HAL
open science

Passage à l'echelle d'un support d'exécution à base de tâches pour l'algèbre linéaire dense

Marc Sergent

To cite this version:

Marc Sergent. Passage à l'echelle d'un support d'exécution à base de tâches pour l'algèbre linéaire dense. Autre [cs.OH]. Université de Bordeaux, 2016. Français. NNT : 2016BORD0372 . tel-01483666

HAL Id: tel-01483666

https://theses.hal.science/tel-01483666v1

Submitted on 6 Mar 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

par **Marc Sergent**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Passage à l'échelle d'un support d'exécution à base de tâches pour l'algèbre linéaire dense

---

Date de soutenance : 8 Décembre 2016

Devant la commission d'examen composée de :

M. Patrick AMESTOY ..	Professeur des Universités, Université de Toulouse ...	Examinateur
M. Olivier AUMAGE ...	Chargé de Recherche, Inria	Encadrant
M. David GOUDIN	Ingénieur Chercheur, CEA	Directeur
M. Guillaume LATU ...	Maître de Conférences, Université de Strasbourg	Examinateur
M. Pierre MANNEBACK .	Professeur des Universités, Université de Mons, Belgique	Rapporteur
M. Jean-François MEHAUT	Professeur des Universités, Université Grenoble Alpes .	Rapporteur
M. Raymond NAMYST ..	Professeur des Universités, Université de Bordeaux ..	Directeur
M. Samuel THIBAULT ..	Maître de Conférences, Université de Bordeaux	Encadrant

– 2016 –

Résumé La complexification des architectures matérielles pousse vers l'utilisation de paradigmes de programmation de haut niveau pour concevoir des applications scientifiques efficaces, portables et qui passent à l'échelle. Parmi ces paradigmes, la programmation par tâches permet d'abstraire la complexité des machines en représentant les applications comme des graphes de tâches orientés acycliques (DAG). En particulier, le modèle de programmation par tâches soumises séquentiellement (STF) permet de découpler la phase de soumission des tâches, séquentielle, de la phase d'exécution parallèle des tâches. Même si ce modèle permet des optimisations supplémentaires sur le graphe de tâches au moment de la soumission, il y a une préoccupation majeure sur la limite que la soumission séquentielle des tâches peut imposer aux performances de l'application lors du passage à l'échelle. Cette thèse se concentre sur l'étude du passage à l'échelle du support d'exécution StarPU (développé à Inria Bordeaux dans l'équipe STORM), qui implémente le modèle STF, dans le but d'optimiser les performances d'un solveur d'algèbre linéaire dense utilisé par le CEA pour faire de grandes simulations 3D. Nous avons collaboré avec l'équipe HiePACS d'Inria Bordeaux sur le logiciel Chameleon, qui est une collection de solveurs d'algèbre linéaire portés sur supports d'exécution à base de tâches, afin de produire un solveur d'algèbre linéaire dense sur StarPU efficace et qui passe à l'échelle jusqu'à 3 000 cœurs de calcul et 288 accélérateurs de type GPU du supercalculateur TERA-100 du CEA-DAM.

Title Scalability of a task-based runtime system for dense linear algebra applications

Abstract The ever-increasing supercomputer architectural complexity emphasizes the need for high-level parallel programming paradigms to design efficient, scalable and portable scientific applications. Among such paradigms, the task-based programming model abstracts away much of the architecture complexity by representing an application as a Directed Acyclic Graph (DAG) of tasks. Among them, the Sequential-Task-Flow (STF) model decouples the task submission step, sequential, from the parallel task execution step. While this model allows for further optimizations on the DAG of tasks at submission time, there is a key concern about the performance hindrance of sequential task submission when scaling. This thesis' work focuses on studying the scalability of the STF-based StarPU runtime system (developed at Inria Bordeaux in the STORM team) for large scale 3D simulations of the CEA which uses dense linear algebra solvers. To that end, we collaborated with the HiePACS team of Inria Bordeaux on the Chameleon software, which is a collection of linear algebra solvers on top of task-based runtime systems, to produce an efficient and scalable dense linear algebra solver on top of StarPU up to 3,000 cores and 288 GPUs of CEA-DAM's TERA-100 cluster.

Keywords High performance computing, parallel programming models, task-based programming, distributed computing, run-time systems

Mots-clés Calcul haute performance, modèles de programmation parallèle, programmation par tâches, calcul distribué, supports d'exécution

Laboratoire d'accueil / Hosting Laboratory Inria Bordeaux Sud-Ouest, 200 Avenue de la Vieille Tour, 33400 Talence

À ma mère,
qui nous a quittés trop tôt.

Remerciements

"Je voulais vous dire quelque chose. [...] Pour quelqu'un comme moi qui ai facilement tendance à la dépression, c'est très important ce que vous faites. [...] C'est systématiquement débile, mais c'est toujours inattendu. Et ça, c'est très important pour la ... la santé du ... du cigare. Je ne vous embête plus, allez-y."

– Arthur, *Kaamelott Livre II, Unagi II*

Je pense qu'il est important de commencer ces remerciements par dire que rien de tout ceci n'aurait été possible si je n'avais pas été aussi bien entouré. Je remercie donc tout d'abord Raymond, qui m'a donné l'envie en Master 1, quand je suis arrivé à Bordeaux, de faire du HPC (et tu t'en es souvenu car tu en as parlé pendant la soutenance, et ça m'a beaucoup touché !) quand j'étais perdu et que je me demandais ce que je faisais en filière GL "architecture distribuée". Je remercie ensuite David qui m'a, on peut le dire, ouvert les yeux sur les réalités du domaine du HPC, et au contact de qui j'ai appris énormément, aussi bien sur un plan professionnel que personnel. Je remercie ensuite mes encadrants de thèse, Samuel et Olivier, de m'avoir accompagné au quotidien durant cette thèse, et d'avoir supporté mes problèmes de communication multiples !

Je remercie également les membres de mon jury : Jean-François et Pierre (qui est venu de Belgique !) pour avoir accepté de relire ce manuscrit, Guillaume et Patrick pour avoir accepté d'assister à ma soutenance de thèse.

Je tiens ensuite à remercier les encadrants plus *officieux* mais qui ont toujours été présents pour m'aider. Je pense notamment à Emmanuel, qui m'a énormément appris sur les aspects numériques de ma thèse et qui a toujours été une oreille attentive et franche quand il le fallait. Je pense également à Cédric qui m'a tellement appris et guidé que je ne saurais même pas par où commencer. Tes duos musicaux avec Xavier resteront éternellement dans ma mémoire.

Je tiens également à remercier tous mes collègues qui ont rendu cette aventure belle et mémorable. Je vais tenter d'être le plus exhaustif possible, mais il est possible que j'en oublie et je m'en excuse par avance. Je vais commencer par parler du CEA-CESTA, et remercier Xavier C. de m'avoir acueilli dans son

service, Agnès P. la sportive, Muriel S. qui a réussi à faire nettoyer son bureau à Cédric, Olivier C. le plus grand fan de sa fille, Pierre B. grand militant du déménagement, Anne-Pascale L-R. de m'avoir supporté, Bruno S. le cycliste et heureux père de triplés, Xavier L. que je souhaite heureux avec son fils Samuel. Je remercie également mes collègues thésards Pierre P. et Simon P. pour nos discussions du matin, qui m'ont beaucoup apporté. Bon courage !

Du côté d'Inria, je vais commencer par remercier mes collègues des équipes STORM et TADaaM (feu Runtime) pour ces moments très enrichissants auprès d'eux : P.A.W, Marie-Christine C., Nathalie F., Jérôme C-O., Guillaume M., Brice G. J'ai également une pensée pour les anciens de l'équipe, qui ont continué à faire vivre l'IRC depuis leur départ, j'ai nommé Sylvain H. monsieur Haskell, Cyril R. grand coureur ~~(de jupons)~~ et grammar nazi accompli, François T. qui se plaît bien à Chicago, et Paul-Antoine A. qui vogue vers de nouveaux projets, que j'espère couronnés de succès.

Ma pensée suivante va vers mes anciens collègues de promotion de Licence et Master qui ont eux aussi démarré ou terminé une thèse, et avec qui j'ai pu partager mes problèmes et mes doutes : Benjamin L., Antoine R. et Ian M. Merci et bon courage les potos.

Ensuite viennent les collègues de l'openspace, avec une pensée particulière pour les fous du carré autoproclamé V.I.P : Farouk M., heureux papa depuis peu et terreur des défenseurs au baby ; Nicolas D. a.k.a No Pain No Gain, grand ambassadeur de la salle de fitness du Leclerc Talence ; Cyril B. recordman du monde des vannes et contrepèteries en série et de la mauvaise foi au baby ; Christopher H. monsieur j'ai-une-référence-pour-tout, et unique responsable de toutes les épigraphes Kaamelott que vous verrez dans ce manuscrit (d'ailleurs, challenge citations Kaamelott : check) et Pierre H., le plus sage de nous tous mais qui a toujours le mot pour rire. Je remercie également Adèle V., toujours discrète mais tellement sympathique, merci de m'avoir supporté avec mes blagues foireuses ; Terry C., monsieur Gentoo et un puits de connaissance et d'inventivité incroyable : continues comme ça, tu iras loin, très loin j'en suis sûr ; Samuel P., détenteur du Râteau d'Or ; Cédric L., heureux papa également et détenteur du record de gamelles du goal au babyfoot ; Hugo B. et Hugo T., nos collègues du CEA que nous n'avons que trop peu vu, et les petits nouveaux Corentin S. grand fan de randonnée le week-end et Clément F., la relève pour perpétuer l'héritage de Kaamelott dans l'openspace.

Je souhaite aussi remercier les collègues de l'équipe HiePACS qui m'ont accompagné durant cette thèse, et en particulier Florent P. qui aimerait plus jouer aux JV, Aurélien F. et Grégoire P. qui ont toujours le mot pour rire, François R. et sa roulette légendaire, et Cyrille P. et Matias H. pour les pauses café du matin. Enfin, je souhaite remercier ma super-secrétaire Sylvie E., qui m'a accompagné pendant ces trois ans dans toutes mes démarches administratives avec tellement d'empathie et de cœur. Merci pour tout, et bon courage pour ton prochain poste !

Passons désormais à une partie de remerciements un peu particulière, à savoir mes amis de jeu. Étant un véritable passionné de jeux vidéos et en particulier de MMORPG, j'ai développé au cours des années (parfois plus d'une dizaine) des liens d'amitié avec pas mal de personnes que je tiens à remercier ici, car ils ont très largement contribué à la réussite de cette thèse à mon sens. Il y en a tellement que je m'excuse d'avance des oublis éventuels (et je ne doute pas que ces derniers se rappelleront à mon bon souvenir !). Je commence par ceux que j'ai rencontrés *IRL* et qui sont devenus des amis avec le temps, Corentin "Ryushin" et sa collection incroyable de consoles et objets liés à la culture geek ; Emeline "Mlyn" et sa passion pour tout ce qui se rapproche de près ou de loin à une peluche, du Carbuncle de FF au Khezu de Monster Hunter ; Louis "Dreiz" le plus gros g33k que j'ai jamais vu ; Victor "Bloodz" qui pense que je prend des douches avec des randonneurs ; Sylvain "Jhora" le premier thésard que j'ai rencontré (et félicitations pour ton mariage bro' !) ; Luigi "Daiixcore" qui m'a toujours fait délirer avec sa voix bizarre. Ensuite, pêle-mêle, Zehyre a.k.a "Félicitations jeune héros !", Zara à Paris dans une coloc', Rayn a.k.a Reinval notre postier de l'impossible, Aldianx a.k.a Sire Aldilenoux et sa petite fille baptisée Shiva depuis qu'elle est née le soir du down du boss éponyme sur FFXIV, Iruhin qui miracule dès qu'il le peut, Angel a.k.a Drek le farmeur fou, Khyden a.k.a frère de Mlyn dont je me demande toujours comment il fait pour arriver à jouer avec autant de boxon à côté de lui. Tous ces gens, et plein d'autres, ont fait de notre petite communauté Nightmare ce qu'elle est aujourd'hui, héritage d'une certaine guilde de WoW Vanilla appelée "Nightmare before Chaos" que j'ai rejointe il y a maintenant 13 ans de cela. Merci à tous d'être encore là pour continuer à faire vivre cet esprit, dans le délire et la bonne humeur.

Ces remerciements ne seraient pas complets sans parler de ma famille. J'ai une pensée émue pour ma mère, qui nous a quittés juste avant que je ne commence cette thèse, et qui lui est dédicacée. Je ne remercierais jamais assez mon père, qui m'a toujours soutenu pendant les moments difficiles, ainsi que les déménagements multiples de ces dernières années. Je pense aussi à mes deux frères, David et Loïc, qui ont eu leur lot de difficultés aussi et j'en profite pour remercier leurs compagnes respectives, Catherine et Anaïs, de les soutenir au quotidien. Bien sûr, petit clin d'œil à mon neveu Liam, qui commence à s'intéresser à l'informatique, ce qui me rend très heureux !

Enfin, je terminerais ces remerciements par une pensée particulière pour Audrey, qui m'a aidé au quotidien à supporter la pression et les attentes de chacun durant ces trois années mouvementées. Merci d'être à mes côtés et de m'avoir toujours soutenu, d'être une oreille attentive et réconfortante, et beaucoup d'autres choses encore.

Table des matières

Table des matières xi

Table des figures xv

Liste des tableaux xix

Introduction 1
La simulation 3D et le calcul haute performance 1
Objectifs et contributions de cette thèse 2
Organisation du document 3

1 Contexte et motivation 5
Résumé du chapitre 5
1.1 La simulation en Furtivité Électromagnétique 6
1.1.1 Introduction au phénomène étudié 6
1.1.2 Besoins en simulation 3D pour la Furtivité Électromagnétique 11
1.1.3 Paralléliser la résolution du système $A\ X = B$ 13
1.2 Délégation du calcul à un support d'exécution 18
1.2.1 Principes d'un support d'exécution 18
1.2.2 Les supports d'exécution à base de tâches 19
1.2.3 Objectifs et choix d'étude 27
1.3 StarPU : un support d'exécution à base de tâches pour architectures hétérogènes 28
1.3.1 Principes de fonctionnement 28
1.3.2 StarPU-MPI : calcul distribué avec StarPU 31
1.4 Entre apports et surcoûts pour le passage à l'échelle 36
1.4.1 Pour le modèle de programmation STF 37
1.4.2 Pour la consommation de mémoire 37
1.4.3 Pour la distribution du travail aux unités de calcul . . . 38

2 Passage à l'échelle du support d'exécution StarPU 41
Résumé du chapitre 42
2.1 Contexte expérimental 43

2.2 Gérer la soumission séquentielle des tâches en distribué 43
2.2.1 Surcoûts de la réplication du graphe de tâches 44
2.2.2 Contribution : retirer les tâches non pertinentes 45
2.2.3 Résultats 46
2.2.4 Discussion 47
2.3 Inconvénients des appels système pour l'allocation de mémoire . 48
2.3.1 Comportement des allocations de mémoire 48
2.3.2 Contribution : utilisation d'un cache d'allocation mémoire 52
2.3.3 Discussion 53
2.4 Contrôler le flot de soumission des tâches 53
2.4.1 Rationalisation de l'utilisation des ressources 54
2.4.2 Contribution : maîtriser le flot de soumission de tâches . 54
2.4.3 Contribution : politiques de contrôle du flot de soumission de tâches 55
2.4.4 Résultats 57
2.4.5 Discussion 60
2.5 Évaluation générale du passage à l'échelle 62
2.5.1 Contexte expérimental 62
2.5.2 Résultats et comparaison avec l'état de l'art 63
2.5.3 Discussion 65
2.6 Intégration dans la chaîne de simulation logicielle 3D 66
Synthèse 67

