différentes tâches en attente d'exécution. Cependant, toutes ne sont pas également pertinentes dans le cadre de vol de tâches. Par exemple, si une liste simplement chaînée peut être pertinente dans le cadre d'accès en mémoire partagée, son implémentation naïve requiert un grand nombre d'allocation mémoire. Ces allocations étant complexes à réaliser dans un environnement à mémoire distribuée utilisant des accès RDMA, cette méthode serait moins pertinente dans un tel contexte.

Pour permettre une utilisation efficace de ces structures, il est nécessaire de limiter l'utilisation des sections critiques lors de leurs accès. De même, à cause de la grande latence des accès à la mémoire des processus distants il est important de limiter le nombre d'accès nécessaires à leur manipulation. Enfin, il est nécessaire d'allouer l'ensemble de la mémoire nécessaire aux structures utilisées dès leur initialisation, les opérations de réallocations mémoires pouvant être complexes dans le cadre d'un système se basant à mémoire distribuée.

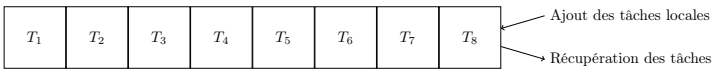


FIGURE 5.29 – Schéma d'un vol de tâche utilisant une pile

Une première méthode envisageable est d'utiliser une pile de tâche comme structure de données, comme le montre la Figure 5.29. Avec celle-ci, les processus locaux empilent et dépilent les tâches au sommet de la pile. Les processus distants dépilent également les tâches situées en haut de la pile quand un vol est nécessaire. Avec cette structure de données nous pouvons observer que les vols s'effectuent au même endroit que les accès locaux, ce qui peut donc ralentir les victimes potentielles.

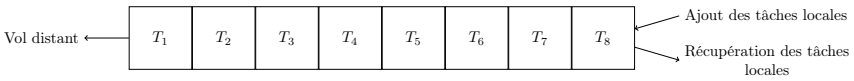


FIGURE 5.30 – Schéma d'un vol de tâche utilisant une liste doublement chaînée

La Figure 5.30 présente une seconde structure possible, qui utilise une liste doublement chaînée. Avec ce modèle, les processus locaux enfilent et défilent les tâches situées à la tête de la file. De leur côté, les processus distants défilent les tâches situées à la queue de la file. Les voleurs et les victimes ne manipulant pas les mêmes données, il est alors possible d'implémenter des algorithmes permettant aux victimes de ne pas attendre de verrous, comme le propose le langage CILK [49]. Un verrou est toujours nécessaire aux voleurs pour s'assurer qu'un seul d'entre eux puisse manipuler la queue de la file.

Pour permettre de limiter la manipulation de la mémoire, il est possible d'utiliser un tableau comme conteneur de la liste de tâche. Dans ce cas, il n'est plus nécessaire d'allouer ou de libérer de la mémoire lors des différents accès à la structure, ces actions étant complexes à exécuter avec des accès distants.

Cependant, dans ce cas nous pouvons observer que la mémoire utilisée par les tâches volées ne peut pas être réutilisée par de nouvelles tâches, conduisant à un

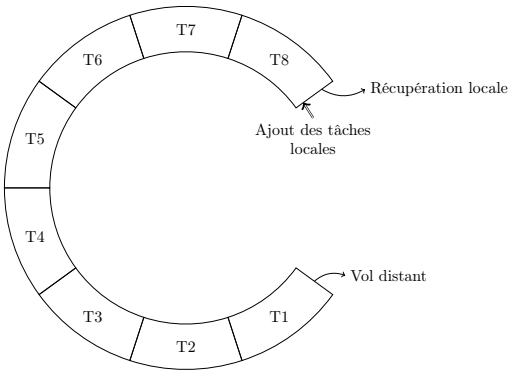


FIGURE 5.31 – Schéma d'un vol de tâche utilisant un tableau circulaire

gaspillage de ressources. Pour résoudre ce problème, il est possible de s'appuyer sur un tableau circulaire, comme le montre la Figure 5.31.

Une liste implémentée avec un tableau circulaire permet de réutiliser les premières cases de la liste pour enfiler de nouveaux éléments, si ces premières cases ne sont plus utilisées. De cette façon, les ressources ne sont plus gaspillées après un vol.

Quelle que soit la solution retenue, il est important que ces structures puissent contenir un nombre de tâche suffisant pour ne pas limiter le vol de tâche par manque de tâche volable. Nous avons noté qu'au dela de quelques centaines de tâches, le vol pouvait se dérouler jusqu'à la fin de l'exécution sans être confronté à des situations de famines.

5.2.3.c Intégration dans un modèle STF

Un autre sujet sur lequel des choix doivent être effectués est l'intégration du vol de tâche avec un modèle de tâche utilisant un système de dépendances. Dans un tel modèle, les données d'entrée des tâches de calcul ne sont pas nécessairement prêtes dès leur soumission, celles-ci pouvant être calculées par une ou plusieurs autres tâches. Dans le modèle STF, cette contrainte sur l'ordre d'exécution est représentée par des dépendances entre les tâches, permettant de garantir que l'une d'entre elles ne peut être exécutée avant que ces dépendances soient calculées.

En s'appuyant sur ces dépendances, il est possible de s'assurer que les tâches ayant une ou plusieurs dépendances en entrée ne puissent pas être transférées à un autre processus avant la fin de l'exécution de ces dépendances. La résolution des dépendances ayant lieu lorsque les données deviennent à nouveau disponibles, à la fin de l'exécution d'une tâche, les tâches dépendant de tâches volées ne deviennent prêtes et ne sont soumises dans les files d'exécution qu'après l'exécution de la tâche initiale. Le mécanisme de vol préserve donc bien un ordre d'exécution cohérent avec les dépendances originales puisque le vol n'intervient que sur des tâches dont les dépendances sont vérifiées, et puisque que la résolution des dépendances intervient avant de rendre les tâches exécutables.

La prise en compte des dépendances en entrée ne pose pas nécessairement de problème, même si une exécution efficace nécessite de copier les données d'entrée suffisamment à l'avance pour masquer la latence introduite par le vol.

En revanche, la situation n'est pas aussi simple pour les dépendances en sortie des tâches. En effet, il peut exister d'autres tâches chez la victime dépendant des tâches volées, et il faut déterminer si les données modifiées et les tâches qui en dépendent doivent être rapatriées immédiatement sur le processus original. Dans ce cas, il existe principalement deux possibilités : soit voler également l'ensemble des tâches filles, soit renvoyer les données produites par la tâche volée dès sa terminaison.

La première solution permet de voler du travail à une granularité plus grossière, augmentant la quantité de travail volé, et donc l'efficacité du vol de travail. Cette forme peut être assez naturelle dans le cadre où le graphe de dépendances forme un arbre, le transfert de données n'étant nécessaire que pour la première tâche. Cependant, dans le cas où une tâche dépend d'au moins deux autres tâches, l'une de ces dépendances peut provenir d'une tâche volée, et l'autre d'une tâche non volée, ce qui rend la gestion des données plus complexe.

La seconde solution, consistant à renvoyer les données produites par les tâches volées aux victimes dès leur terminaison impose de voler les tâches à une granularité plus fine, ce qui peut conduire à une diminution de l'efficacité du vol de tâche, dans le cas ou plusieurs auraient pu être volées.

Cependant, la gestion des dépendances est grandement simplifiée, dans le cas de dépendances multiples, celles-ci étant à la charge du processus d'origine. De cette façon, les algorithmes utilisés pour ce problème restent les mêmes que sans vol de travail.

C'est cette dernière solution que nous avons privilégiée dans le cadre de cette thèse, la plupart des tâches de nos applications dépendant de plusieurs autres tâches.

D'autres approches pourraient supposer que l'on ne migre pas seulement la tâche mais également la responsabilité des données que l'on modifie sur un processus distant, mais cela nécessite une vision totalement distribuée du graphe de tâche que nous supposons seulement connu localement, processus par processus. Bien que cela ne nécessite pas de rapatrier les données modifiées (ou d'effacer celles lues), cette approche de vol particulièrement agressive serait difficile à concevoir sans augmenter considérablement les mouvements de données entre les processus si l'on vole des tâches sans tenir compte de la localité des données.

Enfin, nous pouvons noter que certaines tâches vont soumettre d'autres tâches à l'intérieur de leur calcul, en ne dépendant que de leurs propres entrées. Dans ce cas, le vol d'une unique tâche peut conduire à la création de nouvelles tâches. Si ces nouvelles tâches ne peuvent pas se terminer avant leur tâche parente, alors la gestion des dépendances n'est pas affectée et plus de travail peut être transféré en un vol.

À l'inverse, si ces nouvelles tâches peuvent se terminer après leur tâche parente, ce qui correspond à un modèle asynchrone, alors nous nous retrouvons dans le cas où la gestion des dépendances peut être complexifiée. Nous avons donc choisi que dans ce cas-ci, les tâches parentes ne soient pas volables.

5.2.3.d Choix d'une victime

Enfin, un troisième problème commun à la majorité des systèmes de vol de travail consiste dans le choix des victimes. Celui-ci est important, car pour que le vol équilibre au mieux la charge de calcul, il est nécessaire que les victimes soient sélectionnées parmi les processus possédant le plus de calculs à exécuter.

En revanche, un nombre trop grand de processus tentant d'effectuer un vol auprès d'une même victime peut conduire à de la contention, et donc à réduire l'efficacité du rééquilibrage. Il est donc nécessaire que les différents voleurs ne choisissent pas tous la même victime.

Il est alors naturel de chercher des heuristiques pour sélectionner les processus permettant la meilleure efficacité de vol.

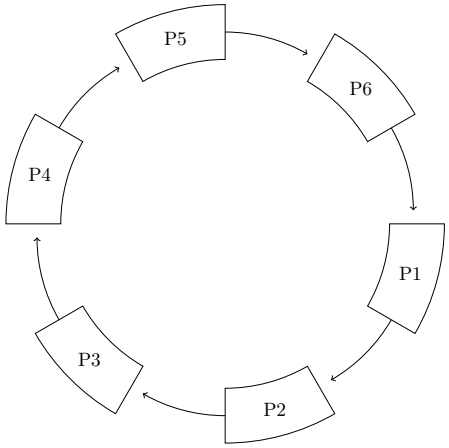


FIGURE 5.32 – Processus disposés en anneau pour le vol de travail

Une première solution simple consiste à désigner pour chaque processus un successeur, comme le propose la Figure 5.32. Le successeur d'un processus n est le processus $n + 1$, sauf pour le dernier processus, qui a pour successeur le processus 0.

Pour sélectionner une victime, les voleurs vont commencer par choisir leur successeur. Si celui-ci ne peut pas être volé, par exemple parce qu'il ne possède pas assez de tâches, le voleur passe au processus suivant. Si le vol se termine sur un succès la victime est mémorisée par le voleur et sera le premier processus consulté.

De cette façon, les voleurs vont consulter tous les autres processus, ce qui permet de garantir qu'un processus possédant du travail ne sera pas ignoré. En revanche, si plusieurs voleurs tentent de voler au même moment un même processus, il est probable qu'ils réussissent tous le vol, ou qu'ils échouent tous. Lors du vol suivant, ils vont alors à nouveau voler la même victime, ce qui peut conduire à de la contention.

Une autre heuristique aurait pu être de favoriser les processus étant les plus proches du voleur pour limiter la latence des communications et donc augmenter la vitesse des vols quand ceux-ci sont possibles. Cependant, l'implémentation de cette heuristique peut s'avérer complexe à mettre en œuvre.