3 Adapter le contrôle de l'encombrement mémoire aux données de taille évolutive 69
Résumé du chapitre 70
3.1 Compression de matrices issues d'équations intégrales 71
3.1.1 Matrices de rang faible 71
3.1.2 Solveur linéaire avec compression 72
3.2 Contribution : implémentation d'un solveur pour matrices compressées avec StarPU 78
3.2.1 Une représentation StarPU des matrices compressées . . 78
3.2.2 Intégration des noyaux de calcul pour matrices compressées dans Chameleon 79
3.3 Contribution : gestion des données de taille variable avec StarPU 82
3.3.1 Estimer l'évolution de la taille des données 82
3.3.2 Mise en œuvre dans le solveur pour matrices compressées 83
3.4 Précision des estimateurs et performances 83
3.4.1 Cas du solveur pour matrices compressées 84
3.5 Discussion 86

4 Combiner facilement différentes heuristiques d'ordonnancement 89
Résumé du chapitre 89
4.1 Limites des ordonnanceurs monolithiques 90
4.2 Contribution : les ordonnanceurs modulaires 91
4.2.1 Principes des ordonnanceurs modulaires 92
4.2.2 Progression des tâches dans l'ordonnanceur 93
4.2.3 Des composants d'ordonnancement 94
4.2.4 Interface des composants : demandes actives et passives . 94
4.3 Résultats 97
4.3.1 Dimensionnement des réservoirs 97
4.3.2 Comportement des ordonnanceurs 99
4.3.3 Performances 101
4.4 Discussion 101

Conclusion et perspectives 105
Contributions 105
La programmation par tâches, vers l'exascale et au-delà 107

Bibliographie 111

Publications 119

Table des figures

1.1 Objet de surface S et de normale sortante $\overrightarrow{n}$ soumis au champ électromagnétique incident $(\overrightarrow{E^{inc}}, \overrightarrow{H^{inc}})$, et au champ électromagnétique diffracté $(\overrightarrow{E}, \overrightarrow{H})$. 7
1.2 Paramètres de la direction d'incidence $\overrightarrow{k}$ 11
1.3 Isocourants (A/m) sur l'amande conductrice du cas test académique NASA Almond pour une onde plane de fréquence 500MHz. 11
1.4 Découpage en blocs de la factorisation de Cholesky et algorithme associé. 14
1.5 Exemple de factorisation de Cholesky par blocs découpée en panneaux entre 3 processus. 15
1.6 Découpage en tuiles de la factorisation de Cholesky et algorithme associé. 16
1.7 Place du support d'exécution dans la pile logicielle 17
1.8 Algorithme de la factorisation de Cholesky en parallélisme *fork-join* et graphe de tâches associé. 21
1.9 Fonctionnement d'une tâche avec dépendances de données. La donnée A est accédée en lecture/écriture, et la donnée B en lecture seulement. 22
1.10 Algorithme de la factorisation de Cholesky avec dépendances de données et graphe de tâches associé. 22
1.11 Graphe de tâches exhibant des chaînes indépendantes. Les chemins rouges et bleus indiquent des ordres possibles de soumission séquentielle des tâches. 24
1.12 Exemple de code en langage JDF du noyau TRSM de la factorisation de Cholesky et vue du graphe de tâches dans le modèle à flot de données. 26
1.13 Dans StarPU, les codelets permettent d'instancier des tâches multi-architectures. 29
1.14 Exemple d'ordonnancement de tâches glouton. 31
1.15 Exemple d'ordonnancement de tâches avec l'heuristique HEFT. Ici, c'est le GPU#2 qui est choisi par l'ordonnanceur pour exécuter la tâche. 32

1.16 Deux processus peuvent collaborer sur un même graphe de tâches et se partager le travail. 32
1.17 Distribution bloc-cyclique des données de la matrice A d'une factorisation de Cholesky, et exemple de distribution automatique des tâches inférée par StarPU-MPI entre 4 processus. . . . 35
1.18 Si deux tâches ayant une dépendance entre elles s'exécutent sur des processus différents, une communication MPI est inférée entre eux. De plus, comme la communication en pointillés rouge est redondante avec la noire, il est possible de l'éliminer. . 36

2.1 Exemple d'élagage du graphe de tâches sur le processus MPI 0 pour une factorisation de Cholesky d'une matrice 4x4 blocs sur 4 processus. Les tâches élaguées sont grisées. 45
2.2 Impact de l'élagage du graphe de tâches sur les temps de soumission et d'exécution en fonction du nombre de nœuds, à quantité de calcul par nœud constant. Les axes X et Y sont en échelle logarithmique. 46
2.3 Extrapolation de la Figure 2.2 jusqu'à 10 millions de nœuds de calcul. 47
2.4 Positionnement dans l'espace d'adressage d'un processus Linux des zones mémoire allouées par les appels système *mmap* et *sbrk*. 49
2.5 Pic d'empreinte mémoire (haut) et performances (bas) avec chaque stratégie d'allocation. 50
2.6 Fragmentation du tas avec *sbrk*. 51
2.7 Fragmentation du tas avec le cache d'allocation. 52
2.8 Performances (haut) et empreinte mémoire (bas) pour divers seuils de blocage de la soumission de tâches sur 144 nœuds de calcul hybrides (1152 CPUs et 288 GPUs). Le seuil de déblocage est fixé à 80% du seuil de blocage. 58
2.9 Empreinte mémoire du processus le plus chargé en mémoire sans (haut) et avec (bas) contrôle mémoire. Les seuils de blocage et de déblocage avec contrôle mémoire sont définis respectivement à 28 et 24 Go. 59
2.10 Performances des factorisations de Cholesky de ScaLAPACK, DPLASMA et Chameleon sur 144 nœuds de calcul (1152 cœurs CPU et 288 GPUs au total). L'axe des Y est en échelle logarithmique. 64

3.1 Une tuile A_{ij} de rang faible admet une représentation approchée $U_{ij}V_{ij}^t$. 71
3.2 Exemple de matrice tuilée compressible. Les nombres correspondent au rang des tuiles compressées. 73

3.3 Partitionnement d'une amande du cas test académique NASA Almond de 929 280 ddls avec une taille de domaine de 3 000 ddls. On observe bien la structure de type pavage sur la face de l'amande. 74
3.4 Représentation de la distance entre le domaine i et le domaine j, normalisée par rapport à la taille du domaine : $d = \frac{distance(i,j)}{taille\ du\ domaine}$ dans le cas du partitionnement de l'amande présenté FIGURE 3.3. 75
3.5 Comparaison des temps de calcul et de l'occupation mémoire des solveurs dense et compressé pour le cas du lanceur CNES à 1 650 875 inconnues du colloque ISAE. 77
3.6 Méthodes de l'interface de données StarPU que nous avons redéfinies pour l'interface des matrices compressibles. 79
3.7 Prototype d'une fonction de codelet StarPU. 79
3.8 Temps d'exécution du solveur pour matrices compressées de Chameleon et du solveur développé par le CEA en fonction du nombre de cœurs de calcul pour un cas de 750 000 inconnues. . 81
3.9 Encombrement mémoire du processus le plus chargé en mémoire durant l'exécution d'un cas de 450 000 inconnues sur 200 cœurs de calcul 82
3.10 Encombrement mémoire du processus le plus chargé en mémoire durant l'exécution d'un cas de 450 000 inconnues sur 25 nœuds de calcul sans (haut) et avec contrôle *partiel* de l'encombrement mémoire (bas). 85
3.11 Temps de calcul consommé sur les nœuds, représenté en heures CPU, en fonction du nombre de nœuds impliqués dans le calcul pour un cas de 450 000 inconnues. 86

4.1 Un exemple d'ordonnanceur modulaire. 92
4.2 Ordonnanceur modulaire de la FIGURE 4.1 avec les pompes. Les traits pointillés délimitent les zones d'ordonnancement de l'ordonnanceur modulaire. 93
4.3 Interface d'un composant d'ordonnancement. 95
4.4 Transformations des demandes *passives* en demandes *actives* : exemple du `Can_Push` 96
4.5 Ordonnanceur modulaire *modular-heft* 97
4.6 Influence du dimensionnement des réservoirs de plus bas niveau de l'ordonnanceur *modular-heft* sur les performances de notre application pour une matrice de 48 000 inconnues. 98
4.7 État des files de tâches des ordonnanceurs *modular-heft* (gauche) et *dmdas* (droite) durant l'exécution. 99
4.8 Performances des ordonnanceurs modular-heft et dmdas. 100

4.9 Un exemple d'ordonnanceur modulaire qui couple plusieurs heuristiques d'ordonnancement : *hierarchical-heft* 102

Liste des tableaux

2.1 Architectures des machines utilisées. 43
2.2 Compilateurs et bibliothèques utilisées. 43
2.3 Configurations utilisées pour les factorisations de Cholesky. . . . 44
2.4 Réglage des paramètres pour chaque bibliothèque d'algèbre linéaire dense et pour chaque configuration de machine. La taille de bloc représente la taille de tuile pour Chameleon et DPLASMA, et la largeur du panneau pour ScaLAPACK. 63
2.5 Réglages des implémentations pour chaque support d'exécution. 63

Introduction

"C'est pas les idées qui vous manquent, c'est la conviction de devoir les réaliser !"

– Léodagan, *Kaamelott Livre II, L'Alliance*

La simulation 3D et le calcul haute performance

PRÉDIRE les phénomènes physiques avec des simulations informatiques est une nécessité dans le monde d'aujourd'hui, dans des domaines aussi variés que la météorologie, la biomédecine ou l'astrophysique par exemple. Pour écrire un code informatique capable de simuler des phénomènes aussi complexes, plusieurs experts de différentes disciplines doivent travailler de concert : les physiciens élaborent des modèles physico-numériques de ces phénomènes ; les numériciens implémentent ces modèles dans un code informatique ; enfin, les experts en calcul haute performance optimisent le code informatique pour qu'il puisse être calculé en parallèle sur les supercalculateurs modernes. Certaines grandes simulations, en particulier les simulations 3D, nécessitent tellement de calculs qu'il est nécessaire d'utiliser efficacement les supercalculateurs pour que le temps de la simulation soit raisonnable pour l'utilisateur : pour exemple, la plus « grande » simulation que nous étudions dans cette thèse dans le cadre d'un code de Furtivité Électromagnétique du CEA prendrait 306 000 heures si elle était exécutée sur un seul cœur de calcul.

Pour répondre à ce besoin grandissant en puissance de calcul, les supercalculateurs modernes sont désormais composés de plusieurs milliers de machines reliées entre elles par un réseau rapide, et ces machines ont des architectures matérielles très hétérogènes. Par exemple, certaines peuvent posséder différents types de ressources de calcul, comme des processeurs généralistes CPU et des accélérateurs de calcul GPU. Pour pouvoir exploiter au mieux ces machines, la plupart des chercheurs et des industriels choisissent de spécialiser les optimisations de leurs codes de calcul par rapport à l'architecture des machines du supercalculateur qu'ils utilisent, car ces optimisations nécessitent souvent un grand nombre d'heures de travail d'experts en calcul haute performance pour obtenir de bonnes performances. Bien que cette spécialisation permette d'exploiter pleinement les ressources de calcul, un changement dans le modèle physico-numérique ou dans l'architecture des machines peut nécessiter de re-développer une grande partie de ces optimisations de performance. Pour résoudre ce problème, une solution consiste à déléguer certaines optimisations

non portables du calcul à un support d'exécution, qui va automatiser leur gestion et donc assurer à l'application une portabilité de ses performances, grâce à l'utilisation d'un modèle de programmation parallèle capable d'offrir une abstraction de la machine au programmeur.

Objectifs et contributions de cette thèse

Bien que les supports d'exécution offrent des avantages indéniables pour la portabilité des performances et facilitent le développement de codes de calcul parallèle grâce à l'abstraction de la machine qu'ils offrent, toute utilisation de code supplémentaire dans un programme peut induire un surcoût, qui peut ne pas être négligeable lorsque la taille de la simulation et la quantité de ressources de calcul augmente. Cette thèse se propose d'étudier le passage à l'échelle du support d'exécution à base de tâches StarPU, développé à Inria Bordeaux Sud-Ouest, dans le cadre de son intégration dans un code de simulation du CEA qui traite des cas 3D de plusieurs millions d'inconnues. Ce support d'exécution propose à l'application de représenter son calcul sous forme de tâches ayant des dépendances entre elles, puis optimise automatiquement la distribution du calcul ainsi que les transferts de données entre les différentes ressources des machines. Nous avons identifié dans cette thèse trois axes à explorer pour permettre à StarPU de passer à l'échelle :

- **Le calcul distribué.** Pour que l'utilisateur de StarPU puisse utiliser des supercalculateurs composés de plusieurs centaines de nœuds de calcul sans avoir à modifier grandement son code de calcul, le support du distribué de StarPU, nommé StarPU-MPI, suppose la réplication de la soumission des tâches sur tous les processus afin que chacun d'entre eux détermine automatiquement et de manière décentralisée, à partir de la distribution des données, la distribution du calcul et les communications entre processus à effectuer. Cette réplication de la soumission des tâches a un coût, tant en performances qu'en mémoire, et nos contributions ont pour but de minimiser ce coût.
- **La gestion de la mémoire.** Une méthode commune d'optimisation des performances d'une application consiste à allouer la mémoire nécessaire au calcul le plus tôt possible afin de pouvoir recouvrir les transferts de données par du calcul. Il peut alors arriver que cette allocation anticipée de mémoire provoque des débordements de mémoire, qui sont alors difficiles à anticiper et à éviter pour le programmeur. Nous montrons que le modèle à flot de tâches séquentiel qu'implémente StarPU permet de garantir que toute tâche acceptée en soumission peut s'exécuter sans déborder des limites de la mémoire, offrant ainsi à l'utilisateur un excellent compromis entre l'allocation anticipée de mémoire et les performances de l'application, tout en garantissant l'absence de débordements

de mémoire. Nous montrons que cette contribution permet également de contrôler l'encombrement en mémoire d'applications dont la taille des données évolue durant l'exécution.

– **L'ordonnancement de tâches.** L'ordonnancement des tâches consiste à décider, pour chaque tâche, quelle ressource de calcul va l'exécuter et à quel moment. Les décisions d'ordonnancement influencent donc de manière critique les performances de l'application. Le problème est que trouver un « bon » ordonnancement a également un coût en calcul qu'il est important de ne pas négliger, en particulier lors du passage à l'échelle où des millions de tâches doivent être ordonnancées. Nous proposons dans cette thèse une organisation structurée des ordonnanceurs de tâches de StarPU afin de faciliter le développement d'heuristiques d'ordonnancement et de minimiser le surcoût lié à la combinaison de plusieurs d'entre elles au sein d'un même ordonnanceur.

Les contributions de cette thèse ont permis de faire passer avec succès le solveur linéaire dense porté sur le support d'exécution StarPU de la bibliothèque Chameleon jusqu'à 144 nœuds de calcul hybrides du supercalculateur TERA-100 du CEA-DAM, soit 1 152 cœurs et 288 GPUs, et jusqu'à 3 000 cœurs de calcul sans accélérateurs. De plus, les performances obtenues sont comparables à l'état de l'art du domaine ainsi qu'au solveur développé par le CEA pour ses besoins, ce qui soutient l'idée qu'un tel support d'exécution peut être utilisé pour permettre le passage à grande échelle de simulations complexes. Ces résultats sont disponibles dans [80]. StarPU a également été utilisé avec succès dans cette thèse pour développer, optimiser et intégrer au code de simulation du CEA un solveur pour matrices compressées efficace jusqu'à 3 000 cœurs de calcul, dont les résultats ont été publiés dans [77]. Toutes les contributions de cette thèse sont disponibles dans les dépôts open-source de StarPU [3] et de Chameleon [1].

Organisation du document

Le chapitre 1 présente un état de l'art de la programmation par tâches et des support exécutifs associés, et motive leur utilisation pour permettre la portabilité des performances des applications. Le chapitre 2 présente nos contributions au passage à l'échelle en nombre de nœuds de calcul de StarPU. Le chapitre 3 montre comment StarPU peut exploiter une estimation de l'évolution dynamique de la taille des données des applications afin de contrôler leur encombrement de mémoire. Le chapitre 4 présente une formalisation des ordonnanceurs de tâches facilitant les combinaisons d'heuristiques d'ordonnancement. Enfin, nous présentons une conclusion générale des travaux de cette thèse et nous discutons de perspectives pour l'avenir de la programmation à base de tâches, à l'exascale et au-delà.