Une solution fréquemment utilisée est de sélectionner un processus au hasard [21], ce qui permet de réduire la probabilité qu'un même processus soit sélectionné par plusieurs processus, et donc de réduire la contention.

5.2.4 Détection de terminaison

Dans cette partie nous allons discuter du problème de terminaison. Nous commencerons par le définir et expliquer l'importance de ce problème, avant de présenter deux algorithmes permettant de le traiter. Enfin nous présenterons les problématiques liées à l'utilisation de ces algorithmes dans un système de tâche existant.

5.2.4.a Présentation du problème de terminaison

La terminaison d'un processus correspond au moment où l'ensemble des calculs localement soumis ont été exécutés, et où le processus ne possède plus de calcul à exécuter.

Déterminer ce moment avec précision est important : un processus qui met trop de temps à déterminer qu'il n'a plus de travail à faire va ralentir l'exécution globale du calcul, tandis qu'un processus terminant son exécution trop tôt va dans le meilleur des cas terminer son exécution alors qu'il aurait pu décharger un autre processus. Dans le pire des cas, si un processus termine son exécution avant que les calculs locaux aient été terminés, le résultat sera manquant ou erroné.

Dans un contexte statique, où les calculs sont exécutés sur le même processus que celui sur lequel ils ont été soumis, la terminaison d'un processus est triviale, et correspond au moment où tous les calculs soumis par le processus ont été exécutés.

En revanche, dans un contexte avec du rééquilibrage de charge dynamique, où des calculs peuvent être transférés d'un processus à un autre, la détection de la terminaison est plus compliquée. En effet, la terminaison devient effective seulement lorsqu'il n'y a plus de tâche à effectuer, et qu'il ne reste plus de tâche en cours d'exécution, car il est possible que de nouvelles tâches soient soumises depuis une tâche. Il s'agit donc d'un problème d'algorithmique distribuée qu'il faut traiter avec soin pour assurer l'efficacité de nos techniques de rééquilibrage de charge.

5.2.4.b Algorithme de consensus

Pour résoudre cette problématique, une première solution a été de propager un *état* lors du vol. Cet état peut prendre différentes valeurs : *Actif*, *En cours d'arrêt*, ou *Terminé*. Au cours de l'exécution, chaque processus indique son état actuel lors de ses tentatives de vol, et maintient la liste du dernier état *connu localement* de l'ensemble des autres processus.

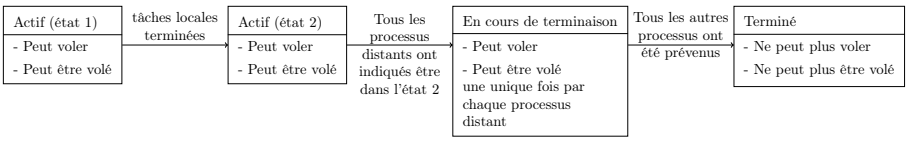


FIGURE 5.33 – Algorithme naïf de terminaison

Comme le montre la Figure 5.33, chaque processus commence son exécution dans le premier état, qui est l'état *Actif*. Dans cet état, les processus vont exécuter les tâches localement disponibles, et essayer de voler du travail aux

autres processus quand il n'y aura pas assez de tâches pour alimenter les fils d'exécution des threads.

Lorsque l'ensemble des tâches localement soumises ont été exécutées, potentiellement avec la coopération d'autres processus, les processus rentrent progressivement dans l'état numéro 2. Cet état n'est utilisé que par l'algorithme de terminaison, les processus vont continuer à exécuter des tâches comme dans l'état numéro 1, et à voler des tâches lorsque nécessaire. De plus, les processus dans cet état peuvent également subir des vols de tâches, sur des tâches volées à un autre processus ou sur des tâches dont les dépendances ont été libérées après que le processus soit rentré dans l'état numéro 2.

Quand l'ensemble des processus auront indiqué être dans ce deuxième état au processus courant, celui-ci basculera dans le troisième état, qui correspond au début de la terminaison. Dans cet état, les processus vont indiquer à chaque autre processus être en train de se terminer. Cela leur permet de ne pas tenter de voler de tâche sur un processus qui a achevé sa terminaison, et est donc dans l'incapacité de répondre aux éventuelles tentatives de vols.

Quand chaque processus a été prévenu, alors les processus passent dans le dernier état. Le processus peut alors se couper, étant assuré que plus aucun processus ne possède du travail, ni ne va lui en demander.

De cette façon, nous pouvons séparer les différentes phases du vol de tâche, ce qui permet de spécifier les différents comportements. Au début de l'algorithme, chaque processus est actif et peut répondre à du vol de travail. Il peut également, lors des situations de famines locales, voler du travail à un autre processus pour contribuer au calcul. Quand toutes les tâches locales ont été exécutées, il est toujours souhaitable de récupérer du travail depuis les processus ayant un surplus de charge de calcul.

Cette méthode est robuste et simple à implémenter, mais peut considérer les processus comme inactifs de manière prématurée. En effet, lorsque tous les processus sont rentrés dans l'état numéro 2, par exemple à la suite de famines temporaires, les processus basculerons dans l'état 3, et ne pourront plus subir de vols. Cependant, si de nouvelles tâches sont générées, alors ces vols auraient pu être intéressants et contribuer au rééquilibrage. Enfin, ce protocole impose que chaque processus vole à chaque autre processus pour faire progresser l'algorithme, et connaître l'état de l'ensemble du système distribué. En pratique, le choix aléatoire des victimes permet de garantir cette propriété avec une grande probabilité, les vols étant assez fréquents pour que chaque processus soit régulièrement tiré au sort.

Pour résoudre ces problèmes, nous avons alors implémenté un algorithme proposé par Nissim Francez et Michael Rodeh [47]. Celui-ci dispose les processus sur les nœuds d'un arbre binaire couvrant la totalité des processus (ou *spanning tree* en anglais). Lorsqu'un processus situé sur une des feuilles ne possède plus de travail local, il prévient son père. Quand un processus situé sur un des nœuds ne possède plus de travail non plus, il va attendre que ses fils lui aient indiqué ne plus avoir de travail avant de prévenir le sien.

Quand le processus situé à la racine de l'arbre ne possède plus de travail, et que ses deux fils lui ont indiqué ne plus en avoir non plus, nous pouvons donc garantir que chaque processus, à un moment donné, ne possédait plus de travail à exécuter. La racine va alors envoyer un message pour demander une confirmation.

À nouveau, ce message va partir des feuilles vers la racine, et chaque proces-

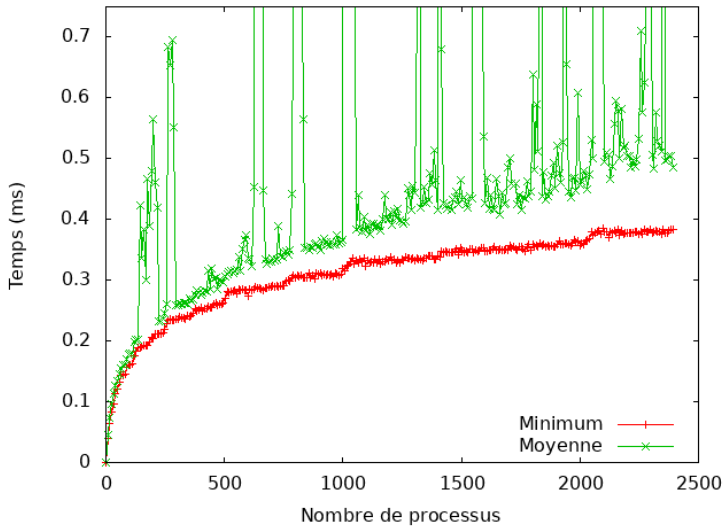


FIGURE 5.34 – Latence du protocole de terminaison par consensus

sus va indiquer si lui ainsi que ses deux sous-arbres sont restés inactifs depuis l'envoi du dernier message, ou s'ils ont reçu du travail depuis. Si la racine reçoit l'information selon laquelle au moins un processus a reçu du travail, alors nous ne pouvons rien en déduire et l'algorithme reprend depuis la première étape. Dans le cas contraire, alors nous pouvons en déduire, qu'aucun processus ne possède de travail à exécuter, et que le calcul peut donc être interrompu.

En supposant que tous les processus soient déjà dans un état inactif, il suffit donc de parcourir deux fois l'arbre couvrant pour garantir la terminaison. Le temps nécessaire pour détecter la terminaison depuis n'importe quel processus est donc de l'ordre de $2log_2(n)L$ pour n processus, si l'on suppose que L correspond à la latence pour traiter les messages. Il s'agit donc d'une approche qui permet un bon passage à l'échelle. La Figure 5.34 montre ainsi la latence de terminaison pour un nombre de processus compris entre 1 et 2400, sur la partition skylake de la machine INTI : malgré des variations notables liées au bruit système sur un code avec un processus par coeur, on constate bien une allure qui correspond à une complexité logarithmique. On constate également qu'il suffit de quelques centaines de micro-secondes pour effectuer cet algorithme sur plusieurs milliers de processus.

Cette nouvelle méthode est plus complexe à implémenter mais permet de garantir que la terminaison de l'algorithme ne sera pas détecté de manière prématurée. De plus, les communications décrites étant indépendantes de celles nécessaires pour le vol, le choix d'une victime est complètement libre, et la progression de l'algorithme de terminaison n'est pas ralentie par celle du vol de tâche.

Pour ces raisons, nous avons choisi d'utiliser ce dernier algorithme dans nos méthodes de vol de travail.

```
void init_termination(struct termination_ctx *ctx, MPI_Comm comm);
void finalize_termination(struct termination_ctx *ctx);

// il ne reste plus de travail local et on ne gère pas de tâche distante
void declare_stability(struct termination_ctx *ctx);

// on vient de voler du travail ou une tâche a été soumise
void declare_unstability(struct termination_ctx *ctx);

//renvoi 1 si l'algorithme est stable, 0 sinon.
int test_stability(struct termination_ctx *ctx);
```

FIGURE 5.35 – Interface de la bibliothèque de détection de terminaison

5.2.4.c Intégration dans un modèle de tâche

Le protocole que nous avons présenté dans la partie précédente peut être implémenté indépendamment du reste des calculs et des communications, ce qui permet une séparation de code efficace. En effet, seules trois fonctions sont nécessaires pour interagir avec ce protocole : une première pour indiquer que le processus local ne possède plus de travail, une deuxième pour indiquer qu'il a pu en récupérer et une dernière pour interroger l'état global. L'algorithme de terminaison que nous utilisons peut donc être implémenté sous la forme d'une bibliothèque facilement réutilisable, dont la Figure 5.35 montre l'interface.

Ce faible nombre de fonctions dans l'interface permet de simplifier l'intégration de cette méthode : la fonction indiquant que du travail a été récupéré s'appelle naturellement à la suite d'un vol réussi, tandis que la fonction testant l'état global sera utilisée comme condition d'arrêt du système de tâche.

On notera ici que si l'algorithme original ne considère qu'un ensemble de processus interagissant avec des messages, il nous faut traiter un parallélisme hybride où chaque processus peut traiter simultanément plusieurs tâches. Il n'est donc plus aussi aisé de déterminer si un processus est *actif* ou *inactif* puisqu'il est possible qu'un thread soit actif pendant que d'autres ne le sont pas au sein du même processus.

La fonction servant à indiquer l'inactivité du processus est donc plus complexe à intégrer à un système de tâche. Le passage d'un état à un autre peut arriver à différentes occasions, soit en cours d'exécution sur un des threads du processus, ou encore lors de la mise en attente sur des dépendances non résolues.

Pour tirer parti de l'implémentation de notre algorithme de terminaison sous la forme d'une bibliothèque qui met à disposition des routines pour déclarer de tels changements d'état indépendamment de l'application sous-jacente, nous nous sommes appuyés sur un mécanisme de callbacks (ou *hooks* en anglais) proposés par le système de tâches que nous utilisons. Ce mécanisme simple permet d'appeler des fonctions arbitraires lors de certains évènements, tels que la soumission d'une tâche, ou lorsqu'un coeur ne parvient pas à récupérer de travail. Quand un thread cherche à récupérer une tâche, après avoir cherché dans les listes locales, il va donc informer le système de terminaison du succès ou non de l'opération. Ainsi, nous pouvons connaître quels threads sont actifs, ou non. Quand tous les threads sont inactifs, alors nous pouvons en déduire qu'il n'existe plus de tâche exécutable sur le processus, et donc que l'ensemble

du processus est *inactif*. À l'inverse, dès qu'un vol est effectué avec succès, le processus rentre dans l'état *actif*, les threads pouvant à nouveau exécuter des tâches.

Cette approche permet donc de déterminer correctement s'il existe encore du travail dans un système distribué reposant sur un mécanisme de tâches interdépendantes. Par exemple, lorsque des tâches sont en attente d'un autre processus (et ne sont donc pas stockées dans les listes de tâches locales), il existe alors au moins une tâche en cours d'exécution, qui est elle-même soit active, soit en attente d'une dépendance.

Dans le premier cas, l'algorithme de terminaison détectera qu'il reste au moins un processus actif, puisque le thread qui exécute cette tâche est actif. Dans le second cas, cette nouvelle tâche dépendra forcément d'une autre tâche qui sera soit active, soit elle-même en attente d'une dépendance. Par un raisonnement récursif, il reste donc forcément au moins une tâche en cours d'exécution sur un des processus. Il n'est donc pas possible de détecter une terminaison de manière trop précoce même si toutes les tâches soumises localement sont bloquées sur des dépendances extérieures au processus.

Nous avons donc vu que cette méthode permet de détecter correctement la terminaison, tout en ne nécessitant qu'un nombre limité de modifications pour l'intégrer dans un système existant. La conception de cet algorithme sous la forme d'une bibliothèque offre par ailleurs la possibilité de réutiliser aisément cet algorithme complexe dans d'autres applications qui nécessiteraient de détecter la terminaison d'un calcul distribué de manière dynamique.

Cette simplicité a d'ores et déjà permis d'utiliser avec succès ce mécanisme pour détecter la terminaison dans un code existant, permettant d'utiliser un algorithme distribué dont le nombre de calcul est difficilement prévisible, et donc dont la terminaison n'était pas *a priori* aisément détectable.

5.2.5 Valorisation dans un code d'aérodynamique hypersonique

Bien que l'algorithme utilisé pour détecter la terminaison ne soit pas nouveau en lui-même, le fait de disposer d'une implémentation flexible avec une interface simple nous a permit de réutiliser ce développement dans un code d'aérodynamique hypersonique basé sur une méthode innovante appelée méthode des frontières immergées [26].

La Figure 5.36 montre un exemple de calcul réalisé à l'aide de ce code actuellement parallélisé à l'aide de MPI et OpenMP. Au delà des détails liés à cette application qui sortent du cadre de notre document, il est intéressant de noter que cette méthode peut tirer parti de notre algorithme de consensus distribué pour détecter la terminaison d'un calcul distribué.

En effet, le problème est réparti en plusieurs domaines qui sont mis à jour de manière distribuée sur les différents processus MPI. Un système de mailles fantômes (ou halo) permet d'échanger des données d'un processus à l'autre entre les différentes itérations. Si ce mécanisme classique est suffisant pour la plupart des calculs, le calcul d'un paramètre physique (hauteur de la couche limite) met en oeuvre un algorithme innovant, mais particulièrement difficile à écrire avec MPI.

Cet algorithme, illustré par la Figure 5.37, nécessite de parcourir les données selon un schéma de spirale dont la longueur peut être particulièrement

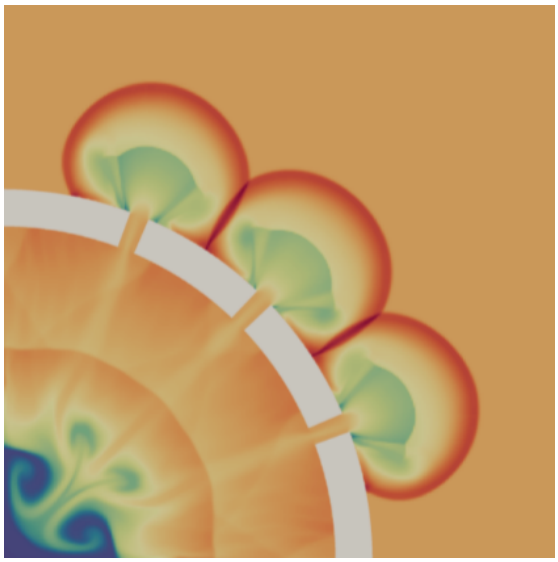


FIGURE 5.36 – Exemple de calcul 2D réalisé avec la méthode des frontières immergées [26]

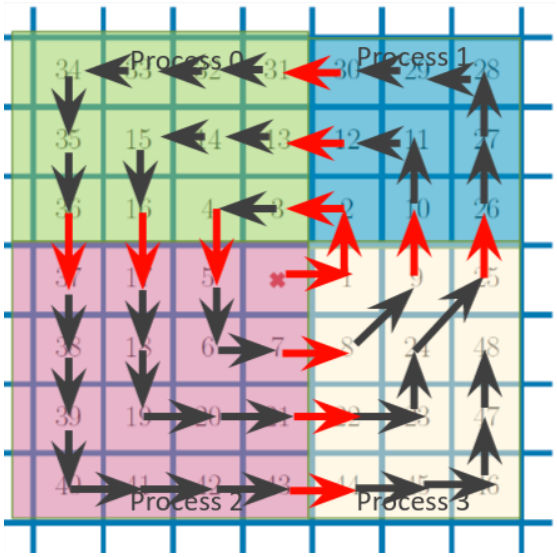


FIGURE 5.37 – Parcours en spirale nécessaire pour calculer la hauteur de la couche limite. La tâche effectue un parcours au sein de 4 domaines (processus) distincts, et nécessite une migration à chaque passage d'un domaine à l'autre. Pour terminer l'algorithme, il est nécessaire à chaque processus d'assurer qu'aucun autre processus ne va lui soumettre de tâche migrable.

importante, et qui peut traverser un nombre arbitraire de domaines. Il n'est pas concevable d'élargir suffisamment la taille des halos car ces parcours sont bien trop longs. L'approche que nous avons adoptée pour résoudre ce problème consiste à utiliser des tâches *migrables*, c'est à dire des tâches que l'on peut migrer explicitement sur un autre processus (à l'instar des tâches déléguées présentées dans la Section 5.2.1). Ainsi, lorsque la tâche qui effectue le parcours en spirale doit accèder à des données qui ne sont pas disponibles sur le processus local, la tâche associée à ce calcul est *migrée* sur le processus auquel appartient la donnée nécessaire. La mise en oeuvre de ces tâches migrables a d'ailleurs été directement inspirée par les mécanismes présentées dans la Section 5.2.1.

Ce mécanisme a ainsi permit de paralléliser de manière finalement très simple le calcul de la hauteur de la couche limite. En revanche, avant de terminer cette phase de calcul, il est désormais nécessaire de s'assurer que toutes les tâches locales sont effectuées, et surtout d'être certain qu'il n'y a plus de tâches distantes qui peuvent être attribuées à un processus qui aurait déjà fini ses propres tâches locales. Nous avons donc réutilisé *sans aucune modification* la bibliothèque développée dans le cadre de cette thèse pour mettre en oeuvre l'algorithme de consensus distribué.

Cela illustre donc non seulement l'intérêt du principe de délégation de tâches de la Section 5.2.1, mais surtout l'apport de l'implémentation efficace de l'algorithme de consensus distribué présenté dans la Section 5.2.4.c.

5.2.6 Vol de tâche coopératif

Dans cette partie nous allons présenter une première méthode dans laquelle la victime et le voleur coopèrent pour effectuer un vol de travail. Nous commencerons par présenter ce type de vol proprement dit, avant de discuter de plusieurs problèmes soulevés par cette méthode. Enfin nous évoquerons plusieurs limites de ce type de vol.

5.2.6.a Principe

Une première méthode pour implémenter du vol de travail est d'utiliser des communications point à point. Dans cette méthode, les victimes vont coopérer lors du vol de travail, que nous nommerons donc *vol coopératif*. Pour cela chaque processus se tient prêt à être la cible d'un vol, en vérifiant régulièrement si des messages indiquant l'intention d'un vol lui ont été envoyés.

Lors de la réception de tels messages, les processus vont indiquer s'il reste suffisamment de tâches dans les listes locales pour un vol, et les envoyer le cas échéant. Ils vont alors se préparer à recevoir un nouveau message, indiquant la complétion des tâches volées, et contenant les données de sortie de ces tâches.

Enfin, après avoir reçu ces résultats, la victime va débloquer les tâches en dépendant, et les exécuter soit localement, soit sur un processus distant en cas de nouveau vol.

Ce mécanisme possède l'avantage de pouvoir être implémenté sur n'importe quels réseaux, et avec n'importe quel schéma de stockage des tâches, au prix d'un coût important pour la victime potentielle. En effet, c'est elle qui aura la charge de gérer la concurrence entre les différents accès aux listes de tâches.

Enfin, nous pouvons remarquer que cette méthode s'approche des RPC décrits dans la Partie 5.2.1, mais où le choix du processus de destination ne se

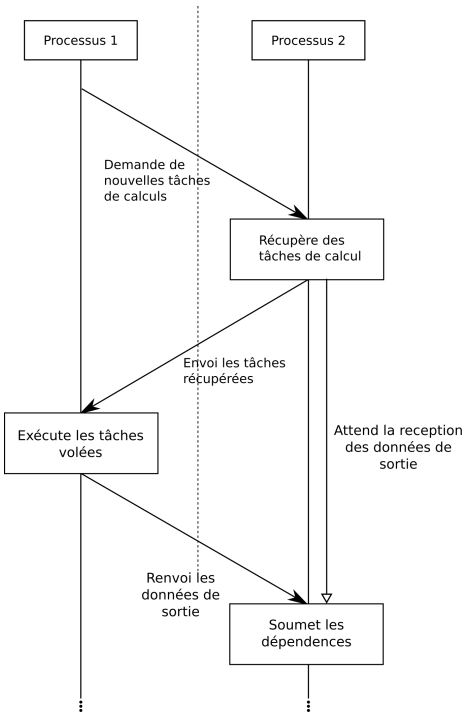


FIGURE 5.38 – Protocole de vol coopératif

ferait pas par le processus d'origine des tâches. Ici, ce serait les processus distants qui indiqueraient pouvoir exécuter des tâches supplémentaires. Cependant, ce serait le processus local qui serait en attente des communications et non pas le processus distant, ce qui permet de ne pas avoir à développer de nouvelles méthodes pour le choix de ce dernier.