Chapitre 1

Contexte et motivation

"Moi, j'ai appris à lire, et ben je souhaite ça à personne !"
– Léodagan, *Kaamelott Livre III, L'assemblée des rois* $2^{\text{ème}}$ *partie*

Résumé du chapitre **5**
1.1 La simulation en Furtivité Électromagnétique . . **6**
1.1.1 Introduction au phénomène étudié 6
1.1.2 Besoins en simulation 3D pour la Furtivité Électromagnétique 11
1.1.3 Paralléliser la résolution du système $A\ X = B$ . . . 13
1.2 Délégation du calcul à un support d'exécution . . **18**
1.2.1 Principes d'un support d'exécution 18
1.2.2 Les supports d'exécution à base de tâches 19
1.2.3 Objectifs et choix d'étude 27
1.3 StarPU : un support d'exécution à base de tâches pour architectures hétérogènes **28**
1.3.1 Principes de fonctionnement 28
1.3.2 StarPU-MPI : calcul distribué avec StarPU 31
1.4 Entre apports et surcoûts pour le passage à l'échelle **36**
1.4.1 Pour le modèle de programmation STF 37
1.4.2 Pour la consommation de mémoire 37
1.4.3 Pour la distribution du travail aux unités de calcul . 38

Résumé du chapitre

Dans ce chapitre, nous expliquons en quoi les besoins grandissants en simulation 3D motivent les programmes à déléguer leurs calculs à une couche logicielle spécialisée pour pouvoir passer à l'échelle sur les supercalculateurs. Pour ce faire, nous commençons par présenter notre cas

d'étude, un code de simulation en Furtivité Électromagnétique du CEA, et les raisons qui nous poussent à nous intéresser à son passage à l'échelle. Nous présentons ensuite les techniques de parallélisation du calcul les plus répandues et quelles sont leurs limites, puis nous motivons l'intérêt de déléguer certains aspects du calcul à des supports d'exécution. À travers la présentation de différents modèles de programmation parallèle connus, comme MPI ou OpenMP, nous montrons que la programmation par tâches est un bon candidat pour répondre au problème de la portabilité des performances des programmes parallèles. Nous présentons ensuite le support d'exécution à base de tâches StarPU développé à Inria Bordeaux Sud-Ouest que nous nous proposons d'étudier. Enfin, nous discutons du compromis entre apports et surcoûts pour le passage à l'échelle de l'utilisation de ce type de support d'exécution.

1.1 La simulation en Furtivité Électromagnétique

Cette section décrit rapidement de la modélisation physico-numérique utilisée par le CEA pour le domaine de la Furtivité Électromagnétique, qui aboutit à un système linéaire complexe plein. Nous discutons des besoins en simulations 3D et de la nécessité de paralléliser et d'optimiser la résolution de ce système linéaire. Nous présentons les techniques informatiques de parallélisation de ce solveur les plus répandues, et nous en montrons les limites.

1.1.1 Introduction au phénomène étudié

Nous souhaitons calculer la réponse d'un objet complexe à un éclairement par une onde plane harmonique. Afin de simplifier le propos, nous supposons dans cette section que l'objet est parfaitement conducteur.

Modèle physique

Dans le milieu extérieur à l'objet, le champ électromagnétique diffracté $(\overrightarrow{E}, \overrightarrow{H})$ par un objet vérifie :

– les équations de Maxwell :

$$\begin{cases} rot\overrightarrow{E} + i\omega\mu_0\overrightarrow{H} &= \overrightarrow{0} \\ rot\overrightarrow{H} - i\omega\varepsilon_0\overrightarrow{E} &= \overrightarrow{0} \end{cases} \tag{1.1}$$

– une condition à la surface de l'objet prenant en compte le caractère conducteur de celui-ci :

$$\overrightarrow{n} \wedge \overrightarrow{E} = -\overrightarrow{n} \wedge \overrightarrow{E^{inc}} \tag{1.2}$$

– une condition de radiation à l'infini, car le milieu extérieur est non borné :

$$\lim_{|\overrightarrow{r}|\to\infty} |\overrightarrow{r}| \left| \overrightarrow{E} + \eta_0 \frac{\overrightarrow{r}}{|\overrightarrow{r}|} \wedge \overrightarrow{H} \right| = 0 \tag{1.3}$$

Nous notons ε_0, μ_0, η_0 la permittivité, la perméabilité et l'impédance du milieu extérieur. La pulsation ω dépend de la fréquence f de l'onde électromagnétique, en suivant la relation $\omega = 2\pi f$, $\overrightarrow{n}$ étant la normale sortante à l'objet et $(\overrightarrow{E^{inc}}, \overrightarrow{H^{inc}})$ le champ électromagnétique incident.

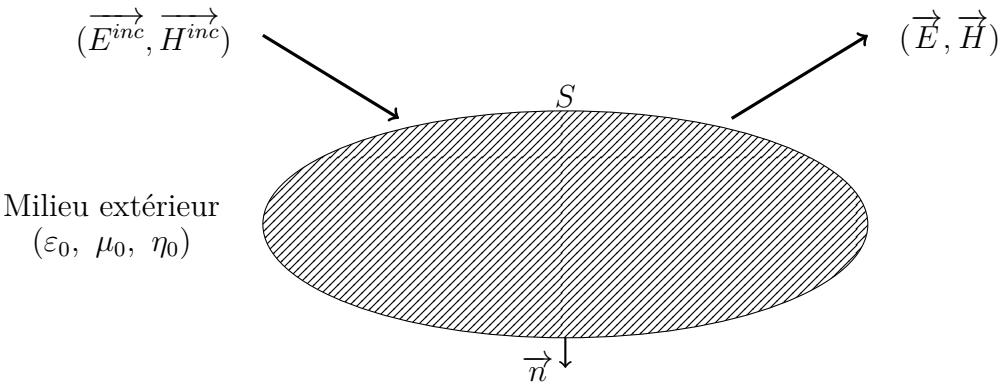


FIGURE 1.1 – Objet de surface S et de normale sortante $\overrightarrow{n}$ soumis au champ électromagnétique incident $(\overrightarrow{E^{inc}}, \overrightarrow{H^{inc}})$, et au champ électromagnétique diffracté $(\overrightarrow{E}, \overrightarrow{H})$.

Formulation intégrale du problème

Le principe de base d'une formule de représentation intégrale est de déduire de la simple connaissance des champs à la surface S de l'objet le champ en tous les points du domaine extérieur.

On note $\overrightarrow{J}$ et $\overrightarrow{M}$ les traces tangentielles des champs électriques et magnétiques totaux sur cette surface, appelés aussi courants électriques et magnétiques :

$$\begin{cases} \overrightarrow{J} &= \overrightarrow{n} \wedge (\overrightarrow{H^{inc}} + \overrightarrow{H}) \\ \overrightarrow{M} &= (\overrightarrow{E^{inc}} + \overrightarrow{E}) \wedge \overrightarrow{n} \end{cases} \tag{1.4}$$

Dans le cas d'un matériau conducteur, $\overrightarrow{M}$ est nul. Le problème se ramène alors à la détermination du courant $\overrightarrow{J}$. Le théorème de représentation intégrale

permet de calculer les champs à partir du courant $\overrightarrow{J}$ comme suit, pour tout x n'appartenant pas à la surface S :

$$\begin{cases} \overrightarrow{E}(x) & = -i\omega\mu_0 \int_S G(x-y)\overrightarrow{J}(y)dS(y) + \frac{1}{i\omega\varepsilon_0} grad_x \int_S G(x-y) div_S \overrightarrow{J}(y)dS(y) \\ \overrightarrow{H}(x) & = rot_x \int_S G(x-y)\overrightarrow{J}(y)dS(y) \end{cases} \tag{1.5}$$

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

$$G(z) = \frac{e^{-ik|z|}}{4\pi|z|} \tag{1.6}$$

On a noté div_S la divergence surfacique, une définition de cet opérateur peut être trouvée dans [30].

Pour déterminer le courant $\overrightarrow{J}$, on écrit une équation sur la surface S en passant à la limite sur une des deux équations de (1.5). Dans la littérature, on parle d'équation EFIE (*Electric Field Integral Equation*) lorsqu'on utilise la représentation en champ électrique :

$$\begin{aligned} -\overrightarrow{n} \wedge \overrightarrow{E^{inc}}(x) = \overrightarrow{n} \wedge \Big(& -i\omega\mu_0 \int_S G(x-y)\overrightarrow{J}(y)dS(y) \\ & + \frac{1}{i\omega\varepsilon_0} grad_x \int_S G(x-y) div_S \overrightarrow{J}(y)dS(y) \Big) \end{aligned} \tag{1.7}$$

et de MFIE (*Magnetic Field Integral Equation*) si on choisit celle en champ magnétique $\overrightarrow{H}$:

$$\overrightarrow{n} \wedge \overrightarrow{H^{inc}}(x) = \frac{1}{2}\overrightarrow{J} - \overrightarrow{n} \wedge rot_x \int_S G(x-y)\overrightarrow{J}(y)dS(y) \tag{1.8}$$

Grâce à la formulation EFIE (ou MFIE), on est passé d'un problème posé sur un domaine 3D non borné à un problème posé sur un domaine 2D borné (la surface de l'objet). La principale difficulté de calcul réside dans la singularité du noyau de Green (1.6) lorsque $z = 0$ (en $x = y$).

La méthode permet le traitement de géométries très diverses : conducteurs parfaits, assemblages de conducteurs et diélectriques, mais elle est limitée aux seuls matériaux homogènes et isotropes.

Discrétisation par la méthode des éléments finis

Une méthode classique d'éléments finis de frontières (BEM [1]) est appliquée ensuite pour discrétiser ces équations intégrales. On commence par écrire une

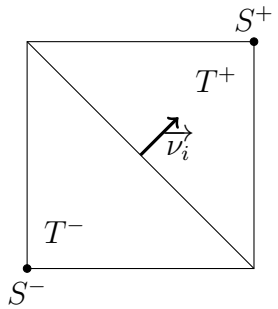
1. *Boundary Element Method*

formulation variationnelle de l'équation choisie. Il suffit pour ce faire de multiplier l'équation par un courant test et d'intégrer le tout sur la surface de l'objet. Ce qui donne par exemple dans le cas de l'équation EFIE (1.7) :

$$\int_S \overrightarrow{E^{inc}}.\overrightarrow{J'(x)}dS(x) = \int_S \int_S G(x-y)\left[i\omega\mu_0 \overrightarrow{J}(y).\overrightarrow{J'}(x) + \frac{1}{i\omega\varepsilon_0} div_S \overrightarrow{J}(y) div_S \overrightarrow{J'}(x)\right] dS(y)dS(x) \tag{1.9}$$

Puis on considère une triangularisation $\mathcal{T}$ de la surface S. La taille des arêtes de chaque triangle doit être de l'ordre de $\lambda/10$, avec $\lambda = \frac{c}{f}$ la longueur d'onde (c étant la célérité de la lumière dans le vide), afin de discrétiser correctement le problème. On emploie l'élément fini de Raviart-Thomas de degré 1 pour approcher les inconnues. À chaque arête A_i, on associe :

- une fonction de base $\overrightarrow{\phi}_i$, dont le support est la réunion de deux triangles adjacents par cette arête :



$$\overrightarrow{\phi_i(x)} = \begin{cases} \frac{1}{2mes(T^+)}\overrightarrow{S^+x} & , x \in T^+ \\ -\frac{1}{2mes(T^-)}\overrightarrow{S^-x} & , x \in T^- \\ \overrightarrow{0} & , \text{sinon} \end{cases} \tag{1.10}$$

- un degré de liberté (que l'on notera ddl par la suite) J_i qui n'est autre que le flux du courant à travers l'arête A_i :

$$J_i = \int_{A_i} \overrightarrow{J}.\overrightarrow{\nu_i}dl \text{ , avec } \overrightarrow{\nu_i} \text{ la normale à l'arête } A_i \tag{1.11}$$

Le courant électrique est ensuite décomposé sur cette base :

$$\overrightarrow{J}(x) = \sum_{i=1}^{n} J_i \overrightarrow{\phi_i}(x) \text{ , } \forall\ x \in S \tag{1.12}$$

où n est le nombre total d'arêtes dans le maillage. On injecte cette décomposition du courant dans la formulation variationnelle (1.9) pour obtenir au final un système linéaire complexe plein de la forme $A\ X = B$.

Les termes de la matrice A (appelée « matrice d'impédance ») sont donnés dans le cas de la formulation EFIE par :

$$A_{ij} = \int_{\mathcal{T}} \int_{\mathcal{T}} G(x-y)\left[i\omega\mu_0 \overrightarrow{\phi_i}(x).\overrightarrow{\phi_j}(y) + \frac{1}{i\omega\varepsilon_0} div_S \overrightarrow{\phi_i}(x) div_S \overrightarrow{\phi_j}(y)\right] dS(y)dS(x) \tag{1.13}$$

Ils dépendent uniquement de la pulsation ω de l'onde incidente. On observe que la matrice d'impédance A ainsi construite est symétrique dans le cas de

l'EFIE. Dans le code du CEA, la majorité des formulations intégrales retenues produisent des matrices d'impédances symétriques.

Le second membre B est donné par la formule qui suit, il est dépendant du champ incident (pulsation et direction d'incidence).

$$B_i = \int_{\mathcal{T}} \overrightarrow{E^{inc}}(x).\overrightarrow{\phi_i}(x)dS(x) \tag{1.14}$$

Enfin, le vecteur inconnu X est le vecteur des ddls. À partir de ce vecteur, il est alors possible de mesurer la réponse de l'objet à l'onde incidente. Cette réponse peut être représentée sous deux formes différentes :

- la Surface Équivalente Radar (SER) ;
- les courants électriques $\overrightarrow{J}$ et magnétiques $\overrightarrow{M}$ circulant à la surface de l'objet (1.4).

La Surface Équivalente Radar

La SER est une mesure de l'énergie diffractée dans une direction donnée. Afin de ne pas dépendre de la distance au point d'observation mais seulement de la direction, la SER est normalisée par un facteur $4\pi r^2$ pour compenser la variation en $1/r$ de l'intensité du champ diffracté. Elle est homogène à une surface, et est souvent exprimée en dBm^2 :

$$\sigma = 10\log_{10}\left(\lim_{|r|\to\infty} 4\pi r^2 \left(\frac{|\overrightarrow{E}|}{|\overrightarrow{E^{inc}}|}\right)^2\right) \tag{1.15}$$

où $\overrightarrow{E^{inc}}$ est le champ électrique incident et $\overrightarrow{E}$ le champ électrique diffracté.

Lorsque l'éclairement et l'observation sont situés dans la même direction, la SER obtenue est appelée SER monostatique. Elle est appelée SER bistatique dans le cas contraire.

La direction d'incidence $\overrightarrow{k}$, le champ magnétique $\overrightarrow{H}$, et le champ électrique $\overrightarrow{E}$ forme un trièdre direct. On parle de polarisation H quand le champ magnétique est situé dans le plan (Ox, Oy) et de polarisation V quand le champ électrique est dans ce plan.

Les isocourants

Un exemple de courants parcourant l'amande conductrice du cas test académique NASA Almond [37] lorsqu'elle est exposée à une onde électromagnétique plane de fréquence 500MHz est présenté sur la Figure 1.3. Ces courants sont obtenus en reconstruisant la trace tangentielle des courants à l'aide de la valeur en chaque ddl.

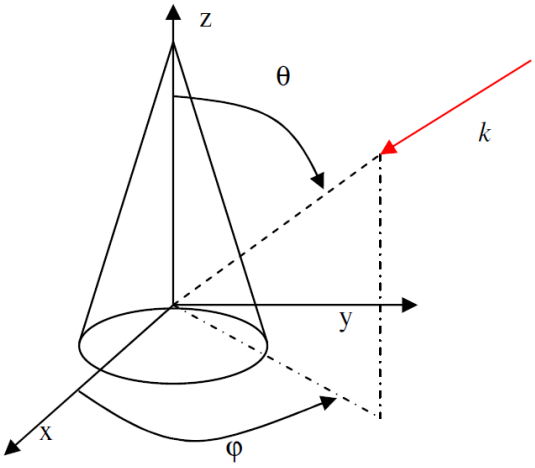


FIGURE 1.2 – Paramètres de la direction d'incidence $\overrightarrow{k}$

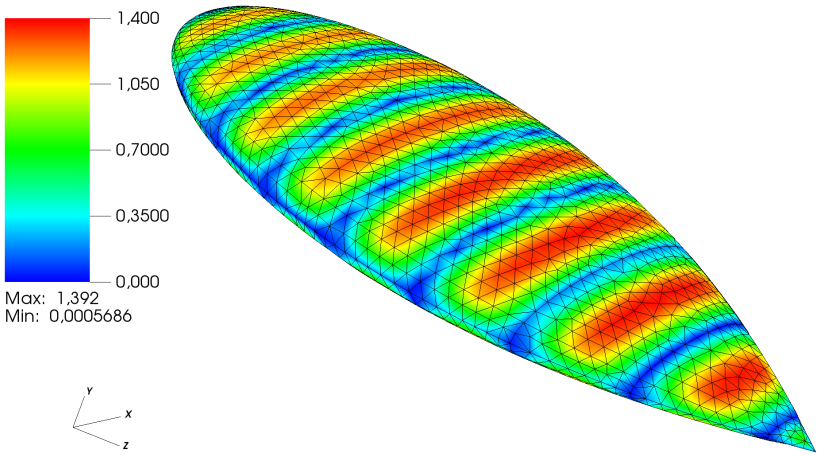


FIGURE 1.3 – Isocourants (A/m) sur l'amande conductrice du cas test académique NASA Almond pour une onde plane de fréquence 500MHz.

1.1.2 Besoins en simulation 3D pour la Furtivité Électromagnétique

La chaîne de simulation logicielle 3D

Dans le cadre des grands calculs 3D considérés par le code du CEA, les composantes du système linéaire $A\ X = B$ à résoudre ont les propriétés suivantes :

- les matrices d'impédance A sont complexes, non hermitiennes, symétriques ou non ;
- le vecteur de ddls X peut comporter plusieurs millions d'inconnues et nécessite une grande précision de calcul ;

- le second membre B étant dépendant de la direction d'incidence de l'onde, une simulation pour plusieurs directions d'incidence (que l'on nomme balayage angulaire) résultera en la présence de multiples solutions X et de multiples seconds membres B : X et B deviennent des matrices non carrées.

Pour simuler cette physique complexe, le CEA a développé depuis le début des années 90 une chaîne de simulation logicielle qui repose sur la formulation « tout Équations Intégrales » présentée en section précédente. Ses principaux avantages en regard des méthodes concurrentes sont sa précision et l'utilisation de maillages surfaciques bien plus faciles à réaliser que les maillages volumiques.

L'inconvénient principal est la nécessité de résoudre des systèmes linéaires denses dont la taille croît de façon quadratique avec la fréquence de calcul. L'emploi de solveurs itératifs n'est pas approprié pour ces problèmes de grande taille et pour la précision recherchée, car d'une part le système linéaire dense obtenu est généralement mal conditionné et la construction d'un pré-conditionneur adapté reste encore un problème ouvert et, d'autre part le nombre de résolutions du système (nombre de seconds membres) engendre un coût prohibitif pour les méthodes itératives lors de balayages angulaires. C'est la raison pour laquelle l'utilisation d'un solveur direct massivement parallèle a toujours été privilégiée, et que nous souhaitons optimiser son exécution.

Passage à l'échelle pour des machines hétérogènes

Pour satisfaire ces besoins de simulation à très grande échelle, il faut pouvoir tirer pleinement parti des supercalculateurs modernes, qui sont de plus en plus complexes à bien des égards. Ils sont généralement composés de plusieurs dizaines de milliers de cœurs de calcul, répartis entre plusieurs centaines de machines qui sont reliées entre elles par un réseau rapide. Ces cœurs de calcul sont regroupés au sein d'architectures matérielles très hétérogènes entre elles, chacun ayant pour but de répondre à un besoin spécifique. Les cœurs de calcul peuvent soit être évolués et groupés en petit nombre afin de traiter efficacement des travaux majoritairement séquentiels, comme dans les processeurs généralistes de type CPU. À l'inverse, si le travail à effectuer est massivement parallèle, les cœurs de calcul peuvent être simplifiés et groupés en très grand nombre pour absorber ce parallélisme, comme dans les accélérateurs de calcul de type GPU et les processeurs manycore comme l'Intel KNL [65].

Pour nos expériences, nous avons pu utiliser le supercalculateur TERA-100 du CEA-DAM [4]. Il est composé de :

- une partition avec 138 368 cœurs de calcul répartis sur 4 324 nœuds (32 cœurs par nœud) ;
- une partition avec 1 584 cœurs de calcul et 396 accélérateurs de de type GPU, répartis sur 198 nœuds (8 cœurs et 2 GPUs par nœud).

Pour pouvoir tirer pleinement parti de cette machine, un programme doit pouvoir exploiter toutes les ressources de calcul disponibles simultanément et durant toute l'exécution. Il doit pouvoir nourrir plusieurs unités de calcul en même temps, transférer les données du calcul entre les différentes mémoires de la machine, et communiquer des données entre les machines pour respecter la sémantique du programme pour le calcul distribué. Les propriétés souhaitées d'un tel programme parallèle sont :

- une utilisation optimisée de multiples ressources de calcul hétérogènes ;
- une adaptabilité des performances aux différentes configurations matérielles utilisées : ce qu'on nomme la portabilité des performances ;
- des performances qui augmentent proportionnellement à la quantité de données à traiter et/ou à la quantité de ressources de calcul utilisée : ce que l'on nomme le passage à l'échelle du programme.

Pour ce faire, le programme doit :

- exprimer les parties du calcul qui peuvent s'exécuter en parallèle ;
- répartir équitablement le travail sur toutes les unités de calcul disponibles (qui peuvent être hétérogènes) ;
- spécifier les parties du programme qui doivent s'échanger des données et effectuer les transferts de données.

Il est également nécessaire de rappeler que les programmes parallèles doivent intégrer les évolutions des méthodes numériques. Pour les problématiques de la Furtivité Électromagnétique, on peut penser aux méthodes FMM [2] [44] ou de compression de matrices [19] par exemple.

Il est donc également souhaitable que l'écriture d'un programme parallèle puisse être formalisée dans un modèle de programmation expressif. Nous présentons à travers un historique du calcul parallèle pour l'algèbre linéaire dense comment il est possible d'arriver à cet objectif en délégant les optimisations de calcul des programmes parallèles à une couche logicielle tierce, que l'on nomme support d'exécution.

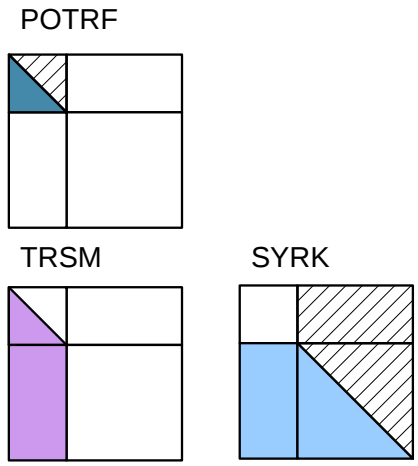
1.1.3 Paralléliser la résolution du système $A\ X = B$

Un programme séquentiel

La première bibliothèque de solveurs linéaires d'algèbre linéaire dense, LINPACK [38], a été écrite en Fortran dans les années 1970 pour profiter des supercalculateurs de l'époque. Elle implémente tous les solveurs d'algèbre linéaire dense usuels, tels que les factorisations LU, QR et Cholesky, les calculs de valeurs singulières, etc. Dans LINPACK, le calcul est purement séquentiel et se fait élément par élément.

2. *Fast Multipole Method*

Avec l'apparition des architectures à mémoire partagée dans les années 1980 [34], une nouvelle bibliothèque de solveurs linéaires conçue pour tirer pleinement parti de ces nouvelles machines a supplanté LINPACK : c'est la bibliothèque LAPACK [3] [12], reconnue comme la bibliothèque de référence pour l'algèbre linéaire dense pour les machines à mémoire partagée. Elle propose une version optimisée des solveurs de LINPACK, où le calcul est découpé en blocs, sur lesquels des routines de calcul du standard BLAS [4] [53] sont appelées. Les



Algorithme 1 : Algorithme de la factorisation de Cholesky par blocs

```
for (k = 0 ; k < N ; k++) do
  POTRF (A[k][k]) ;
  for (m = k+1 ; m < N ; m++) do
    TRSM (A[k][k], A[m][k]) ;
  end for
  for (n = k+1 ; n < N ; n++) do
    SYRK (A[n][k], A[n][n]) ;
  end for
end for
```

FIGURE 1.4 – Découpage en blocs de la factorisation de Cholesky et algorithme associé.

BLAS sont un ensemble de routines conçues pour optimiser les calculs basiques de l'algèbre linéaire dense (comme la multiplication de matrices par exemple) pour les architectures en mémoire partagée, en tirant parti des capacités de vectorisation des processeurs, en favorisant le parallélisme d'instructions, et en optimisant la réutilisation des données pour minimiser le nombre de défauts de cache.

L'avènement des machines multi-cœurs dans les années 2000 a provoqué une nouvelle évolution des méthodes de calcul, car le parallélisme d'instructions et la vectorisation ne suffisent pas pour exploiter ce type de machine : il est nécessaire que le programme exprime les calculs qui peuvent être effectués en parallèle sur différents cœurs.

Du programme séquentiel au programme parallèle

Pour tirer parti de ces multiples unités de calcul en même temps, il est donc nécessaire que le programme spécifie les parties du calcul qui peuvent s'exécuter en parallèle. Dans le cas de l'algèbre linéaire dense, une première proposition

3. *Linear Algebra PACKage*
4. *Basic Linear Algebra Subprograms*

a été de conserver le découpage du calcul en blocs proposé par LAPACK, mais de partitionner les données des matrices en panneaux qui sont répartis de manière statique entre les processus qui vont alors collaborer ensemble, comme présenté sur la FIGURE 1.5. Cette méthode est implémentée dans la bibliothèque ScaLAPACK [29], qui reste à l'écriture de ces lignes l'une des bibliothèques les plus utilisées sur les supercalculateurs pour résoudre des problèmes d'algèbre linéaire dense.

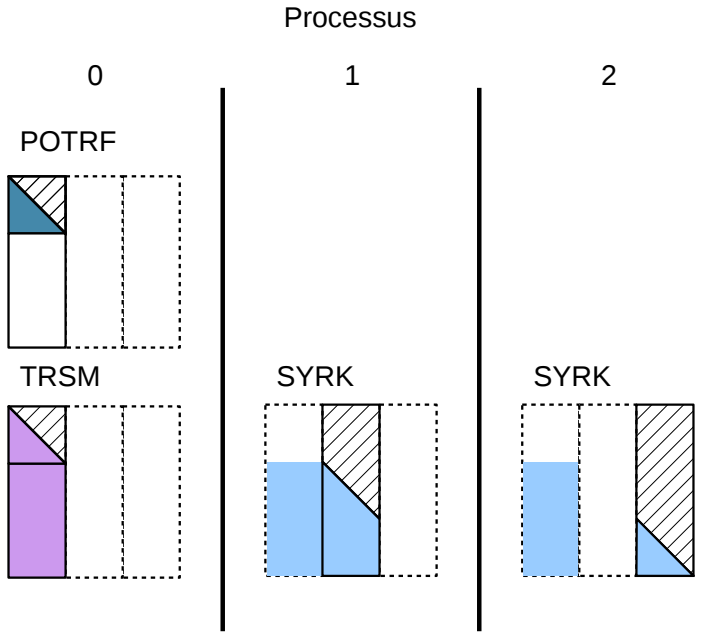
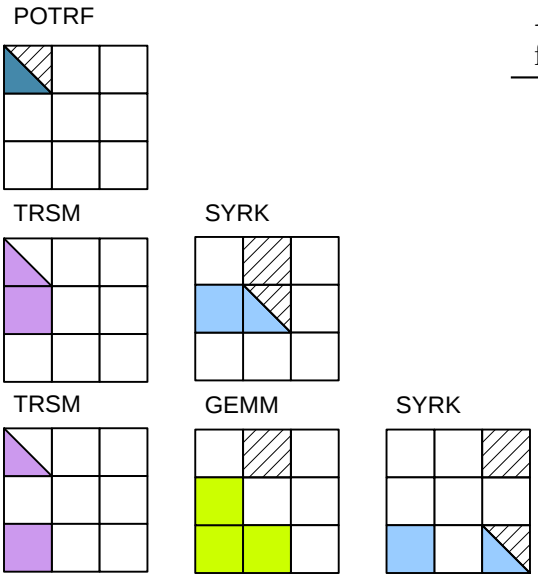


FIGURE 1.5 – Exemple de factorisation de Cholesky par blocs découpée en panneaux entre 3 processus.

Bien que cette solution permette d'exposer du parallélisme de calcul, elle n'est pas suffisante pour obtenir de bonnes performances pour ces machines multi-processeurs. En effet, le découpage par blocs n'expose pas assez de parallélisme pour alimenter toutes les unités de calcul : par exemple, tous les processus chargés de la mise à jour de la matrice doivent attendre que le processus chargé du calcul du panneau termine pour pouvoir commencer à calculer. Trouver une distribution statique du travail permettant une utilisation efficace des unités de calcul n'est alors pas évident, car il faut pouvoir quantifier et équilibrer correctement la charge de calcul, la quantité de données à stocker et le volume de données à communiquer pour chaque processus. De plus, la distribution statique du travail entre les unités de calcul nécessite de spécifier cette distribution pour toutes les configurations possibles de machines. En pratique, des compromis sur les performances sont admis pour limiter le nombre de configurations à supporter.



Algorithme 2 : Algorithme de la factorisation de Cholesky tuilée

```
for (k = 0 ; k < N ; k++) do
  POTRF (A[k][k]) ;
  for (m = k+1 ; m < N ; m++) do
    TRSM (A[k][k], A[m][k]) ;
  end for
  for (n = k+1 ; n < N ; n++) do
    SYRK (A[n][k], A[n][n]) ;
    for (m = n+1 ; m < N ; m++)
    do
      GEMM (A[m][k], A[n][k],
      A[m][n]) ;
    end for
  end for
end for
```

FIGURE 1.6 – Découpage en tuiles de la factorisation de Cholesky et algorithme associé.

Pour que l'algorithme puisse exhiber plus de parallélisme, la communauté a proposé de partitionner davantage les données en décomposant chaque bloc de données en une multitude de tuiles qui peuvent alors être calculées en parallèle [46]. Pour la factorisation de Cholesky par exemple, la FIGURE 1.6 montre que ce découpage en tuiles permet de calculer en parallèle la mise à jour du panneau de la matrice avec deux appels à la routine TRSM, là où ce calcul était séquentiel dans ScaLAPACK. Des bibliothèques qui exploitent ce parallélisme nouveau ont alors vu le jour, comme PLASMA [27] ou FLAME [45] pour les processeurs généralistes, mais aussi CUBLAS [58] pour pouvoir profiter des accélérateurs de type GPU pour les calculs d'algèbre linéaire dense.

Les solutions proposées par ces bibliothèques ont toutefois des limites. En effet, chaque bibliothèque a sa propre distribution du travail sur les ressources de calcul et est conçue pour fonctionner seules. Cela les rend incompatibles entre elles et empêche donc l'utilisation simultanée de toutes les ressources de calcul. Un effort a cependant été fait dans ce sens avec le développement de solveurs hybrides capables d'utiliser simultanément les CPUs et les GPUs, comme dans la bibliothèque MAGMA [11].

Une autre problématique reste la distribution statique du travail entre les ressources de calcul. En effet, bien qu'une telle distribution permette de contrôler et d'optimiser facilement et finement le déroulement de l'exécution

du programme, elle empêche le programme de s'adapter dynamiquement au déséquilibre de charge dû à la présence sur les machines de ressources de calcul de plus en plus hétérogènes (CPUs, GPUs, manycores) et hétérogènes entre elles (plusieurs générations différentes de matériels peuvent cohabiter au sein d'un même supercalculateur).