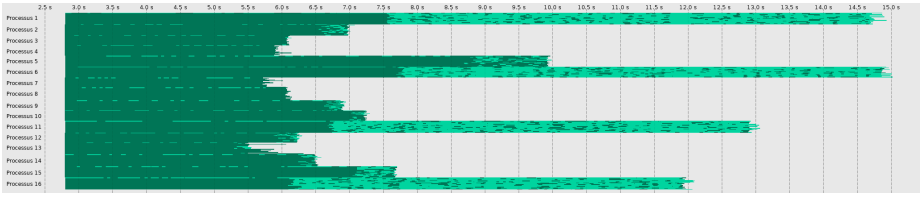
Ce type de vol de travail a été intégré dans notre solveur pour y équilibrer la partie de l'assemblage, permettant un gain de 48% dans les cas les plus extrèmes que nous avons testé.

Ce cas est illustré par la Figure 5.39, qui compare respectivement une exécution sans vol de travail avec une seconde exécution en ayant activé cette méthode de rééquilibrage. Ces deux exécutions ont eu lieu sur le calculateur TERA1000-1 présenté dans la Partie 2.5, et ont consisté en l'assemblage du cas du cône IEEE présenté dans la Partie 2.4.1.

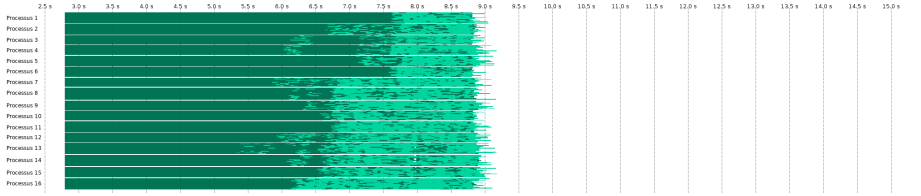
Les deux cas ont été exécutés sur 16 processus, utilisant 16 threads chacun. L'hyperthreading ayant été activé, les processus ont été disposés sur 4 nœuds de calcul. Les données ont été réparties parmi les différents processus suivant une distribution 2D bloc cyclique, sans utiliser d'autres méthodes de rééquilibrage statique comme les processus virtuels, ces derniers étant complexes à mettre en place sur l'étape d'assemblage.

Nous pouvons voir que le premier cas n'est rééquilibré que grâce à la bibliothèque OpenMP, qui permet d'obtenir une bonne répartition de charge aux sein des nœuds mais pas entre ceux-ci.

Le second quand à lui utilise, en plus d'OpenMP pour répartir les calculs sur les différents threads d'un processus, notre implémentation du vol de travail



(a) cas exécuté sans vol de travail



(b) Equilibrage de la phase d'assemblage avec des mécanismes de vol de travail

FIGURE 5.39 – Comparaison d'une trace d'exécution avec et sans vol de travail

pour l'équilibrage entre les nœuds, en sélectionnant les victimes au hasard.

Nous pouvons voir que le premier cas, illustré par la Figure 5.39a est globalement très déséquilibré : le processus le plus rapide a mis 3,3 secondes et le processus le plus lent a mis 12,1 secondes, le temps moyen est quand à lui de 5,8 secondes, ce qui indique un gain potentiel d'environ 50%.

Le second cas quand à lui, illustré par la Figure 5.39a. Le temps moyen est ici de 6,2 secondes alors que le processus le plus lent met 6,3 secondes pour terminer son exécution.

Si l'équilibrage n'est pas toujours aussi bon, cet exemple montre clairement l'intéret de ce type d'équilibrage dynamique dans le cas de l'assemblage de matrices hiérarchiques.

5.2.6.b Détermination d'un tag MPI unique

Dans la partie précédente nous avons vu un protocole permettant de transférer des tâches de manière dynamique entre deux processus. Ces transferts ont lieu de manière *a priori* non prévisible, et potentiellement concurrente.

Il est alors important que ces différents transferts ne puissent pas entrer en conflit les uns avec les autres, ou avec les autres communications de l'application. Pour cela la norme MPI utilise le concept de tag, qui permet de distinguer les communications provenant d'un même processus en utilisant un numéro unique. Il est alors nécessaire de sélectionner des tags uniques pour les différentes communications utilisées par le vol de tâches.

En effet, s'il est naturel de ne pas autoriser plusieurs vols simultanés, le renvoi des données de sortie est moins prévisible, et le retour de plusieurs tâches peut s'effectuer dans un ordre différent de celui du vol. Il est alors nécessaire de pouvoir différencier les différents vols.

Puisque la tâche victime d'un vol contient une information qui indique le tag à utiliser, il est possible de ne choisir ce tag uniquement au sein du processus victime, et il n'est pas nécessaire de disposer d'un protocole distribué entre les différents processus pour choisir un tag.

Une première solution serait d'affecter à chaque tâche un numéro unique. Cependant, les tâches étant majoritairement exécutées localement, l'espace des tags MPI pourraient être rapidement épuisés. Au delà des considérations pratiques au sein des implémentation de la pile MPI qui peut souffrir de l'utilisation d'un nombre considérable de tags, il est alors plus intéressant de n'allouer ces numéros qu'au moment du vol. De cette façon il n'y a pas de tags non utilisés, et ceux ayant déjà servis peuvent être réutilisés par la suite.

Enfin, nous pouvons noter que si cet algorithme permet de distinguer les différentes communications du vol de tâche à l'aide des tags MPI, il peut toujours exister des conflits avec les communications du reste de l'application qui n'a *a priori* pas de raison de se limiter dans l'utilisation des tags puisqu'il les tags utilisés par notre système de tâche sont invisibles pour l'utilisateur. Pour éviter ces éventuels conflits, il a suffit d'effectuer les différentes communications sur un communicateur dédié au vol de tâche. Celui-ci sera créé au début de l'exécution du vol de tâche, et détruit après sa terminaison.

5.2.6.c Serveur de requête

Nous pouvons observer que l'algorithme de vol de tâche coopératif, tel que décrit précédemment, va poster la réception du résultat des tâches volées dès leur transfert vers le voleur. La victime va alors *régulièrement* vérifier si le processus distant a commencé à renvoyer les données, ce qui lui occasionne un surcoût et peut devenir une source de contention très importante.

En effet, ces communications s'effectuant à l'intérieur de tâches de communication, pour tester leur complétion il est nécessaire de charger la tâche de communication, tester la complétion MPI, puis de passer à la tâche de communication suivante. Lorsque suffisamment de tâches de communication sont destinées à la réception de communications qui n'ont pas encore été commencées par l'envoyeur, ces étapes peuvent ralentir d'autres échanges de messages, qui seraient quant à eux prêts à être complétés.

Pour illustrer ce phénomène nous avons fait communiquer deux processus sur la partition skylake du calculateur Inti suivant le protocole suivant : le premier va poster n réceptions MPI, chacune sur un tag différent. Quand l'une d'elle est complétée, il va l'indiquer à l'émetteur. Le second processus va quant à lui sélectionner l'un de ces tags au hasard et effectuer une communication dessus. Il va ensuite attendre la confirmation du premier processus, ce qui permet de garantir que les messages ne soient pas envoyés de manière asynchrone à l'aide du mode *eager* de MPI.

La Figure 5.40 montre alors le temps d'exécution nécessaire en fonction du nombre de communications échangées. Ces test ont été lancés sur la partition skylake du calculateur Inti. La courbe rouge illustre le comportement de MPI sans prendre de dispositions particulières, et montre que le temps de communication passe d'environ un dixième de seconde pour 1000 requêtes à une cinquantaine de secondes pour 20 000 requêtes.

Nous proposons donc de mutualiser la vérification des communications via un *serveur de requêtes*. Pour cela, les tâches en attente de la complétion d'une communication sont associées avec cette dernière et sont confiées au serveur de requête. Celui-ci va par la suite se charger de vérifier si les requêtes qui lui ont été confiées ont été complétées, et de réinsérer les tâches dans leurs listes le cas échéant. De cette façon, les structures de données utilisées pour stocker les

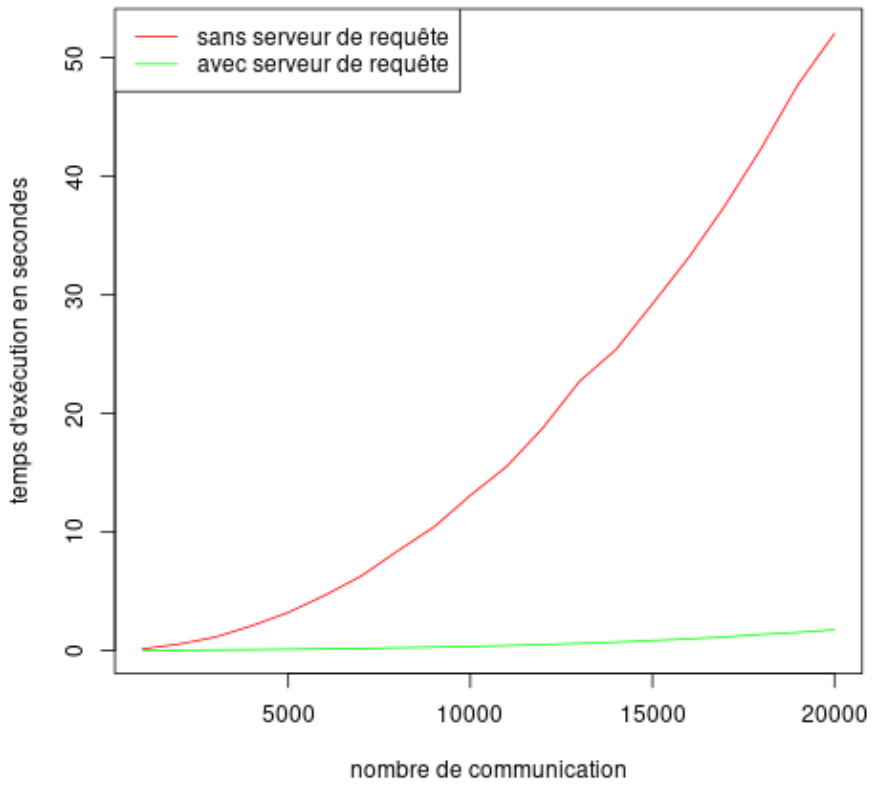


FIGURE 5.40 – Comparaison du temps nécessaire à l'exécution d'un grand nombre de communications, avec et sans serveur de requête

tâches ne sont pas sollicitées pour des communications non terminées. Un second avantage est la possibilité de tester l'ensemble des communications en un appel de fonction, comme avec la fonction *MPI_ Test_ any* par exemple. Cette possibilité permet de réduire à nouveau le temps passé à tester les communications MPI, et donc de réduire le coût des réceptions soumises en avance.

Toujours dans la Figure 5.40, la courbe verte montre le temps d'exécution nécessaire en fonction du nombre de communications échangées, en utilisant un serveur de requêtes cette fois. Nous pouvons voir que dans ce cas, la durée nécessaire pour effectuer 1000 communications est d'un peu moins d'un centième de seconde, tandis que la durée nécessaire pour 20 000 communications est d'un peu moins de deux secondes.

Nous pouvons donc observer que quel que soit le nombre de communications utilisées, ce serveur de requête permet d'être plus rapide d'au moins un ordre de grandeur dans le cas de cet exemple.