Pour résumer, ces bibliothèques, de par leur abstraction complète du calcul, ne permettent pas au programmeur d'applications scientifiques de contrôler directement et dynamiquement l'exécution du programme sur les ressources de calcul pour optimiser ses performances.

Vers un programme parallèle efficace et portable

Pour que le programmeur puisse contrôler la distribution du travail pour tirer pleinement parti de toutes les ressources de calcul quelle que soit la machine qui exécute le programme, il doit pouvoir :

- contrôler finement la distribution du travail entre les différentes ressources de calcul ;
- piloter les transferts de données pour recouvrir les temps de communication par du calcul.

Pour que le programmeur n'ait pas à contrôler directement la machine par des appels système, ce qui reste difficile même pour un programmeur expert, il est nécessaire de l'aider à optimiser l'exécution parallèle de son programme.

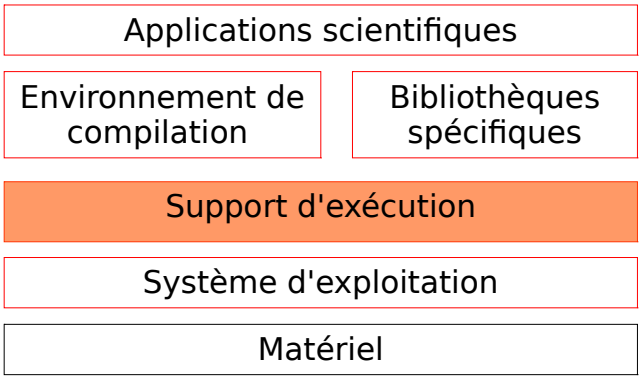


FIGURE 1.7 – Place du support d'exécution dans la pile logicielle

Pour ce faire, une vision abstraite de la machine est proposée au programmeur à travers un modèle de programmation parallèle lui permettant de déléguer l'optimisation des aspects complexes du calcul parallèle à une couche logicielle, située entre l'application et le système d'exploitation (cf. FIGURE 1.7), nommée support d'exécution.

1.2 Délégation du calcul à un support d'exécution

1.2.1 Principes d'un support d'exécution

Pour remplir les objectifs d'abstraction de la machine et d'optimisation des peformances, les supports d'exécution proposent un modèle de programmation parallèle plus abstrait que les appels système. Ainsi, à partir de la formalisation proposée par le modèle de programmation, le support d'exécution peut implémenter un modèle d'exécution parallèle capable d'optimiser automatiquement les transferts de données entre la mémoire principale et un accélérateur de calcul, la distribution du travail sur les unités de calcul, ou encore les communications entre machines distantes. Le support d'exécution peut également proposer des points d'entrée pour que le programmeur puisse, s'il le souhaite, contrôler plus finement l'exécution du programme.

La plupart des modèles de programmation parallèle utilisent déjà un support d'exécution pour automatiser certaines optimisations de performance. On peut citer MPI [31], dont le support d'exécution gère les transferts de données sur le réseau ; CUDA [57], dont le support d'exécution gère l'exécution du calcul sur les cœurs des GPUs (plusieurs milliers sur les cartes récentes) et offre une abstraction des transferts mémoire entre les GPUs et la mémoire principale, ou encore OpenMP [60], dont le support d'exécution gère le parallélisme de threads sur les CPUs. Un avantage de ces modèles de programmation est qu'ils peuvent être utilisés de concert au sein d'un même programme car chaque support d'exécution gère une sous-partie différente des ressources des machines. Ainsi, la combinaison du modèle de programmation MPI pour le calcul distribué avec un modèle de programmation en mémoire partagée pour le calcul local à une machine (couramment appelée MPI+X, X pouvant être des threads POSIX, OpenMP ou même MPI 3.0 [47]) reste la méthode de parallélisation de code la plus utilisée dans l'industrie et la recherche. Elle permet en effet un contrôle très fin des optimisations de performance d'un programme, comme le recouvrement des mouvements de données par du calcul ou le placement des threads par rapport au placement des données, grâce à une proximité toujours forte des modèles de programmation avec le matériel. Ces modèles de programmation sont de plus standardisés, ce qui facilite grandement la portabilité des codes de calcul parallèle sur les architectures modernes.

De l'abstraction pour la portabilité des performances

La programmation directe des unités de calcul n'a cependant pas que des avantages. Pour la programmation par threads par exemple, le programmeur doit détecter et éviter tous les problèmes dus à l'accès simultané des threads aux mêmes ressources partagées en introduisant des synchronisations.

On pense notamment à l'utilisation de mutexes pour protéger contre des écritures concurrentes de la même zone mémoire. Le risque ici est que le programmeur entrave excessivement le parallélisme son programme pour assurer que la sémantique soit respectée, résultant en une perte de performances. Des modèles de programmation parallèle comme OpenMP permettent d'aider le programmeur en lui proposant des constructeurs de calcul parallèle, qui garantissent le respect de la sémantique du programme au prix d'une restriction sur le type de parallélisme qu'il est possible d'exprimer. De même, obtenir de bonnes performances des accélérateurs de type GPU n'est pas évident. Bien que ces architectures disposent d'une grande puissance de calcul, la vitesse des transferts de données entre leur mémoire embarquée et la mémoire principale est conditionnée par la bande passante du port PCIe des machines. Il est ainsi plus difficile d'obtenir un bon recouvrement des communications par du calcul sur ces architectures.

Pour toutes ces raisons, le programmeur doit avoir une expertise en calcul haute performance pour obtenir une application performante en utilisant ce type de programmation. De surcroît, comme les optimisations de performance sont dépendantes des matériels sous-jacents, il peut être nécessaire de retravailler tout ou partie des optimisations pour l'adapter à de nouveaux matériels.

Afin de limiter cet effort d'adaptation à chaque évolution des architectures de calcul, la portabilité des performances devient de plus en plus un besoin essentiel lors du développement d'une application parallèle.

Un support d'exécution doit donc d'une part proposer un modèle de programmation suffisamment abstrait pour faciliter l'expression du parallélisme et le dissocier de l'architecture sous-jacente des machines, et d'autre part son modèle d'exécution doit pouvoir optimiser la parallélisation du programme, et s'adapter facilement aux nouvelles ressources de calcul. Dans ce manuscrit, nous nous proposons d'étudier une famille de modèles de programmation parallèle : la programmation par tâches. Nous présentons en quoi elle répond à cette problématique d'abstraction et de portabilité des performances.

1.2.2 Les supports d'exécution à base de tâches

Décrire un programme avec des tâches

Une tâche représente un calcul élémentaire qui ne fait aucun effet de bord, soit unité fonctionnelle pure. Les tâches permettent de représenter le travail d'un programme de manière abstraite, ce qui le rend indépendant de la machine sur laquelle il va être exécuté.

Écrire un programme en tâches consiste donc à décrire les unités élémentaires du calcul du programme (les tâches) et les instancier. C'est ensuite le support d'exécution qui aura la charge de diriger l'exécution parallèle des tâches du programme. La programmation par tâches permet donc d'exprimer les élé-

ments de calcul indépendants et qui peuvent être exécutés en parallèle.

Par contre, tous les modèles de programmation par tâches ne permettent pas aux supports d'exécution sous-jacents d'optimiser l'exécution parallèle des tâches de la même manière. Nous nous intéressons plus particulièrement à trois modèles de programmation : par tâches indépendantes, par graphe de tâches et par graphe à flot de données, et nous discutons des avantages et inconvénients de chacun d'entre eux pour motiver notre choix d'étude.

Programmation par tâches indépendantes

Dans ce modèle de programmation, les tâches sont créées de manière totalement indépendantes à partir d'appels à une bibliothèque [21, 61, 54], des directives [59] ou à un constructeur de langage [62]. C'est ensuite au programmeur d'assurer le respect de la sémantique du programme en insérant des points de synchronisation là où c'est nécessaire : pour un programme OpenMP, via une directive `pragma omp taskwait` ; pour un programme C++, via un appel à la méthode `wait` des objets *futures* du standard C++11 [48]. Ce type de parallélisme est nommé *fork-join*, car le calcul parallèle est généré à partir d'un seul point, et les calculs parallèles sont synchronisés sur un même point.

La Figure 1.8 montre comment il est possible d'utiliser ce modèle de programmation pour écrire une factorisation de Cholesky. L'exécution des tâches est présentée sous la forme d'un graphe dirigé acyclique [5].

Ce modèle de programmation a l'avantage d'être simple à implémenter. Il permet aussi de décrire aisément certains types de programmes, comme les programmes récursifs où il est possible d'imbriquer les créations de tâches les unes dans les autres, comme le propose Cilk++ [54] par exemple.

Par contre, pour décrire un programme ayant des dépendances, le programmeur doit introduire des points de synchronisation qui ralentissent la progression du chemin critique du DAG. On appelle le chemin critique d'un DAG la plus longue chaîne de tâches du graphe, qui doivent donc être exécutées les unes après les autres. Le temps de calcul de l'application ne peut donc pas être inférieur au temps de calcul séquentiel cumulé de toutes les tâches de ce chemin, même avec un nombre de processeurs infini. Sur la Figure 1.8 par exemple, nous constatons que le chemin critique passe par des points de synchronisation qui forcent l'exécution des tâches de ce chemin à attendre des tâches qui n'en font pas partie. Le problème ici est donc que le programmeur, qui doit ajouter des barrières afin de gérer la synchronisation des tâches, ajoute également des dépendances entre tâches qui ne correspondent pas à des dépendances effectives de l'algorithme. Comme le modèle de programmation ne permet pas d'expliciter finement les dépendances entre tâches nécessaires pour respecter la sémantique du programme, le programmeur doit surcontraindre le parallélisme

5. Pour faciliter la lecture, on parlera de DAG (*Directed Acyclic Graph*) dans la suite du manuscrit

Algorithme 3 : Factorisation de Cholesky tuilée avec des tâches indépendantes

```
for (k = 0 ; k < N ; k++) do
    task_insert (&POTRF, A[k][k]) ;
    task_wait_for_all () ;
    for (m = k+1 ; m < N ; m++) do
        task_insert (&TRSM, A[k][k],
        A[m][k]) ;
    end for
    task_wait_for_all () ;
    for (n = k+1 ; n < N ; n++) do
        task_insert (&SYRK, A[n][k],
        A[n][n]) ;
        for (m = n+1 ; m < N ; m++) do
            task_insert (&GEMM, A[m][k],
            A[n][k], A[m][n]) ;
        end for
    end for
    task_wait_for_all () ;
end for
```

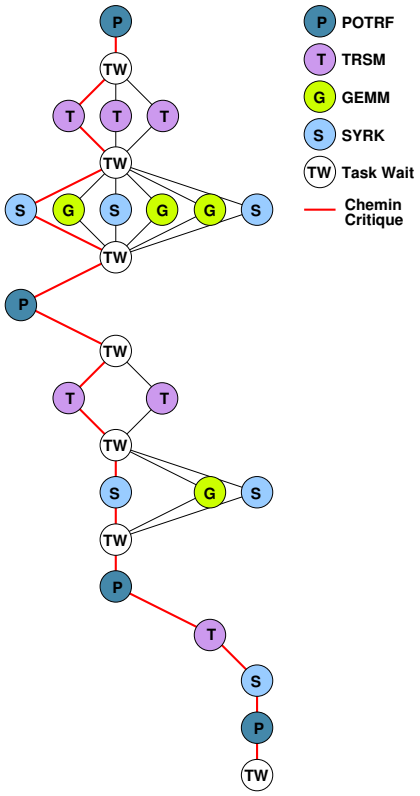


FIGURE 1.8 – Algorithme de la factorisation de Cholesky en parallélisme *fork-join* et graphe de tâches associé.

de son application pour obtenir le résultat souhaité.

Programmation par tâches dépendantes

Dans ce type de modèle, le programme doit spécifier, en plus du corps des tâches, les données accédées par les tâches ainsi que le mode d'accès à ces données : lecture et/ou écriture, comme présenté en FIGURE 1.9. Contrairement au modèle de programmation précédent, où l'insertion des dépendances entre tâches pour garantir la sémantique du programme doit être faite explicitement par le programmeur, il est possible dans ce modèle de construire automatiquement les dépendances entre tâches uniquement depuis les données accédées et le mode d'accès aux données.

Il est possible de déléguer la construction des dépendances entre les tâches au support d'exécution : lors de la soumission d'une tâche, le support d'exécution détecte, pour chaque donnée accédée par la tâche, quelles tâches précédemment soumises y accèdent aussi. Chaque dépendance de donnée avec une tâche précédemment soumise correspond ainsi à une arête le DAG généré par

```
Task (Data A, Data B)
{
    A = A + B;
}
```

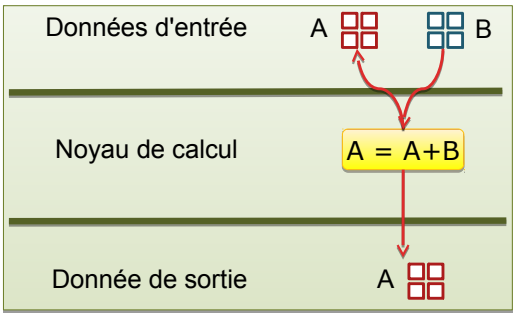


FIGURE 1.9 – Fonctionnement d'une tâche avec dépendances de données. La donnée A est accédée en lecture/écriture, et la donnée B en lecture seulement.

le support d'exécution. Pour notre exemple de la factorisation de Cholesky, la FIGURE 1.10 montre le graphe de tâches ainsi créé.

Algorithme 4 : Factorisation de Cholesky tuilée avec des tâches dépendantes

```
for (k = 0 ; k < N ; k++) do
    task_insert(&POTRF, RW, A[k][k], 0) ;
    for (m = k+1 ; m < N ; m++) do
        task_insert(&TRSM, R, A[k][k], RW, A[m][k], 0) ;
    end for
    for (n = k+1 ; n < N ; n++) do
        task_insert(&SYRK, R, A[n][k], RW, A[n][n], 0) ;
        for (m = n+1 ; m < N ; m++) do
            task_insert(&GEMM, R, A[m][k], R, A[n][k],
                RW, A[m][n], 0) ;
        end for
    end for
end for
task_wait_for_all() ;
```

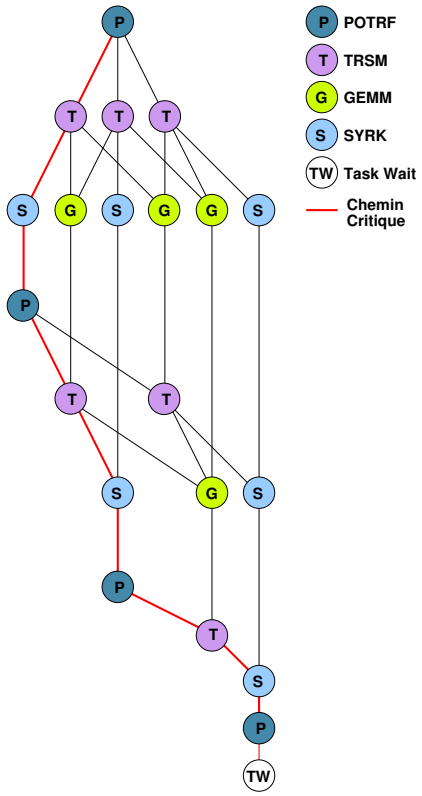


FIGURE 1.10 – Algorithme de la factorisation de Cholesky avec dépendances de données et graphe de tâches associé.

Ce modèle de programmation est dit à flot de tâches séquentiel[6], car les dépendances de données sont calculées lors de la soumission séquentielle des tâches.

Le modèle de programmation STF est implémenté au sein de plusieurs supports d'exécution, comme Quark [72], StarPU [16] ou SuperGlue [69]. Il a également été intégré dans certains modèles de programmation parallèle, notamment OmpSs [39] et OpenMP 4 [60].

Un des avantages de ce modèle par rapport au modèle par tâches indépendantes est la gestion fine des dépendances entre tâches, ce qui permet de décrire des DAG qui exposent davantage le parallélisme des programmes. Par exemple, on voit sur la Figure 1.10 que l'exécution du chemin critique est désormais uniquement liée par les dépendances nécessaires pour préserver la sémantique du programme, ce qui permet au support d'exécution de faire progresser le calcul du chemin critique sans attendre nécessairement l'exécution de toutes les tâches d'un niveau du DAG, comme c'était le cas pour le modèle précédent en Figure 1.8.