Nous pouvons également noter que les interactions avec le serveur de requêtes se fait au travers d'un nombre réduit de fonctions, qui sont indiquées dans la

```
struct req_server_ctx* init_req_server(void *, void*);
void destroy_req_server(struct req_server_ctx*);

/* Ajout d'une communication au serveur de requête, ainsi que de sa tâche
↪  associée */
void add_comm_request(struct req_server_ctx*, struct task*, MPI_Request);

/* Fonction appelée par le thread de communication, et qui permet de tester les
↪  communications et de libérer les tâches pouvant continuer leur exécution */
void check_req_server(struct req_server_ctx*);
```

FIGURE 5.41 – Interface du serveur de requête

Figure 5.41. Outre les fonctions d'initialisation et de destruction du module, il existe une fonction pour ajouter une communication et sa tâche associée, et une fonction qui permet de vérifier la complétion des communications et de libérer la tâche associée.

Enfin, nous pouvons noter que cette méthode peut s'appliquer dans d'autres contextes que le vol de travail, et a été appliquée avec succès dans d'autres parties de notre code.

En pratique, il s'agit d'une optimisation essentielle dans un système de tâche avec un parallélisme agressif qui peut conduire à générer des millions de tâches.

5.2.6.d Limitations

Dans cette partie nous avons présenté un modèle de vol de tâche généraliste, ainsi que deux problèmes à résoudre pour pouvoir utiliser cette méthode de manière efficace. Cependant, plusieurs problèmes sont intrinsèquement liés à ce type de vol.

En effet, lorsqu'un processus tente de voler du travail à un autre, il doit attendre que la victime soit prête à traiter sa demande. Si celle-ci est surchargée, alors cette attente peut devenir longue et limiter le gain d'équilibrage apporté par le vol de tâche. De plus, si un grand nombre de processus tentent de voler des tâches à une même victime, une contention peut apparaître, augmentant à nouveau le délai de réponse.

De plus cette attente peut survenir même si le processus ciblé ne possède pas assez de tâches pour un vol, rendant ce délai d'autant plus dommageable qu'il sera vain.

5.2.7 Vol de tâche one-sided

Dans cette partie nous allons présenter une nouvelle méthode de vol de tâche s'appuyant sur des communications one-sided et permettant de résoudre les problèmes évoqués dans la Partie 5.2.6.d. Nous commencerons par évoquer les principes utilisés par cette méthode de vol de tâche, puis nous discuterons des verrous bloquants et non bloquants. Enfin nous évoquerons les limites de cette nouvelle méthode.

5.2.7.a principe

Plutôt que de se reposer sur des communications *point à point* (e.g. `MPI_Send`), une autre solution est d'utiliser des communications *one sided* (e.g. `MPI_Put`). Celles-ci s'effectuant à l'aide d'un seul processus, il est possible de créer des protocoles de vol de tâche ne nécessitant pas la coopération de la victime, et donc ne dépendant pas de sa réactivité.

Avec ces communications, un processus distant peut accéder à la mémoire d'un autre processus en lecture et en écriture. Les voleurs peuvent donc examiner la liste de tâches des victimes potentielles pour déterminer eux-même si assez de tâches sont disponibles pour un vol. Ainsi, si ce n'est pas le cas le voleur passe au processus suivant sans impacter l'exécution de la victime, et sans être impacté par un éventuel ralentissement de celle-ci. En revanche, si assez de tâches sont disponibles, il est possible de manipuler les structures des listes de tâches pour supprimer une ou plusieurs tâches de la mémoire de la victime, et de les transférer vers sa propre mémoire. Une fois que ces tâches volées auront été exécutées, le voleur peut à nouveau utiliser les communications *one sided* pour inscrire dans la mémoire de la victime le résultat de ces tâches.

Enfin nous pouvons noter que cette méthode de communication permet d'utiliser des algorithmes conçus pour fonctionner en mémoire partagée, comme celui utilisé par le langage cilk [49]. Cet algorithme propose de stocker les tâches dans une liste à laquelle les processus peuvent accéder depuis les deux bouts. Chaque processus possède une liste à laquelle il peut accéder localement, et peut également accéder aux listes des autres processus.

Les accès locaux se font depuis la fin de la liste tandis que les accès distants se font depuis le début de la liste. Ces deux positions sont indiquées par leurs indices dans le tableau servant au stockage de la liste. Lors d'un accès local, si l'indice de fin est strictement supérieur à l'indice de début, nous pouvons être sûrs que les accès locaux ne sont pas entrés en collision avec les accès distants, et donc qu'aucun verrou n'est nécessaire. Dans le cas contraire l'opération est annulée, et peut être rejouée en ayant verrouillé la liste de tâches.

Les accès distant doivent quant à eux être protégés en verrouillant la liste de tâches, pour s'assurer qu'un seul processus tente de voler une tâche à un processus distant. Si l'indice de début devient strictement supérieur à l'indice de fin, alors nous pouvons déterminer que les accès locaux sont entrés en collision et donc que le vol doit être annulé.

Cet algorithme permet donc d'effectuer un maximum d'accès locaux sans devoir verrouiller la liste, ce qui est une opération coûteuse. De cette façon, les accès locaux ne sont pas impactés par les différents vols tant qu'il reste assez de tâches.

5.2.7.b Verrous non bloquants

Nous avons vu dans la partie précédente qu'il existe des algorithmes permettant de gérer la possibilité du vol de tâche sans qu'un processus ait besoin de manipuler des verrous pour accéder à sa propre liste de tâches. En revanche ces verrous restent nécessaires pour garantir que plusieurs voleurs n'entrent pas en collision.

De manière générale, l'Algorithme 14 résume les opérations à effectuer lors d'un vol : après avoir sélectionné une victime, un voleur va verrouiller sa liste

Algorithme 14 : Structure du vol de tâche avec des verrous bloquants

```
   Résultat : Sélectionne un processus et tente de lui voler une tâche de
              calcul
 1 v ← select_victim();
 2 lock(v);
 3 t ← steal(v);
 4 unlock(v);
 5 si t alors
 6 |   insert_task(t);
 7 |   Renvoyer SUCCESS;
 8 sinon
 9 |   Renvoyer FAILURE;
10 fin
```

de tâches, avant d'effectuer un vol. Une fois ce vol effectué, il va déverrouiller cette liste, avant d'insérer la tâche volée en cas de succès.

Nous pouvons noter que lorsqu'un grand nombre de processus tentent simultanément de voler des tâches à une même victime, cette prise de verrou peut devenir l'élément limitant des algorithmes de vol.

Algorithme 15 : Structure du vol de tâche avec des verrous non bloquants

```
   Résultat : Sélectionne un processus et tente de lui voler une tâche de
              calcul
 1 v ← select_victim();
 2 si trylock(v) alors
 3 |   t ← steal(v);
 4 |   unlock(v);
 5 |   si t alors
 6 |   |   insert_task(t);
 7 |   |   Renvoyer SUCCESS;
 8 |   sinon
 9 |   |   Renvoyer FAILURE;
10 |   fin
11 sinon
12 |   Renvoyer FAILURE;
13 fin
```

Une solution est d'utiliser des verrous non bloquants, comme le propose l'Algorithme 15. Ceux-ci sont similaires aux verrous normalement utilisés, mais possèdent une primitive supplémentaire : *trylock*. Cette nouvelle primitive permet d'obtenir l'accès exclusif au verrou si aucun autre processus ne le possédait. Dans le cas contraire, la primitive se termine immédiatement, en indiquant un échec.

Cette nouvelle primitive permet aux processus de ne pas être bloqué à l'entrée d'une section critique, et d'effectuer autre chose à la place. Dans le cas du

vol de tâche, cela permet de sélectionner une nouvelle victime. Si celle-ci n'est pas déjà bloquée par un voleur, alors un vol peut être effectué sans attendre l'accès aux listes de tâches.

Ce type de verrou n'étant pas nativement supporté par MPI, nous avons dû les implémenter, en nous appuyant sur les primitives *test_and_set*.

5.2.7.c Limitations

Nous avons vu cette Partie une deuxième méthode de vol de tâche. Celle-ci permet d'effectuer des transferts de tâches sans avoir besoin d'assistance de la victime, ce qui permet de ne pas être dépendant de sa réactivité. Nous allons maintenant lister plusieurs limitations qui rendent complexe son utilisation dans des codes industriels.

Une des limitations les plus importantes est la difficulté de gérer les dépendances entre les tâches. En effet, la méthode que nous avons présentée ne possède pas de mécanisme simple pour indiquer à une victime la complétion d'une tâche volée. C'est donc au voleur d'insérer les tâches filles. Nous pouvons tout d'abord noter que cette opération est normalement faite localement dans les différents algorithmes existants.

De plus, la manipulation des différentes méthodes de synchronisation des tâches, ainsi que l'insertion de ces dernières dans les listes de tâches exécutables demandent un grand nombre d'accès mémoire. Ces accès doivent être protégés d'opérations concurrentes de la part d'autres voleurs mais aussi de la victime. Les accès locaux de cette dernière doivent donc prendre en compte cette nouvelle concurrence, complexifiant grandement les algorithmes utilisés.

Une seconde limitation est la gestion des données d'entrée et de sortie des tâches. En effet celles-ci doivent se situer dans des zones accessibles aux accès RDMA, ce qui impose que leur allocation se fasse via les bibliothèques de communication. Avec le modèle proposé par MPI, il n'est cependant pas possible à un processus d'allouer de la mémoire en dehors d'une opération collective. Il est donc nécessaire de connaître la quantité de mémoire nécessaire pour les données de toutes les tâches volables. Dans le cas où la quantité exacte n'est pas connue, une borne supérieure pourra être utilisée, au prix d'un gaspillage de ressource.

Ces deux limitations réduisent grandement l'intérêt d'un modèle de vol de tâche distribué s'appuyant *uniquement* sur des accès mémoire distant.

5.2.8 Vol de tâche par rendez-vous

Dans cette partie, nous allons proposer une troisième méthode de vol de tâche s'appuyant sur un système de rendez-vous qui permet de résoudre les problèmes évoqués dans la Partie 5.2.7.c.

5.2.8.a Principe

Comme l'illustre la Figure 5.42, cette méthode de vol de tâche repose en partie sur le vol one-sided décrit dans la Partie 5.2.7. Cependant, au lieu de transférer l'intégralité de la tâche de calcul, ce nouveau protocole ne va transférer que l'identifiant de la tâche, ainsi que la fonction qu'elle est chargée d'exécuter. Le voleur va ensuite demander à la victime de lui envoyer les données d'entrée de la tâche volée. Cette demande pourra être envoyée à l'aide d'un *active message*

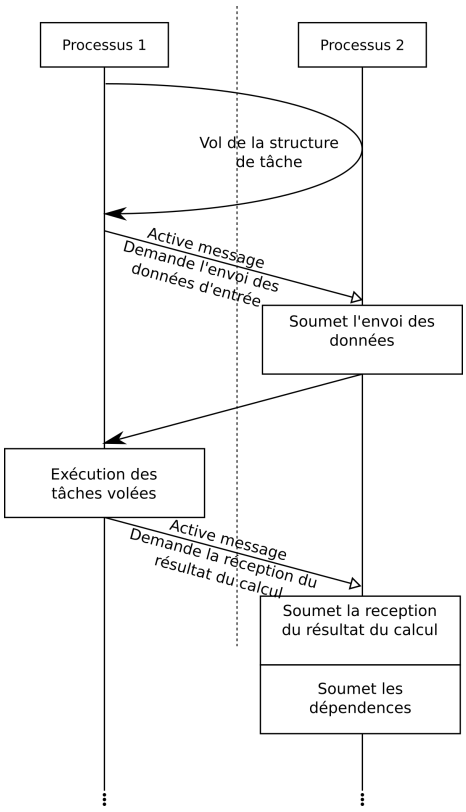


FIGURE 5.42 – Protocole de vol par rendez-vous

qui permet d'invoquer une fonction sur un processus distant. Ces *active messages* seront décrits plus précisément dans la Partie 5.2.8.b.

De cette façon, c'est au voleur de s'assurer qu'il reste suffisamment de travail pour un vol, et c'est lui qui aura la charge du surcoût des différentes synchronisations. En revanche, une fois la tâche sélectionnée, la victime va pouvoir envoyer des données stockées en dehors des zones MPI et de taille arbitraires, pour qu'elles soient utilisées en temps que données d'entrée des tâches volées.

Ces envois vont pouvoir se faire à l'aide des tâches de communication décrites dans la Section 3.5.2, ce qui permet de réutiliser les méthodes de recouvrement des communications déjà existantes, ainsi que d'opérer un choix sur les données transférées, certaines pouvant être communes aux différents processus.

De la même façon, une fois la tâche exécutée, le voleur va envoyer un nouvel *active message* pour indiquer sa complétion à la victime. La fonction appelée via cet *active message* va réceptionner les données produites, en les plaçant là où une exécution locale les aurait générées. De plus cette fonction va également débloquer les dépendances en utilisant les mécanismes locaux.

Ainsi, cette méthode de vol par rendez-vous permet de sélectionner une victime éligible pour du vol de travail, et de déterminer les tâches à voler, sans intervention de processus distant. De plus, l'utilisation d'*active messages* pour le transfert des données liées aux tâches permet d'utiliser des données arbitraires en entrée ou en sortie, sans être limité par leur taille comme pour le vol

one-sided.

Enfin, cette nouvelle méthode permet de conserver simplement les dépendances du système de tâche, ce qui permet de l'utiliser dans un code existant avec un minimum de modifications.

5.2.8.b Active messages

Dans la partie précédente nous nous sommes appuyés sur la notion d'*active message*. Nous allons maintenant les présenter plus en détail.

Un *active message* est un message qui va déclencher, chez le destinataire, l'exécution d'une fonction définie par l'envoyeur. Cette fonction peut typiquement consister à réceptionner des données envoyées via des communications pair à pair, ou à libérer des dépendances sur des données.

Cette méthode de communication n'étant pas définie dans la norme MPI, nous avons implémentés un système d'*active messages* à l'aide des préconisations de Schuchart *et al.* [81]. Dans cet article, les auteurs proposent un algorithme se basant sur les accès RDMA de MPI et n'utilisant pas de verrous.

Cet algorithme utilise deux listes de messages, l'une étant utilisée par les expéditeurs pour y écrire les nouveaux messages, l'autre étant utilisé par le destinataire pour lire les messages reçus.

Lorsque le destinataire a traité l'ensemble des messages de la liste de réception, les deux listes peuvent être interverties, pour traiter les messages déposés dans la liste d'envoi.

Cette structure sous forme de liste permet de recevoir plusieurs *active messages* sans devoir attendre que les précédents aient été traités par le destinataire. Nous pouvons également observer qu'intervertir les listes de lecture et d'écriture permet aux expéditeurs de toujours pouvoir inscrire de nouveaux *active messages*, sans devoir attendre le traitement de ceux déjà postés.

Enfin, les listes sont allouées au moment de l'initialisation du programme et ne peuvent donc pas être agrandies. Lorsque les listes sont pleines, l'envoi d'un *active message* n'est donc pas possible. Dans ce cas, l'expéditeur peut réessaye de poster l'*active message* jusqu'à ce que le récepteur ait traité les messages déjà placés dans ses listes de tâches à traiter. Un échec peu également être notifié à l'utilisateur qui aura la charge de demander un nouvel envoi ultérieurement.

L'utilisation de tâches interruptibles permettant simplement de mettre en pause une tâche de communication, nous avons choisi cette deuxième possibilité. Les *active messages* sont donc soumis depuis ces tâches. Si la soumission se termine sur un succès, la tâche passe à son étape suivante, dans le cas contraire elle va interrompre son exécution pour tenter à nouveau de poster un *active message* ultérieurement.

5.2.9 Support pour le langage Fortran

Les différents systèmes de vol de tâches que nous avons présentés ont été développés dans le langage C. Cependant certaines parties du code ont été implémentées dans le langage Fortran. Pour permettre de lier ces deux langages, le fortran dispose d'un binding C depuis la version fortran 2003. Cependant, les modèles mémoire n'étant pas identiques, les variables ne peuvent pas être nativement partagés d'un langage à un autre. Pour cela il est nécessaire de déclarer

les variables fortran comme compatibles avec le langage C pour qu'elles aient la même taille et que les données soient codées de la même façon.

De plus, le langage Fortran ne supporte pas les fonctions variadiques, qui sont cependant nécessaires pour supporter la soumission de tâches ayant un nombre arbitraire de paramètres.

Nous avons donc mis en place une interface permettant d'interfacer le système de tâche en C avec du code Fortran. Cette interface peut être vue comme un second système de tâches, auquel il manque le moteur d'exécution et la gestion des dépendances. À la place ce nouveau système se repose sur le système de tâches déjà existant, codé en C.

Par exemple, quand le code Fortran insère une tâche t_f, l'interface soumet une tâche t_c dans le système de tâche C, avec un pointeur vers la tâche t_f en paramètre. Puis, quand le système de tâche exécutera la tâche t_c, cette dernière appellera la tâche t_f, avec les paramètres qui auront été indiqués lors de la soumission.

Les arguments d'une tâche sont généralement fournis, en C, via les paramètres variadiques des fonctions de soumission. Dans le langage Fortran de telles fonctions n'existant pas, il n'est pas possible d'utiliser ce mécanisme. En revanche il existe des tableaux de taille variable, qui peuvent stocker des pointeurs vers des données arbitraires. En stockant les paramètres des tâches dans de tels tableaux il est possible de les transmettre au système de tâches, puis à la tâche de calcul ce qui permet de simuler ces fonctions variadiques.

De plus, ce système de tâches fait l'interface entre le code Fortran pour les fonctions utilisateur de transfert des données décrites dans la Partie 5.2.8.a, qui seront écrites en Fortran. Il est alors possible de voler des tâches Fortran, en créant les tâches nécessaires lors de la réception des paramètres.

Ainsi, grâce à cette interface, le système de tâches peut être utilisé depuis le langage Fortran sans nécessiter de modifications importantes du runtime qui aurait eu sinon à gérer le mélange de code C/Fortran.

Enfin, nous pouvons noter que la méthode décrite dans cette partie pourrait être réutilisée dans d'autres langages que le Fortran, à condition de disposer d'un interfaçage minimal avec le langage C. Il est notamment nécessaire de disposer de pointeurs de fonction pour pouvoir créer des tâches lors de l'exécution, de pouvoir indiquer des pointeurs vers un ensemble de données de taille non définie, tel qu'un tableau de taille variable ou des paramètres variadiques d'une fonction, pour indiquer les paramètres des tâches, et de pouvoir appeler des fonctions depuis ce langage vers le C et vice-versa, pour pouvoir interfacer les deux systèmes de tâches.

5.2.10 Bilan

Dans cette partie, nous avons présenté trois méthodes de vol de travail. La première, le vol coopératif consiste à la mise en place d'un protocole MPI qui permet de transférer les tâches et qui est déclenché par le destinataire. Cette première méthode est simple à implémenter et fonctionne avec tous les types de réseaux mais entraîne un surcoût pour la victime, y compris quant le vol échoue.

Cette première méthode a été utilisée avec succès pour rééquilibrer la partie d'assemblage de notre solveur, permettant de réduire presque totalement le déséquilibre de charge de cette partie, permettant de réduire sa durée d'exécution de 50% dans les cas les plus extrêmes.

La seconde méthode que nous avons présentée est le vol one-sided. Ce type de vol exploite les accès RDMA que permettent certains réseaux. À l'aide de ceux-ci il est possible d'effectuer un vol sans ralentir la victime. En revanche, cette méthode ne permet pas de voler des tâches ayant des paramètres ou des dépendances complexes, ces derniers étant difficilement manipulés par une interface RDMA.

Le troisième type de vol de tâche que nous avons présenté est le vol par rendez-vous. Cette méthode s'appuie sur le vol one-sided pour sélectionner les tâches, et utilise ensuite des active messages pour initier le transfert des paramètres. Cette méthode permet donc de limiter le coût d'un vol pour les victimes, tout en permettant de voler des tâches arbitraires.

Enfin, pour mettre en place ces méthodes de vol de travail, nous avons également dû mettre en place des méthodes pour déterminer la terminaison du calcul, quand les tâches peuvent migrer dynamiquement d'un processus à l'autre, mais aussi des verrous non bloquants sur les zones RDMA, et les actives messages en mémoire distribuée.

Dans la Section 5.2.5, nous avons montré que le mécanisme de délégation de tâches combiné avec un algorithme de consensus distribué permet de paralléliser de manière innovante la méthode des frontières immergées sur une machine à mémoire distribuée.

Nous avons montré sur plusieurs cas tests que ces méthodes peuvent servir à transférer des tâches semblables à celles utilisées dans un solveur hiérarchique, et donc à équilibrer la charge de ce type de calcul. Nous avons également montré que le vol par rendez vous est utilisable dans notre runtime, en demandant à l'utilisateur d'écrire les fonctions d'échange des données, qui sont automatiquement appelées lors des vols.

Cependant, pour pouvoir utiliser ces vols de tâches dans un solveur hiérarchique de manière efficace, il est nécessaire de ne pas voler les tâches de petite taille, mais seulement les plus longues. En effet, plus la durée d'exécution des tâches volées est grande, plus le vol sera efficace pour rééquilibrer la charge.

Les tâches traitant les feuilles de la matrice s'exécutant rapidement, il est donc nécessaire de ne pas les voler. Cependant les tâches de plus haut niveau posent un autre problème : en effet, à cause des dépendances hiérarchiques, les tâches générées peuvent libérer des dépendances pour d'autres tâches que les sœurs et les parents.

Il sera donc intéressant de permettre à ces dépendances de pouvoir être résolues de manière distribuée, ce qui permettra de conserver les gains du vol de tâche, tout en garantissant une bonne efficacité parallèle grâce au vol de travail.

Chapitre 6

Conclusion et perspectives

6.1 Contributions

L'objectif principal de cette thèse était d'étudier le rééquilibrage de charge pour des applications hiérarchiques distribuées, telles qu'un solveur direct pour des matrices compressées hiérarchiquement. Nous avons tout d'abord introduit la notion de processus virtuels, dans la Partie 5.1.2, permettant de redistribuer les blocs de la matrice hiérarchique tout en conservant une interface analogue aux distributions 2D bloc cycliques utilisées classiquement. Cela permet de passer d'une distribution canonique à une distribution optimisée avec un coût de développement minimal. Cette notion permet de conserver une certaine régularité dans les distributions, tout en autorisant une plus grande souplesse dans le choix de l'affectation des blocs aux différents processus.

Nous avons ensuite montré dans la Partie 5.1.4 que ces processus virtuels peuvent servir à rééquilibrer la consommation de mémoire d'un solveur hiérarchique distribué. Nous avons également vu que ce rééquilibrage en mémoire permet également de mieux rééquilibrer les temps de calcul. Dans la Partie 5.1.5 nous avons vu qu'en supposant les temps de calcul des différents blocs de la matrice connus, la charge des différents processus pouvait être rééquilibrée de manière plus efficace.

Puis dans la Partie 5.1.6 nous avons montré comment estimer le temps d'exécution d'un solveur hiérarchique, en utilisant des modèles de performance pour déterminer la durée des tâches. Dans la Partie 5.1.8 nous avons introduit la notion de compteur de performances asynchrone, permettant de connaître la durée d'exécution et le nombre d'opérations d'une tâche et de ses descendants dans un contexte de soumission asynchrone de tâches récursives.