Par contre, ce modèle se prête moins à l'expression de certains schémas de calcul. Pour exprimer la récursivité par exemple, comme il est nécessaire de spécifier les données accédées par une tâche, le support d'exécution doit partitionner les données de la tâche parente pour permettre le calcul des dépendances pour les sous-tâches générées. De plus, comme la génération des dépendances entre tâches est faite lors de leur soumission, le support d'exécution peut inférer des dépendances cycliques lorsque des tâches crééent des sous-tâches dont elles doivent attendre l'exécution pour terminer. En effet, si le programme a soumis, *entre* la soumission et l'exécution de la tâche récursive, une tâche qui dépend des mêmes données que la tâche récursive, les sous-tâches créées par cette dernière seront alors dépendantes de la tâche tierce, qui est elle même dépendante de la tâche récursive. Ce cycle va alors provoquer un blocage du programme. Il est possible ce résoudre ce problème en permettant aux tâches d'être préemptibles, mais cela nécessite alors de changer le modèle de programmation, car le graphe de tâches n'est plus acyclique.

Enfin, l'ordre séquentiel de soumission des tâches peut restreindre le parallélisme disponible pour le support d'exécution. Prenons l'exemple du graphe de tâches de la Figure 1.11, repris depuis [35]. Ce graphe de tâches est composé d'un nombre arbitraire de chaînes de tâches pouvant s'exécuter en parallèle. Pour ce graphe, l'ordre de soumission séquentiel des tâches peut grandement influencer le parallélisme disponible pour le support d'exécution. En effet, si l'application soumet les tâches dans l'ordre correspondant au chemin rouge, le support d'exécution ne disposera pas de tâches pouvant s'exécuter en parallèle avant le début de la soumission de la deuxième chaîne de tâches. À l'inverse, si l'application soumet les tâches dans l'ordre du chemin bleu, le parallélisme

6. Pour faciliter la lecture, on préfèrera parler de modèle STF, pour *Sequential Task Flow*.

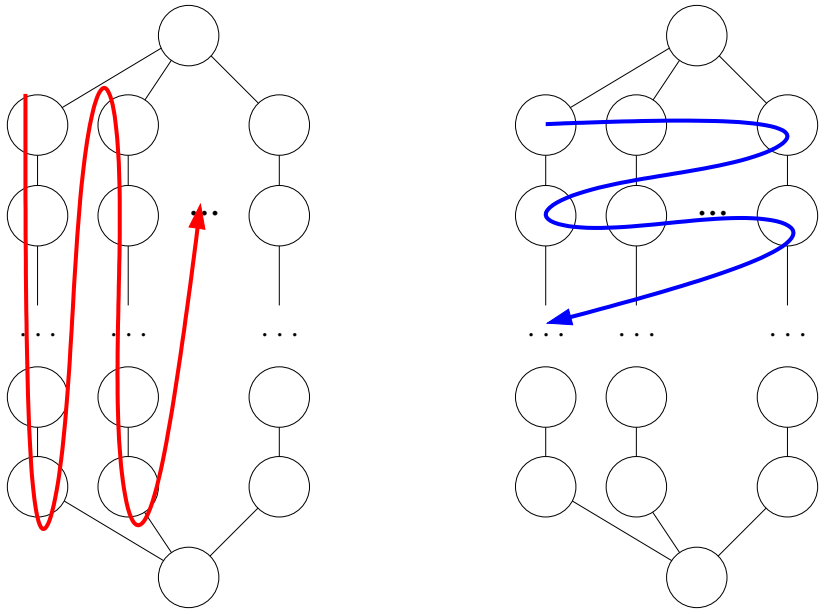


FIGURE 1.11 – Graphe de tâches exhibant des chaînes indépendantes. Les chemins rouges et bleus indiquent des ordres possibles de soumission séquentielle des tâches.

disponible sera correct car chaque chaine fournit du parallélisme.

Pour résumer, les restrictions sur le parallélisme de ce modèle proviennent de la soumission séquentielle du graphe de tâches. Pour lever ces limites, il est nécessaire de pouvoir soumettre les tâches en parallèle.

Des graphes de tâches aux graphes à flot de données

Pour pouvoir soumettre les tâches en parallèle tout en respectant la sémantique du programme, il est nécessaire que le modèle de programmation abstraie également la soumission des tâches, de sorte que le support d'exécution les créée automatiquement lorsque les données de la tâche sont disponibles. Un modèle de programmation qui répond à ce besoin est le modèle à flot de données.

Dans ce modèle, le corps des tâches est décrit par le programme sous la forme d'*acteurs*. Un acteur décrit en plus les *règles* que doit respecter le flot de données entrant dans l'acteur pour activer son exécution (par exemple, attendre d'avoir reçu une quantité de données correspondant à une tuile de matrice), ainsi que la destination du flot de données sortant : il peut soit être transmis à un ou plusieurs autres acteurs, soit être écrit en mémoire. Ainsi, écrire un programme dans ce modèle consiste à décrire le squelette du programme sous forme d'acteurs, puis spécifier où les données de calcul doivent être lues en mémoire au départ, et où elles doivent être écrites à la fin.

L'exécution concrète du programme correspond ainsi au passage du flot de données de départ d'acteur en acteur, jusqu'à ce que les acteurs écrivent le résultat dans la zone mémoire prévue à cet effet par le programme. Si l'on fait un parallèle avec les modèles d'exécution par graphe de tâches, on a ici un modèle d'exécution par graphe d'acteurs dont les arêtes représentent les flots de données entre acteurs. Avec ce modèle, contrairement aux tâches qui doivent être créées séquentiellement, il est possible d'instancier les acteurs dynamiquement et en parallèle dès que toutes leurs opérandes sont disponibles.

Il existe plusieurs supports d'exécution basés sur ce modèle d'exécution à flot de données, comme FastFlow [42], StreaMIT [68] ou Intel CnC [26]. On peut également citer Charm++ [49, 50], qui offre un modèle de programmation à flot de données via la programmation orientée objet : les acteurs sont les objets et la progression du flot de données entre objets se fait par des appels de méthodes entre objets.

Pour représenter un DAG comme un graphe à flot de données, Cosnard et al. présentent dans [33] une méthode permettant de compresser algébriquement les DAG pour les représenter sous une forme paramétrique proche d'une représentation à flot de données. On dit alors que le DAG est représenté en format PTG [7]. Il est ainsi possible de spécifier un modèle de programmation dans lequel les DAG sont décrits en format PTG. De tels DAG peuvent être générés en parallèle car les dépendances entre acteurs sont directement encodées dans les dépendances de données.

Par exemple, le support d'exécution PaRSEC [23] implémente une conversion automatique à la compilation du DAG décrit séquentiellement par le programmeur en un ensemble d'acteurs décrits dans le langage JDF [8] [24].

Un des avantages du modèle à flot de données sur les modèles par graphe de tâches est que seules les tâches *actives* existent. Les tâches futures sont créées dynamiquement depuis les règles de propagation du flot de données décrites par l'application, comme présenté en Figure 1.12. Ce modèle facilite également la réutilisation des données par le support d'exécution, qui peut forcer des acteurs liés par des flots de données à s'exécuter sur des unités de calcul proches. La représentation algébrique du graphe de tâches dans le format PTG permet également l'extraction à la compilation des schémas de communication collectives.

Par contre, la méthode de calcul automatique d'une représentation en format PTG d'un DAG présentée dans [24] se base sur une analyse par *optimisation linéaire en nombre entiers* [9] sur les nids de boucles séquentiels. Dans le cas général, le programmeur doit décrire lui-même son DAG dans le modèle à flot de données, et garantir que la sémantique du programme décrit dans ce modèle est identique au DAG décrit séquentiellement peut ne pas être évident pour le

7. *Parameterized Task Graph*
8. *Job Data Flow*
9. Aussi appelée *Integer Programming*

Algorithme 5 : Noyau TRSM de la factorisation de Cholesky tuilée dans le modèle PTG

```
TRSM(k, m)

// Espace d'exécution
k = 0   .. N-1
m = k+1 .. N-1

// Distribution des tâches
: A[m][k]

// Flots de données et leurs dépendances
READ  A <- A POTRF(k)
RW    C <- (k == 0) ? A[m][k]
        <- (k != 0) ? C GEMM(k-1, m, k)
        -> A SYRK(k, m)
        -> A GEMM(k, m, k+1..m-1)
        -> B GEMM(k, m+1..N-1, m)
        -> A[m][k]

BODY
  trsm( A /* A[k][k] */,
        C /* A[m][k] */ );
END
```

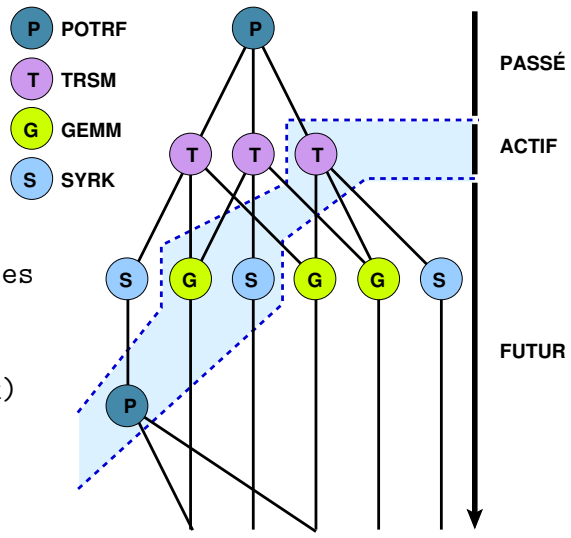


FIGURE 1.12 – Exemple de code en langage JDF du noyau TRSM de la factorisation de Cholesky et vue du graphe de tâches dans le modèle à flot de données.

programmeur. L'utilisation d'un tel modèle de programmation parallèle dans un code séquentiel existant peut donc nécessiter d'importants changements dans le code. Une autre problématique liée à ce modèle est l'absence d'état global de l'exécution, qui limite le contrôle qu'il est possible d'avoir sur l'exécution du programme, notamment pour optimiser l'ordonnancement des tâches ou pour contrôler la consommation de ressources.

De l'algèbre linéaire dense sur support d'exécution à base de tâches

Le modèle de programmation par tâches est un bon candidat pour implémenter les algorithmes tuilés d'algèbre linéaire dense. Le calcul et les données étant déjà découpés et décrits de manière modulaire par l'algorithme, la conversion

en tâches des implémentations est ainsi facilitée.

C'est pourquoi la programmation par tâches et les supports d'exécution sont de plus en plus utilisés dans ce domaine. Le support d'exécution Quark, qui implémente le modèle STF, a été substitué à la distribution statique du travail dans la bibliothèque PLASMA [11]. La bibliothèque Chameleon [8], développée conjointement par Inria Bordeaux Sud-Ouest et l'UTK [10] dans le cadre du projet MORSE [11] [7], est une collection de solveurs d'algèbre linéaire dense sur supports d'exécution STF qui est compatible avec les supports d'exécution StarPU, Quark et PaRSEC. Enfin, la bibliothèque DPLASMA [12] [25], développée par l'UTK, utilise le support d'exécution PaRSEC afin de tirer pleinement parti de la représentation à flot de données des graphes de tâches de l'algèbre linéaire dense.

1.2.3 Objectifs et choix d'étude

Nous souhaitons dans cette thèse étudier l'utilisation d'un support d'exécution à base de tâches pour optimiser le solveur d'algèbre linéaire dense du code de simulation 3D du CEA. Notre choix s'est porté sur le support d'exécution StarPU, qui implémente le modèle STF, pour deux raisons. La première est que son modèle de programmation séquentiel ainsi que son couplage avec la bibliothèque Chameleon facilitent grandement l'intégration de cette solution dans le code de simulation. La seconde est que StarPU a déjà été utilisé avec succès dans de nombreux contextes applicatifs, industriels comme académiques, comme le traitement d'images [55], l'algèbre linéaire creuse [51], ou la simulation d'ondes sismiques [56], ce qui nous convainc de sa robustesse. Pour les besoins du CEA, il faut également que la solution retenue soit suffisamment performante pour accélérer le temps de calcul des grands cas 3D traités par ce code. Bien que StarPU ait été utilisé précédemment avec succès sur une machine, le passage à l'échelle distribué de StarPU n'a été que peu exploré. Ainsi, cette thèse se propose de répondre à ces deux questionnements :

- le modèle STF permet-il de faire passer à grande échelle des grands codes complexes comme celui du CEA, et facilite-t-il leur implémentation ?
- les limites du passage à l'échelle de StarPU sont-elles liées au modèle STF en lui-même comme le suggèrent Yarkhan et al. dans [72], ou à des problématiques d'implémentation qui sont corrigibles ?

10. *Université du Tenessee, Knoxville*
11. *Matrices Over Runtime Systems @ Exascale*
12. *Distributed PLASMA*

1.3 StarPU : un support d'exécution à base de tâches pour architectures hétérogènes

Le support d'exécution StarPU [16], développé à Inria Bordeaux Sud-Ouest, vise à offrir au programmeur une abstraction des machines hétérogènes en lui présentant une vue unifiée des différentes ressources de calcul (CPUs, GPUs, manycores), et en prenant en charge la distribution efficace du travail et le recouvrement des transferts mémoire par du calcul.

Nous présentons dans cette section, à travers le passage de l'algorithme STF du code de la factorisation de Cholesky vers un code StarPU, comment le modèle de programmation offert par StarPU ainsi que les principes de fonctionnement du support d'exécution associé permettent l'exécution efficace de graphes de tâches sur des machines hétérogènes. Nous montrons ensuite comment le modèle de programmation de StarPU a été étendu dans StarPU-MPI pour permettre le calcul distribué du graphe de tâches tout en minimisant les changements nécessaires dans le code applicatif.

1.3.1 Principes de fonctionnement

Les codelets, des tâches multi-architectures

Dans le modèle de programmation de StarPU, une codelet est une structure qui contient toutes les informations nécessaires à la création d'une tâche. Elle contient notamment les différentes implémentations possibles d'une tâche, en fonction de l'architecture visée. Cette description flexible des tâches permet au programmeur de simplement décrire auprès de StarPU les différentes implémentations d'une tâche sur plusieurs architectures, laissant ainsi au support d'exécution la possibilité de choisir dynamiquement quelle implémentation de la tâche il souhaite utiliser.

Créer une tâche StarPU consiste alors à associer une codelet (le code de la tâche) à un ensemble de données (les entrées de la tâche) spécifié par l'utilisateur. Ainsi, le code StarPU équivalent à l'algorithme 4 de la Figure 1.10 est présenté par l'algorithme 6.

On remarque qu'outre le changement de nommage des fonctions `task_insert`, la structure de l'algorithme est exactement similaire à celle de l'algorithme 4. Les deux changements notables par rapport à l'algorithme précédent sont la description des calculs, qui sont désormais des codelets (caractérisés par le suffixe `_cl` dans l'algorithme), et les données accédées, qui sont désormais des *handles* StarPU.

Une vision unifiée de la mémoire

Pour pouvoir gérer automatiquement les mouvements de données entre les différentes mémoires de la machine, StarPU implémente une mémoire virtuelle

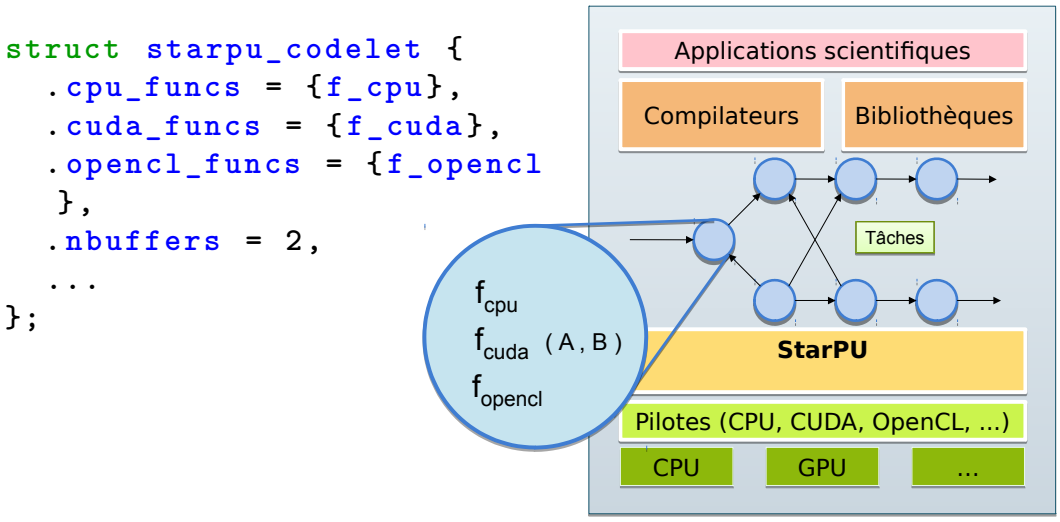


FIGURE 1.13 – Dans StarPU, les codelets permettent d'instancier des tâches multi-architectures.

Algorithme 6 : Factorisation de Cholesky tuilée avec StarPU

```
for (k = 0 ; k < N ; k++) do
  starpu_task_insert(&POTRF_cl, RW, Ahandles[k][k], 0) ;
  for (m = k+1 ; m < N ; m++) do
    starpu_task_insert(&TRSM_cl, R, Ahandles[k][k], RW,
    Ahandles[m][k], 0) ;
  end for
  for (n = k+1 ; n < N ; n++) do
    starpu_task_insert(&SYRK_cl, R, Ahandles[n][k], RW,
    Ahandles[n][n], 0) ;
    for (m = n+1 ; m < N ; m++) do
      starpu_task_insert(&GEMM_cl, R, Ahandles[m][k], R,
      Ahandles[n][k],
        RW, Ahandles[m][n], 0) ;
    end for
  end for
end for
starpu_task_wait_for_all() ;
```

partagée [14], ou VSM [13], afin de masquer au programmeur les transferts de données entre ces mémoires multiples. Le modèle de programmation de StarPU

13. *Virtual Shared Memory.*

permet au programmeur d'enregistrer simplement les données de l'application auprès de cette VSM sous la forme de *handles*, qui rajoute une indirection entre le code applicatif et le pointeur effectif, permettant de déplacer une donnée d'un espace à l'autre. Pour l'exemple de notre factorisation de Cholesky, l'algorithme 7 permet d'enregistrer auprès de la VSM de StarPU toutes les tuiles de notre matrice A.

Algorithme 7 : Enregistrement des données de la matrice A auprès de StarPU.

```
for (m = 0 ; m < N ; m++) do
  for (n = 0 ; n < N ; n++) do
    starpu_matrix_data_register(&Ahandles[m][n], A[m][n], ...) ;
  end for
end for
```

Il suffit ensuite au programmeur de fournir ces handles lors de l'insertion des tâches pour que le support d'exécution s'occupe automatiquement des mouvements de données à effectuer pendant l'exécution.

Une fois les données enregistrées et les tâches insérées, le support d'exécution pousse les tâches pour lesquelles les données d'entrée et de sortie sont disponibles (elles sont dites *prêtes*) vers l'ordonnanceur de tâches qui va décider, pour chaque tâche, quelle unité de calcul va l'exécuter.

Ordonnancement de tâches sur architectures hétérogènes

Le problème de l'ordonnancement de tâches appliqué aux supports d'exécution consiste à trouver la distribution des tâches sur les unités de calcul qui minimise le temps de restitution de l'application [14]. Ce problème étant NP-complet [41, 71], de nombreuses heuristiques ont été élaborées par la communauté pour tenter d'y répondre.

Dans StarPU, chaque unité de calcul est représentée par un Worker StarPU. Un Worker est un thread qui pilote l'exécution des tâches sur l'unité de calcul qui lui est associée en réclamant des tâches à exécuter à l'ordonnanceur et en les faisant exécuter par l'unité de calcul.

Dans le cas simple d'un ordonnanceur gloton, nommé *eager* dans StarPU et présenté en Figure 1.14, l'ordonnanceur contient une simple liste dans laquelle les tâches prêtes sont stockées. Lorsqu'une unité de calcul est libre, son Worker associé récupère une tâche depuis la liste de l'ordonnanceur et l'exécute. Pour répartir équitablement les tâches entre les différentes unités de calcul d'une machine, des heuristiques de type vol de travail [22] peuvent également être utilisées.

14. L'usage est de parler de *makespan* pour désigner ce temps.

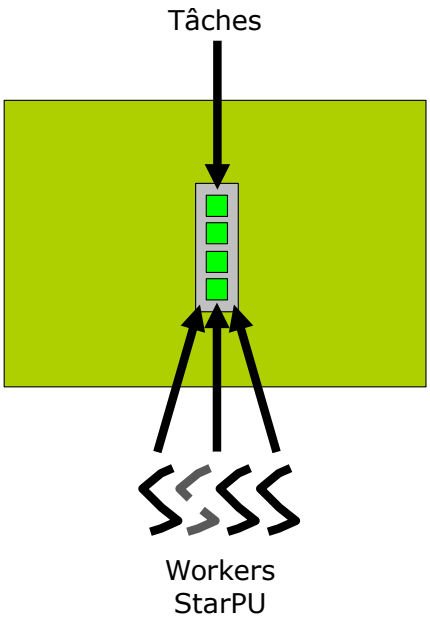


FIGURE 1.14 – Exemple d'ordonnancement de tâches glouton.

Pour des machines disposant d'accélérateurs de calcul par exemple, l'hétérogénéité des unités de calcul complexifie le problème car une même tâche peut s'exécuter plus ou moins vite selon l'unité de calcul choisie. Un ordonnanceur glouton peut alors, par malchance, ne donner à une unité de calcul que des tâches pour lesquelles elle n'est pas bonne, alors qu'une autre unité de calcul de la machine aurait pu l'exécuter bien plus rapidement.

Une méthode pour prendre en compte cette hétérogénéité est d'estimer le coût d'une tâche en fonction de l'unité de calcul qui l'exécute, et de prendre en compte ces coûts dans l'heuristique d'ordonnancement. Par exemple, l'heuristique Heterogeneous Earliest Finish Time (HEFT) [70], présentée en FIGURE 1.15 et implémentée en plusieurs déclinaisons dans StarPU sous les noms *DMDA** [15], utilise des modèles de performance précalculés [15] pour prédire le temps d'exécution de chaque tâche sur chaque unité de calcul. Ainsi, elle peut choisir d'exécuter la tâche sur l'unité de calcul qui minimise la durée de l'application.

1.3.2 StarPU-MPI : calcul distribué avec StarPU

Nous abordons dans cette section le cas du calcul distribué. Nous discutons des solutions existantes pour distribuer le calcul du graphe de tâches entre plusieurs processus, puis nous présentons le support distribué de StarPU, nommé StarPU-MPI.

15. *Deque Model Data Aware*

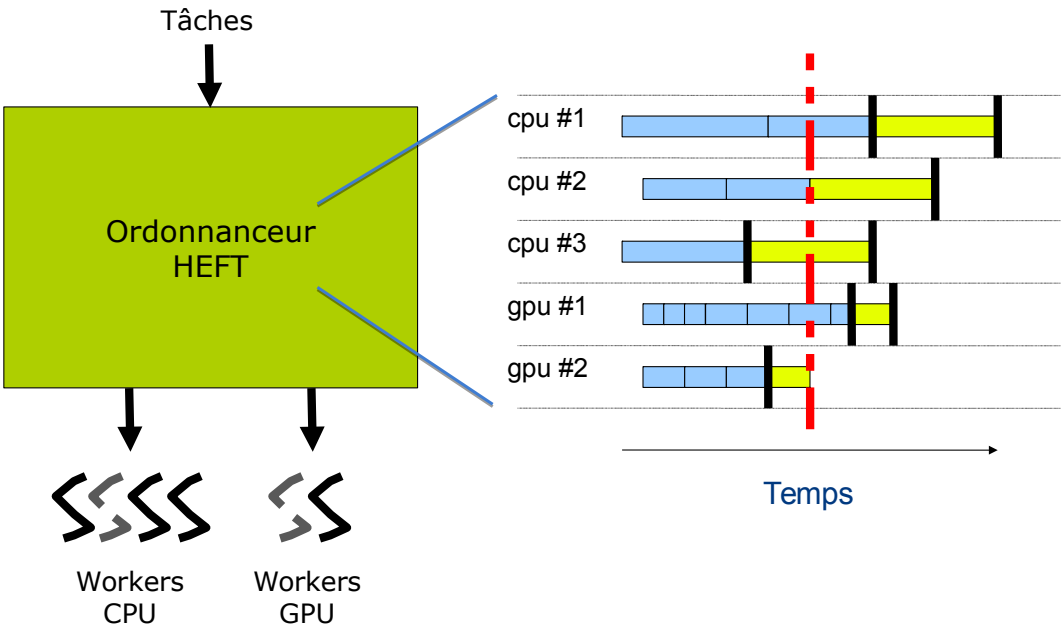


FIGURE 1.15 – Exemple d'ordonnancement de tâches avec l'heuristique HEFT. Ici, c'est le GPU#2 qui est choisi par l'ordonnanceur pour exécuter la tâche.

Graphe de tâches et calcul distribué

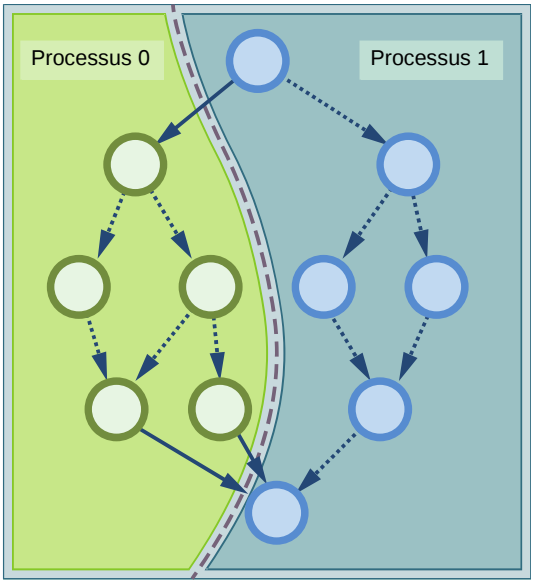


FIGURE 1.16 – Deux processus peuvent collaborer sur un même graphe de tâches et se partager le travail.

Pour exécuter un programme à base de tâches sur une machine distribuée,

il est possible de faire collaborer plusieurs processus sur un même graphe de tâches, comme présenté en Figure 1.16.

StarPU propose une surcouche logicielle, nommée StarPU-MPI [13], qui étend le modèle de programmation de StarPU pour permettre au programmeur de distribuer les tâches de son application entre plusieurs processus. Toutefois, l'objectif de StarPU-MPI est de gérer automatiquement les communications de données nécessaires à l'exécution distribuée du graphe de tâches de manière transparente pour l'application. Pour ce faire, le support d'exécution a besoin de trois informations :

- la distribution des données entre les processus ;
- la distribution du travail entre les processus ;
- les communications à effectuer entre les processus.

Ainsi, l'extension du modèle de programmation STF proposée par StarPU-MPI a été pensée pour minimiser les changements à faire dans le code applicatif pour pouvoir l'exécuter de manière distribuée.

Il existe différents modèles de programmation distribuée d'un graphe de tâches pour spécifier ces informations. Dans le modèle maître-esclave, seul le processus maître soumet le graphe de tâches. Il orchestre ensuite les mouvements de données et l'exécution des tâches sur les esclaves. Le processus maître peut posséder toutes les données du calcul ou les distribuer aux esclaves. Bien que ce modèle soit simple à mettre en place et permette un bon équilibrage de charge entre les processus, comme le processus maître est un point de centralisation, il est également un point de contention, ce qui limite le passage à l'échelle de ce modèle. Il est possible de réduire la contention du maître en adoptant une approche hiérarchique, mais les points de centralisation vont redevenir bloquants lorsque le nombre de processus augmentera. Ce modèle de programmation est notamment implémenté par le support d'exécution OmpSs dans son support du distribué ClusterSs [67].

Un autre modèle de programmation distribuée consiste à répliquer le déroulement du graphe de tâches sur tous les processus. Dans ce modèle, tous les processus soumettent le graphe de tâches. Le support d'exécution est ensuite capable d'inférer toutes les communications de données depuis le déroulement du graphe. En effet, comme la soumission des tâches est séquentielle, toutes les décisions prises par un processus durant la soumission des tâches (comme préparer un transfert de données dès que la tâche sera terminée par exemple) sont déterministes et reproductibles. Si la soumission du graphe de tâches est répliquée sur tous les processus comme dans ce modèle, tous les processus prendront les mêmes décisions à la même étape du déroulement séquentiel du graphe de tâches, sans avoir besoin de se synchroniser entre eux. Grâce à cette propriété, chaque processus peut inférer automatiquement la distribution globale du travail et les communications entre processus de manière autonome, et ce uniquement depuis la distribution des données entre les processus.

Par exemple, une heuristique de distribution du travail par rapport à la distribution de données peut être faite sur le modèle *le travail bouge vers les données*, c'est à dire que le processus qui va exécuter une tâche est celui qui possède le plus de données en écriture de cette tâche. Il est ensuite capable de détecter quels processus possèdent les données de la tâche et demander les données manquantes aux processus concernés. À l'opposé, si le processus n'exécute pas la tâche mais possède au moins une des données impliquées dans son calcul, il va initier l'envoi des données au processus qui doit exécuter la tâche. Ce modèle de programmation distribuée permet donc de déterminer les trois informations nécessaires au calcul distribué de manière totalement décentralisée, ce qui lui donne des propriétés susceptibles de permettre son passage à l'échelle. Ce modèle a été choisi par Quark pour implémenter son support du calcul distribué, nommé Quarkd, et par StarPU-MPI, le support du distribué sur MPI de StarPU.

Réplication du graphe de tâches avec StarPU-MPI

Pour convertir un programme StarPU en programme StarPU-MPI qui peut s'exécuter en distribué, le seul ajout que le programmeur a donc besoin de faire est de fournir à StarPU-MPI la distribution des données entre les processus. Si l'on reprend l'exemple de la factorisation de Cholesky, l'algorithme 8 montre comment décrire une distribution 2D-bloc cyclique d'une matrice A à StarPU-MPI. À partir de la distribution de données, StarPU-MPI va automatiquement inférer la distribution du travail entre les processus selon la politique *le travail se déplace vers les données*, comme le montre le graphe de tâches de la Figure 1.17. De plus, StarPU-MPI va inférer automatiquement sur chaque processus les communications MPI à effectuer en fonction du graphe de tâches, comme présenté en Figure 1.18 : s'il existe une dépendance entre deux tâches s'exécutant sur deux processus différents, StarPU-MPI va automatiquement soumettre des communications non bloquantes entre les processus pour effectuer les transferts de données nécessaires à l'exécution de la tâche.

Élaguer les communications redondantes

Bien que cette méthode d'automatisation des communications entre processus permette de respecter la sémantique du programme distribué, elle peut inférer plusieurs communications d'une même donnée, comme montré sur la Figure 1.18 : dans cet exemple, la donnée produite par la tâche POTRF est communiquée deux fois au processus 1. On souhaiterait donc élaguer une des deux communications (ici, la rouge). Du fait que les communications sont inférées séquentiellement depuis le déroulement du graphe de tâches, StarPU-MPI est ainsi capable de maintenir sur chaque processus un cache contenant les communications de données déjà effectuées ou programmées avec les autres processus.