Enfin, dans les parties 5.2.6 à 5.2.8, nous avons présenté plusieurs méthodes de vol de travail en mémoire distribuée, ainsi que les problématiques liées à ce type de rééquilibrage. Plus précisément nous avons montré dans la Partie 5.2.6 qu'un vol de travail coopératif est possible mais possède des limitations intrinsèques dans son passage à l'échelle. Dans la Partie 5.2.7 nous avons montré que s'il est possible d'implémenter un vol one-sided, ce type de vol ne permet pas de transférer des tâches aux paramètres ou aux dépendances complexes. Dans la Partie 5.2.8 nous avons enfin présenté une méthode de vol par rendez-vous qui permet de voler des tâches arbitraires tout en n'imposant qu'un surcoût limité pour la victime.

6.2 Perspectives

Dans cette thèse nous avons présenté plusieurs méthodes permettant de rééquilibrer la charge d'applications irrégulières telles qu'un solveur hiérarchique.

Nous allons maintenant présenter les perspectives qu'apporte cette thèse pour de futures recherches.

Si l'utilisation de techniques statiques s'avère d'ores et déjà effectives, il n'a pas été possible d'exploiter les techniques de vol de tâche au sein de l'application industriel au moment de la rédaction de ce document. Il sera donc nécessaire d'industrialiser les techniques que nous avons proposé afin de les utiliser effectivement lors de la phase de factorisation du solveur hiérarchique, notamment. Il faudra pour cela d'abord résoudre des limitations techniques liés à des limites des couches réseaux, et aux implémentation de MPI à notre disposition. Le solveur *actuel* présente également un inconvénient majeur vis à vis du vol de tâche : pour optimiser la quantité de parallélisme, nous avons opté pour une approche où des tâches de très fines granularités sont soumises avec des dépendances hiérarchiques. La mise à jour d'un bloc hiérarchique ne constitue donc pas une unique tâche mais un graphe de tâches qu'il n'est pas forcément efficace de voler efficacement. Résoudre ce problème sans remettre en cause les dépendances hiérarchiques nécessite de soumettre des tâches récursives avec des dépendances hiérarchiques, c'est à dire être capable de soumettre des tâches dans les tâches qui accèdent à des données à l'aide de dépendances hiérarchiques. Cette limitation difficile devrait notamment être levée grâce à l'introduction de contextes hiérarchiques proposés récemment [68].

Une première perspective intéressante sera de continuer à développer des méthodes de décision pour la distribution des processus virtuels parmi les processus physiques, ainsi que sur la forme de la grille. En effet, une meilleure compréhension de l'influence de ces paramètres sur le rééquilibrage permettrait de décider d'une distribution au début de la factorisation en se basant sur la quantité de mémoire des différents blocs.

De plus, il sera intéressant de développer une méthode permettant de déterminer la liste des différentes tâches nécessaires à la factorisation de chaque bloc, ce qui permettra, à l'aide des modèles de performances proposés dans la Partie 5.1.6 d'estimer leur temps de calcul et donc d'effectuer des redistributions en temps au début de la factorisation, permettant de se passer du système d'oracle utilisé jusqu'ici. Le nombre très important de tâches que l'on doit considérer rend cette tâche particulièrement difficile, et il sera probablement nécessaire prendre les décisions à une granularité adaptée, qui n'est pas forcément la granularité du graphe de tâche final.

Enfin, il sera intéressant de prendre en compte la priorité des tâches dans le vol de travail. En effet, jusqu'ici, nous n'avons pas pris en compte ce paramètre dans la sélection des tâches lors du vol. Cette décision est complexe à prendre en compte car d'une part les tâches les moins prioritaires sont généralement celles dont la rapidité d'exécution est la moins critique, et donc qui ne sont pas forcément pertinentes pour un vol. À l'inverse les tâches les plus prioritaires sont celles à exécuter le plus rapidement. Cependant, la durée nécessaire au transfert de la tâche lors du vol, puis aux données produites peut être supérieure au temps qu'il aurait fallu à la victime pour sélectionner et commencer l'exécution de la tâche volée. Dans ce cas le vol peut retarder la libération des dépendances, et donc ralentir le calcul.

De plus, il sera nécessaire de développer un système permettant de prioriser certaines tâches par rapport aux autres lors du vol pour tirer parti de ce système de priorité. En effet, les systèmes présentés ne disposent que d'une seule liste de tâches, ce qui ne laisse pas aux voleurs la possibilité de sélectionner la tâche

volée. Une des possibilités sera par exemple de mettre en place plusieurs listes de tâches volables, en les classant par priorité. De cette façon il sera possible aux voleurs de sélectionner les tâches les plus pertinentes pour l'efficacité du rééquilibrage.

Bibliographie

[1] classement du top500 de juin 2021. https ://top500.org/lists/top500/list/2021/06/.

[2] Emmanuel Agullo, Olivier Aumage, Mathieu Faverge, Nathalie Furmento, Florent Pruvost, Marc Sergent, and Samuel Paul Thibault. Achieving high performance on supercomputers with a sequential task-based programming model. *IEEE Transactions on Parallel and Distributed Systems*, 2017.

[3] Emmanuel Agullo, Bérenger Bramas, Olivier Coulaud, Luka Stanisic, and Samuel Thibault. Modeling Irregular Kernels of Task-based codes : Illustration with the Fast Multipole Method. Research Report RR-9036, INRIA Bordeaux, February 2017.

[4] Kadir Akbudak, Hatem Ltaief, Aleksandr Mikhalev, and David Keyes. Tile low rank Cholesky factorization for climate/weather modeling applications on manycore architectures. In Julian M. Kunkel, Rio Yokota, Pavan Balaji, and David Keyes, editors, *High Performance Computing*, pages 22–40, Cham, 2017. Springer International Publishing.

[5] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, 1967.

[6] Patrick Amestoy, Cleve Ashcraft, Olivier Boiteau, Alfredo Buttari, Jean-Yves L'Excellent, and Clément Weisbecker. Improving multifrontal methods by means of block low-rank representations. *SIAM Journal on Scientific Computing*, 37(3) :A1451–A1474, 2015.

[7] Patrick R. Amestoy, Alfredo Buttari, Jean-Yves L'Excellent, and Theo Mary. Performance and scalability of the block low-rank multifrontal factorization on multicore architectures. *ACM Trans. Math. Softw.*, 45(1) :2 :1–2 :26, February 2019.

[8] Patrick R Amestoy, Iain S Duff, Jean-Yves L'Excellent, and Jacko Koster. MUMPS : a general purpose distributed memory sparse solver. In *International Workshop on Applied Parallel Computing*, pages 121–130. Springer, 2000.

[9] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[10] Cédric Augonnet, Olivier Aumage, Nathalie Furmento, Raymond Namyst, and Samuel Thibault. StarPU-MPI : Task programming over clusters of machines enhanced with accelerators. In *European MPI Users' Group Meeting*, pages 298–299. Springer, 2012.

[11] Cédric Augonnet, David Goudin, Matthieu Kuhn, Xavier Lacoste, Raymond Namyst, and Pierre Ramet. A hierarchical fast direct solver for distributed memory machines with manycore nodes. Technical report, CEA/DAM ; Total E&P ; Université de Bordeaux, 2019.

[12] Cédric Augonnet, Samuel Thibault, and Raymond Namyst. Automatic Calibration of Performance Models on Heterogeneous Multicore Architectures.

In *HPPC - Proceedings of the International Euro-Par Workshops, Highly Parallel Processing on a Chip*, volume 6043 of *Lecture Notes in Computer Science*, pages 56–65, Delft, The Netherlands, August 2009. Springer.

[13] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU : A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *CCPE - Concurrency and Computation : Practice and Experience, Special Issue : Euro-Par 2009*, 23 :187–198, February 2011.

[14] Satish Balay, Kris Buschelman, William D Gropp, Dinesh Kaushik, Matthew G Knepley, L Curfman McInnes, Barry F Smith, and Hong Zhang. PETSc. *See http ://www. mcs. anl. gov/petsc*, 2001.

[15] Josh Barnes and Piet Hut. A hierarchical o (N log N) force-calculation algorithm. *nature*, 324(6096) :446–449, 1986.

[16] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion : Expressing locality and independence with logical regions. In *SC'12 : Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2012.

[17] Olivier Beaumont, Vincent Boudet, Fabrice Rastello, and Yves Robert. Load balancing strategies for dense linear algebra kernels on heterogeneous two-dimensional grids. In *14th International Parallel and Distributed Processing Symposium (IPDPS'2000)*, pages 783–792, Cancun, Mexico, 2000. IEEE Computer Society Press.

[18] Mario Bebendorf. *Hierarchical Matrices - A Means to Efficiently Solve Elliptic Boundary Value Problems*, volume 63 of *Lecture Notes in Computational Science and Engineering*. Springer, 2008.

[19] Andrew D Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, 2(1) :39–59, 1984.

[20] L Susan Blackford, Jaeyoung Choi, Andy Cleary, Eduardo D'Azevedo, James Demmel, Inderjit Dhillon, Jack Dongarra, Sven Hammarling, Greg Henry, Antoine Petitet, et al. *ScaLAPACK users' guide*. SIAM, 1997.

[21] Robert D Blumofe and Charles E Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5) :720–748, 1999.

[22] Dan Bonachea and Paul Hargrove. GASNet specification, v1. 8.1. Technical report, Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 2017.

[23] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack J Dongarra. Parsec : Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering*, 15(6) :36–45, 2013.

[24] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Pierre Lemarinier, and Jack Dongarra. DAGuE : A generic distributed DAG engine for high performance computing. *Parallel Computing*, 38(1-2) :37–51, 2012.

[25] George Bosilca, Aurelien Bouteiller, Thomas Herault, Pierre Lemarinier, Narapat Ohm Saengpatsa, Stanimire Tomov, and Jack J Dongarra. Per-

formance portability of a GPU enabled factorization with the dague framework. In *2011 IEEE International Conference on Cluster Computing*, pages 395–402. IEEE, 2011.

[26] Thibault Bridel-Bertomeu. Immersed boundary conditions for hypersonic flows using ENO-like least-square reconstruction. *Computers & Fluids*, 215 :104794, 2021.

[27] Javier Bueno, Luis Martinell, Alejandro Duran, Montse Farreras, Xavier Martorell, Rosa M Badia, Eduard Ayguade, and Jesús Labarta. Productive cluster programming with ompss. In *European Conference on Parallel Processing*, pages 555–566. Springer, 2011.

[28] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1) :38–53, 2009.

[29] William W Carlson, Jesse M Draper, David E Culler, Kathy Yelick, Eugene Brooks, and Karen Warren. Introduction to UPC and language specification. Technical report, Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, 1999.

[30] Rocío Carratalá-Sáez, Mathieu Faverge, Grégoire Pichon, Guillaume Sylvand, and Enrique S Quintana-Ortí. Tiled Algorithms for Efficient Task-Parallel H-Matrix Solvers. Research Report RR-9327, Inria, February 2020.

[31] Bradford L Chamberlain, David Callahan, and Hans P Zima. Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications*, 21(3) :291–312, 2007.

[32] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. Introducing OpenSHMEM : SHMEM for the PGAS community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, pages 1–3, 2010.

[33] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. X10 : an object-oriented approach to non-uniform cluster computing. *Acm Sigplan Notices*, 40(10) :519–538, 2005.

[34] Chao Chen, Hadi Pouransari, Sivasankaran Rajamanickam, Erik G. Boman, and Eric Darve. A distributed-memory hierarchical solver for general sparse linear systems. *Parallel Computing*, 74 :49 – 64, 2018. Parallel Matrix Algorithms and Applications (PMAA'16).

[35] Jaeyoung Choi, Jack Dongarra, Susan Ostrouchov, Antoine Petitet, David Walker, and R Clinton Whaley. A proposal for a set of parallel basic linear algebra subprograms. In *International Workshop on Applied Parallel Computing*, pages 107–114. Springer, 1995.

[36] Philippe G Ciarlet. *Introduction a l'analyse numerique matricielle et a l'optimisation PG Ciarlet*. masson, 1990.

[37] Ronald Coifman, Vladimir Rokhlin, and Stephen Wandzura. The fast multipole method for the wave equation : A pedestrian prescription. *IEEE Antennas and Propagation magazine*, 35(3) :7–12, 1993.

[38] Melvin E Conway. Design of a separable transition-diagram compiler. *Communications of the ACM*, 6(7) :396–408, 1963.

[39] Michel Cosnard and Emmanuel Jeannot. Compact dag representation and its dynamic scheduling. *Journal of Parallel and Distributed Computing*, 58(3) :487–514, 1999.

[40] O Dahl. Simula 67 common base language. *Technical report*, pages S–22, 1970.

[41] Kieran Delamotte. *Une étude du rang du noyau de l'équation de Helmholtz : application des H-matrices à l'EFIE*. PhD thesis, Sorbonne Paris Cité, 2016.

[42] Saïd Derradji, Thibaut Palfer-Sollier, Jean-Pierre Panziera, Axel Poudes, and François Wellenreiter Atos. The BXI interconnect architecture. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pages 18–25. IEEE, 2015.

[43] James Dinan, D Brian Larkins, Ponnuswamy Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. Scalable work stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11. IEEE, 2009.

[44] Jack Dongarra, Mark Gates, Azzam Haidar, Jakub Kurzak, Piotr Luszczek, Panruo Wu, Ichitaro Yamazaki, Asim YarKhan, Maksims Abalenkovs, Negin Bagherpour, et al. PLASMA : Parallel linear algebra software for multicore using OpenMP. *ACM Transactions on Mathematical Software (TOMS)*, 45(2) :1–35, 2019.

[45] Jack Dongarra and David Walker. The design of linear algebra libraries for high performance computers. 05 1997.

[46] Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. Ompss : a proposal for programming heterogeneous multi-core architectures. *Parallel processing letters*, 21(02) :173–193, 2011.

[47] Nissim Francez and Michael Rodeh. Achieving distributed termination without freezing. *IEEE Transactions on Software Engineering*, (3) :287–292, 1982.

[48] Valérie Frayssé, Luc Giraud, Serge Gratton, and Julien Langou. Algorithm 842 : A set of GMRES routines for real and complex arithmetics on high performance computers. *ACM Transactions on Mathematical Software (TOMS)*, 31(2) :228–238, 2005.

[49] Matteo Frigo, Charles E Leiserson, and Keith H Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 212–223, 1998.

[50] Thierry Gautier, Joao VF Lima, Nicolas Maillard, and Bruno Raffin. Xkaapi : A runtime system for data-flow task programming on heterogeneous architectures. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 1299–1308. IEEE, 2013.

[51] W. Hackbusch. A sparse matrix arithmetic based on H-matrices. part i : Introduction to H-matrices. *Computing*, 62(2) :89–108, May 1999.

[52] Pascal Hénon, Pierre Ramet, and Jean Roman. PaStiX : A parallel sparse direct solver based on a static scheduling for mixed 1d/2d block distributions. In *International Parallel and Distributed Processing Symposium*, pages 519–525. Springer, 2000.

[53] Pascal Hénon, Pierre Ramet, and Jean Roman. PaStiX : a high-performance parallel direct solver for sparse symmetric positive definite systems. *Parallel Computing*, 28(2) :301–321, 2002.

[54] Mathias Jacquelin, Esmond Ng, Katherine Yelick, and Yili Zheng. symPACK : a solver for sparse symmetric matrices.

[55] Pritish Jetley, Filippo Gioachin, Celso Mendes, Laxmikant V Kale, and Thomas Quinn. Massively parallel cosmological simulations with changa. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–12. IEEE, 2008.

[56] Laxmikant V. Kalé and Milind A. Bhandarkar. Structured dagger : A coordination language for message-driven programming. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *Euro-Par'96 Parallel Processing*, pages 646–653, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[57] Laxmikant V Kale and Sanjeev Krishnan. Charm++ a portable concurrent object oriented system based on C++. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 91–108, 1993.

[58] George Karypis and Vipin Kumar. METIS : A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. 1997.

[59] Charles H Koelbel, David B Loveman, Robert S Schreiber, Guy Lewis Steele Jr, and Mary Zosel. *The high performance Fortran handbook*. MIT press, 1994.

[60] David P Koester, Sanjay Ranka, and Geoffrey C Fox. A parallel Gauss-seidel algorithm for sparse power system matrices. In *Supercomputing'94 : Proceedings of the 1994 ACM/IEEE Conference on Supercomputing*, pages 184–193. IEEE, 1994.

[61] Ronald Kriemann. H-lu factorization on many-core systems. *Computing and Visualization in Science*, 16(3) :105–117, 2013.

[62] Aleksey Nikolaevich Krylov. On the numerical solution of the equation by which in technical questions frequencies of small oscillations of material systems are determined. *Izvestija AN SSSR (News of Academy of Sciences of the USSR), Otdel. mat. i estest. nauk*, 7(4) :491–539, 1931.

[63] Chuck L Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3) :308–323, 1979.

[64] Xavier Leroy. https ://web.archive.org/web/19961128030318/http ://pauillac.inria.fr/ xleroy/linuxthreads/, 1996.

[65] Xiaoye S Li and James W Demmel. SuperLU_DIST : A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Transactions on Mathematical Software (TOMS)*, 29(2) :110–140, 2003.

[66] Benoît Lizé. *Résolution directe rapide pour les éléments finis de frontière en électromagnétisme et acoustique : H-Matrices. Parallélisme et applications industrielles*. PhD thesis, Université Paris-Nord-Paris XIII, 2014.

[67] Loris Marchal. *Memory and data aware scheduling.* Habilitation à diriger des recherches, École Normale Supérieure de Lyon, March 2018.

[68] Thomas Morin. Extension du modèle STF à travers les dépendances hiérarchiques. rapport de stage. Technical report, CEA CESTA, 2021.

[69] Robert W Numrich and John Reid. Co-array Fortran for parallel programming. In *ACM Sigplan Fortran Forum*, volume 17, pages 1–31. ACM New York, NY, USA, 1998.

[70] Yu Pei, George Bosilca, Ichitaro Yamazaki, Akihiro Ida, and Jack Dongarra. Evaluation of programming models to address load imbalance on distributed multi-core CPUs : A case study with block low-rank factorization. In *PAW-ATM Workshop at SC19*, Denver, CO, 2019-11 2019. ACM, ACM.

[71] François Pellegrini and Jean Roman. Scotch : A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *International Conference on High-Performance Computing and Networking*, pages 493–498. Springer, 1996.

[72] Gregory F Pfister. An introduction to the infiniband architecture. *High performance mass storage and parallel I/O*, 42(617-632) :10, 2001.

[73] Grégoire Pichon, Eric Darve, Mathieu Faverge, Pierre Ramet, and Jean Roman. Sparse supernodal solver using block low-rank compression : Design, performance and analysis. *Journal of Computational Science*, 27 :255 – 270, 2018.

[74] Alex Pothen and Chunguang Sun. A mapping algorithm for parallel sparse Cholesky factorization. *SIAM Journal on Scientific Computing*, 14(5) :1253–1257, 1993.

[75] James Reinders. *Intel threading building blocks : outfitting C++ for multi-core processor parallelism.* " O'Reilly Media, Inc.", 2007.

[76] Remcom. RCS Analysis of 3D Bodies of Revolution. https ://www.remcom.com/examples/rcs-analysis-of-3d-bodies-of-revolution.html.

[77] Heinz Rutishauser. The Jacobi method for real symmetric matrices. *Numerische Mathematik*, 9(1) :1–10, 1966.

[78] Youcef Saad. Highly parallel preconditioners for general sparse matrices. In *Recent Advances in Iterative Methods*, pages 165–199. Springer, 1994.

[79] Youcef Saad and Martin H Schultz. GMRES : A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on scientific and statistical computing*, 7(3) :856–869, 1986.

[80] Kevin Sala, Jesus Labarta, and Vicenç Beltran. Task-aware MPI (TAMPI) and OpenMP. JLESC Workshop, 2019.

[81] Joseph Schuchart, Aurelien Bouteiller, and George Bosilca. Using MPI-3 RMA for active messages. In *2019 IEEE/ACM Workshop on Exascale MPI (ExaMPI)*, pages 47–56. IEEE Computer Society, 2019.

[82] T. Senior. The backscattering cross section of a cone-sphere. *IEEE Transactions on Antennas and Propagation*, 13(2) :271–277, 1965.

[83] Luka Stanisic, Emmanuel Agullo, Alfredo Buttari, Abdou Guermouche, Arnaud Legrand, Florent Lopez, and Brice Videau. Fast and Accurate Simulation of Multithreaded Sparse Linear Algebra Solvers. In *The 21st IEEE International Conference on Parallel and Distributed Systems*, Melbourne, Australia, December 2015.

[84] Guy L Steele Jr. Parallel programming and parallel abstractions in fortress. In *FLOPS*, page 1, 2006.

[85] Samuel Thibault. *On runtime systems for task-based programming on heterogeneous platforms*. PhD thesis, Université de Bordeaux, 2018.

[86] Richard S Varga. p-CYCUC matrices : A generalization of the young-frankel successive overrelaxation scheme. *SUPPORTING INSTITUTIONS*, page 617, 1959.

[87] Thorsten Von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net : A user-level network interface for parallel and distributed computing. *ACM SIGOPS Operating Systems Review*, 29(5) :40–53, 1995.

[88] Thorsten Von Eicken, David E Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages : a mechanism for integrated communication and computation. *ACM SIGARCH Computer Architecture News*, 20(2) :256–266, 1992.

[89] Lawrence M Widrow and John Dubinski. Equilibrium disk-bulge-halo models for the milky way and andromeda galaxies. *The Astrophysical Journal*, 631(2) :838, 2005.

[90] A. C. WOO. Benchmark radar targets for the vaildation of computational electromagnetics programs. *IEEE Antenna and Propagation Magazine*, 35(1) :86, 1993.

[91] Asim YarKhan, Jakub Kurzak, and Jack Dongarra. Quark users' guide : Queueing and runtime for kernels. *University of Tennessee Innovative Computing Laboratory Technical Report ICL-UT-11-02*, 2011.

[92] Asim YarKhan, Jakub Kurzak, Piotr Luszczek, and Jack Dongarra. Porting the PLASMA numerical library to the OpenMP standard. *International Journal of Parallel Programming*, 45(3) :612–633, 2017.

[93] Katherine Yelick, Dan Bonachea, Wei-Yu Chen, Phillip Colella, Kaushik Datta, Jason Duell, Susan L Graham, Paul Hargrove, Paul Hilfinger, Parry Husbands, et al. Productivity and performance using partitioned global address space languages. In *Proceedings of the 2007 international workshop on Parallel symbolic computation*, pages 24–32, 2007.