Comme les modes d'accès aux données sont spécifiés à la soumission des

Algorithme 8 : Spécification d'une distribution de données bloc-cyclique en deux dimensions

```
for (m = 0 ; m < N ; m++) do
  for (n = 0 ; n < N ; n++) do
    starpu_matrix_data_register(
    &Ahandles[m][n], A[m][n], ...) ;

    starpu_data_set_rank(
    Ahandles[m][n],
    m%P+(n%Q)*P) ;
  end for
end for
```

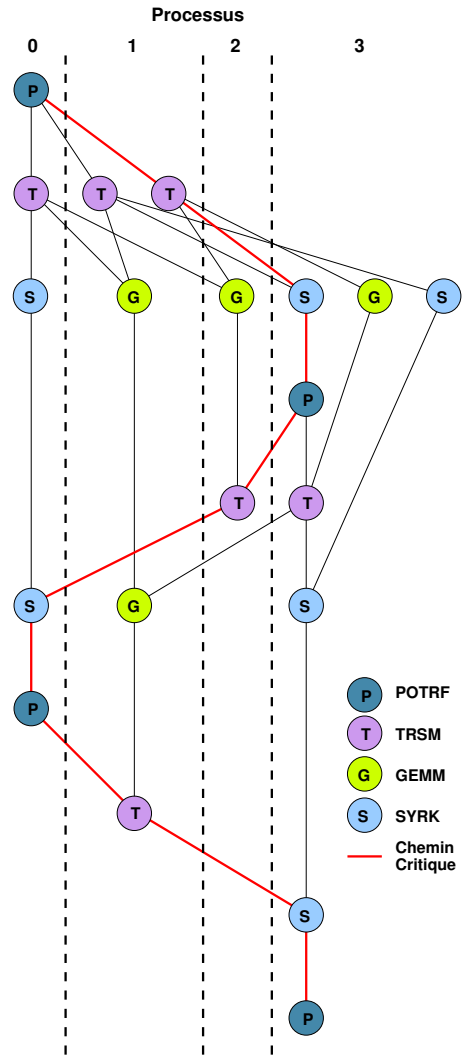


FIGURE 1.17 – Distribution bloc-cyclique des données de la matrice A d'une factorisation de Cholesky, et exemple de distribution automatique des tâches inférée par StarPU-MPI entre 4 processus.

tâches, StarPU-MPI est capable de détecter si une tâche soumise sur un autre processus va modifier une des données précédemment transférées, et peut effacer la donnée du cache de communications. Ainsi, la prochaine fois qu'une communication sur cette donnée va être inférée, la communication de la nouvelle donnée va être programmée par StarPU-MPI. Ainsi, grâce aux propriétés de la réplication séquentielle du graphe de tâches, StarPU-MPI est capable de détecter et d'élaguer les communications de données redondantes entre processus tout en conservant la cohérence des versions des données, et ce sans nécessiter de synchronisations.

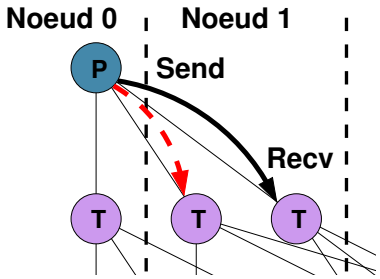


FIGURE 1.18 – Si deux tâches ayant une dépendance entre elles s'exécutent sur des processus différents, une communication MPI est inférée entre eux. De plus, comme la communication en pointillés rouge est redondante avec la noire, il est possible de l'éliminer.

Bien que l'utilisation de supports d'exécution à base de tâches facilite grandement la portabilité des performances en abstrayant l'architecture des machines pour le programmeur et en automatisant les optimisations de performance, il est également nécessaire de questionner le surcoût induit par leur utilisation. En effet, contrairement à une application écrite dans le modèle de programmation MPI+X où un programmeur expert peut optimiser finement l'exécution des calculs à l'écriture du programme, le calcul dynamique des optimisations de performance de l'application par un support d'exécution peut avoir un coût non négligeable par rapport au temps d'exécution du programme, notamment lors du passage à l'échelle. Nous présentons dans la section suivante les compromis, entre apports et surcoûts, que nous explorons dans cette thèse pour permettre le passage à l'échelle des applications utilisant StarPU.

1.4 Entre apports et surcoûts pour le passage à l'échelle

Dans cette section, nous discutons des surcoûts pour le passage à l'échelle des trois composantes du support d'exécution StarPU : son modèle de programmation, sa gestion de la mémoire et sa gestion de l'exécution des tâches. Pour chacun, nous détaillons les points pouvant limiter le passage à l'échelle des applications portées sur StarPU, points sur lesquels portent les contributions de cette thèse.

1.4.1 Pour le modèle de programmation STF

La soumission séquentielle des tâches pose plusieurs problèmes pour le passage à l'échelle. Le coût de la soumission des tâches peut devenir prohibitif dans le cadre d'applications où le temps d'exécution des tâches est de l'ordre de grandeur du temps de leur soumission. On peut citer pour exemple le cas de la bibliothèque ScalFMM [9], qui utilise StarPU pour résoudre des problèmes à N corps avec une méthode FMM, qui a besoin de calculer un grand nombre de tâches, potentiellement très petites. Par exemple, pour le cas d'un cube dans lequel 1 million de particules sont réparties de manière uniforme, et où les particules sont calculées par blocs de 100, le temps moyen de calcul d'une tâche est de l'ordre de 3 millisecondes, là où le temps de soumission d'une tâche est de l'ordre de 300 microsecondes, soit à peine 10 fois moins. On comprend alors aisément que la soumission séquentielle des tâches limite fortement le parallélisme disponible pour nourrir les unités de calcul.

À l'inverse, si la soumission des tâches est plus rapide que leur exécution et que l'application soumet un grand nombre de tâches, l'enregistrement de toutes les informations liées au graphe de tâches (codelets utilisées, données accédées et modes d'accès associés, dépendances entre tâches, données à allouer, etc.) a un coût en temps et en espace non négligeable.

La réplication du graphe de tâches sur tous les processus contribue également à la surcharge d'informations lors du passage à l'échelle. En effet, plus le nombre de processus impliqués dans le calcul augmente, plus les données sont réparties entre les processus, et plus la sous-partie du graphe de tâches à laquelle un processus doit contribuer est petite. Ainsi, chaque processus soumet de plus en plus de tâches inutiles au calcul local, ce qui a également un coût non négligeable, notamment lors du passage à l'échelle.

Pour résoudre ces problèmes, il est possible soit de modifier la granularité des tâches pour augmenter leur temps d'exécution, soit de réduire le coût de la soumission des tâches pour permettre aux applications de soumettre plus de tâches, et potentiellement plus petites. Nos contributions, qui s'axent sur ce second point, seront détaillées dans le Chapitre 2.

1.4.2 Pour la consommation de mémoire

Comme le modèle de programmation de StarPU dissocie complètement la soumission des tâches de leur exécution, le programmeur est incité à soumettre les tâches le plus tôt possible, afin que le support d'exécution puisse prendre de bonnes décisions concernant les optimisations de performance. On peut penser au recouvrement des communications par du calcul par exemple. Cependant, comme expliqué en section précédente, l'enregistrement de toutes les informations liées aux tâches par le support d'exécution a un coût en temps, mais également un coût en mémoire. En effet, de par son modèle de programmation,

StarPU doit allouer à la soumission des tâches différentes structures pour les tâches, les données ou les dépendances entre tâches.. Enfin, à la terminaison des tâches, toutes ces données sont déallouées. Nous présentons dans le Chapitre 2 une étude des problèmes que cela soulève ainsi qu'une méthode pour les contourner.

Dans le cadre du calcul distribué, cette soumission non bornée des tâches par l'application peut également provoquer des débordements de mémoire. En effet, pour pouvoir recouvrir les communications entre processus par du calcul, StarPU-MPI tente de démarrer les communications de données dès que possible, et alloue toute la mémoire nécessaire pour les recevoir dès la soumission des tâches. Cette allocation anticipée de mémoire peut alors provoquer des débordements de mémoire si trop de tâches sont soumises en avance de l'exécution. Pour contrôler le flot de soumission de tâches afin d'empêcher l'allocation trop aggressive de ressources, nous présentons dans le Chapitre 2 une méthode basée sur les propriétés du modèle STF qui permet au programme de réguler le flot de soumission de tâches en fonction de critères tels que le nombre de tâches soumises ou la quantité de mémoire consommée par les tâches.

Dans le cadre du code de simulation du CEA, nous souhaitons également étudier le passage à l'échelle d'un solveur linéaire plus complexe, qui opère sur des matrices qui sont compressées numériquement. La particularité de ce solveur est que le taux de compression de ces matrices évolue de manière imprédictible durant l'exécution, ce qui modifie dynamiquement la mémoire consommée par le programme. En particulier, le contrôle de l'encombrement de mémoire tel que présenté en Chapitre 2 ne fonctionne pas pour ce solveur car la consommation de mémoire d'une tâche est calculé à la soumission de la tâche. StarPU ignore tout de l'évolution de la taille des données, ce qui peut provoquer des débordements de mémoire si ces données grossissent de manière trop importante. Pour résoudre ce problème, nous présentons dans le Chapitre 3 une extension du contrôle de l'encombrement mémoire qui permet au programmeur de fournir à StarPU des estimateurs de l'évolution de la taille des données. StarPU va alors se servir de ces estimateurs pour tenter de prédire à la soumission des tâches l'évolution de la taille des données, et pour corriger automatiquement les estimations à la terminaison des tâches, lorsque la taille des données est connue.

1.4.3 Pour la distribution du travail aux unités de calcul

Bien que l'ordonnancement de tâches permette d'optimiser le makespan d'une application, calculer une heuristique d'ordonnancement de tâches a un coût. Les heuristiques simples comme *eager* sont peu coûteuses à calculer mais ne permettent pas d'obtenir de bonnes performances pour des problèmes d'ordonnancement complexes, comme l'ordonnancement sur architectures hétérogènes par exemple, présenté précédemment en Section 1.3.1. À l'inverse, des

heuristiques évoluées comme HEFT réussissent à proposer des ordonnancements performants pour des problèmes complexes, mais peuvent parfois être très coûteuses à calculer. Par exemple, le coût en calcul de l'heuristique HEFT est directement proportionnel au nombre d'unités de calcul présentes sur la machine. Dans le cadre de processeurs manycores comme l'Intel KNL, qui disposent de plusieurs dizaines de cœurs de calcul par carte, il peut être contre-productif d'utiliser une telle heuristique car son coût en calcul peut devenir prohibitif. [16]

Pour pouvoir tirer le meilleur des deux mondes, nous souhaitons combiner l'utilisation de différentes heuristiques au sein des ordonnanceurs de tâches. Nous proposons ainsi dans le Chapitre 4 une implémentation modulaire des ordonnanceurs de tâches offrant un cadre pour aider les programmeurs d'ordonnanceurs à coupler différentes heuristiques d'ordonnancement et à faire évoluer les ordonnanceurs de tâches dans StarPU.

Chapitre 2

Passage à l'échelle du support d'exécution StarPU

"Quand on a pas de technique, il faut y aller à la zob."
– Perceval, *Kaamelott Livre III, Morituri*

Résumé du chapitre **42**
2.1 Contexte expérimental **43**
2.2 Gérer la soumission séquentielle des tâches en distribué **43**
2.2.1 Surcoûts de la réplication du graphe de tâches . . . 44
2.2.2 Contribution : retirer les tâches non pertinentes . . . 45
2.2.3 Résultats 46
2.2.4 Discussion 47
2.3 Inconvénients des appels système pour l'allocation de mémoire **48**
2.3.1 Comportement des allocations de mémoire 48
2.3.2 Contribution : utilisation d'un cache d'allocation mémoire 52
2.3.3 Discussion 53
2.4 Contrôler le flot de soumission des tâches **53**
2.4.1 Rationalisation de l'utilisation des ressources 54
2.4.2 Contribution : maîtriser le flot de soumission de tâches 54
2.4.3 Contribution : politiques de contrôle du flot de soumission de tâches 55
2.4.4 Résultats 57
2.4.5 Discussion 60
2.5 Évaluation générale du passage à l'échelle **62**
2.5.1 Contexte expérimental 62
2.5.2 Résultats et comparaison avec l'état de l'art 63
2.5.3 Discussion 65

2.6 Intégration dans la chaîne de simulation logicielle 3D 66
Synthèse 67

Résumé du chapitre

Nous présentons dans ce chapitre une étude des solutions mises en œuvre pour permettre le passage à l'échelle du support d'exécution StarPU sur un grand nombre de nœuds de calcul. Nous présentons des résultats de performances obtenus sur un solveur d'algèbre linéaire dense utilisant StarPU jusqu'à 144 nœuds de calcul hybrides du supercalculateur TERA-100 du CEA (1152 CPUs et 288 GPUs).

Pour réduire le coût de la soumission séquentielle du graphe de tâches sur des grandes grappes de machines, nous présentons en Section 2.2 une méthode d'élagage du graphe de tâches qui limite fortement le nombre de tâches soumises par processus. Nous discutons en Section 2.3 du fonctionnement des appels système allocateurs de mémoire de Linux et pourquoi leur comportement peut limiter les performances de StarPU. Pour améliorer la réutilisation des données, nous présentons une méthode pour cacher les allocations de données effectuées par StarPU durant la soumission des tâches. Comme les données allouées par StarPU ne sont libérées qu'à la terminaison des tâches et que la soumission des tâches est souvent bien plus rapide que leur exécution, il peut être nécessaire de ralentir le flot de soumission de tâches pour que le cache d'allocations fonctionne correctement. Nous présentons ainsi dans la Section 2.4 une méthode pour réguler le flot de soumission de tâches qui se base sur le modèle STF, ainsi que deux politiques de contrôle de ce flot. Nous présentons en Section 2.5 une comparaison des performances du solveur Chameleon, qui utilise StarPU, aux solutions de référence de l'état de l'art ScaLAPACK et DPLASMA. Enfin, nous expliquons en Section 2.6 les étapes nécessaires à l'intégration du solveur Chameleon dans le code de simulation du CEA.

2.1 Contexte expérimental

Les Tables 2.1, 2.2 et 2.3 présentent respectivement l'architecture des machines, les bibliothèques de calcul et les différentes configurations de la factorisation de Cholesky que nous avons utilisé pour faire les expériences de cette thèse. En plus du supercalculateur TERA-100 du CEA-DAM, nous avons également pu utiliser la partition Mirage de la plateforme collaborative PlaFRIM [1] [2] pour nos expériences. Comme nos calculs nécessitent une grande précision, tous les calculs ont été faits en double précision.

Table 2.1 – Architectures des machines utilisées.

Machine	CPU	GPU	Mémoire	Interface réseau
TERA-100 Hybrid	Intel Xeon E5620 (4 cœurs) @ 2.4 GHz x 2	NVIDIA Tesla M2090 (6 Go) x 2	24 Go	Infiniband QDR @ 40 Go/s
TERA-100 Parallele	Intel Xeon X7560 (8 cœurs) @ 2.26 GHz x 4	-	128 Go	Infiniband QDR @ 40 Go/s
PlaFRIM Mirage	Intel Xeon X5650 (8 cœurs) @ 2.67 GHz x 2	NVIDIA Tesla M2070 (6 Go) x 3	36 Go	Infiniband QDR @ 40 Go/s

Table 2.2 – Compilateurs et bibliothèques utilisées.

Machines / Bibliothèques	Compilateur C	Compilateur GPU	Bibliothèque MPI	Bibliothèque BLAS
TERA-100 Hybrid	Intel ICC 14.0.3.174	NVIDIA CUDA 4.2	BullxMPI 1.2.8.2	Intel MKL 14.0.3.174
TERA-100 Parallele	Intel ICC 14.0.3.174	-	BullxMPI 1.2.8.2	Intel MKL 14.0.3.174
PlaFRIM Mirage	GNU GCC 4.8.3	NVIDIA CUDA 5.5	OpenMPI 1.6.5	Intel MKL 11.0.1.117

2.2 Gérer la soumission séquentielle des tâches en distribué

Maintenant que nous avons intégré notre solveur sur support d'exécution Chameleon dans la chaîne de simulation logicielle 3D, nous souhaitons optimi-

1. Plateforme Fédérative pour la Recherche en Informatique et Mathématiques.

Table 2.3 – Configurations utilisées pour les factorisations de Cholesky.

Numéro	Taille de bloc	Éléments
1	512	Réel
2	512	Complexe
3	960	Réel

ser les performances de ce solveur qui occupe la majorité du temps de calcul de notre simulation. Nous discutons dans cette section des limites de la soumission séquentielle du graphe de tâches lorsque le nombre de nœuds de calcul augmente. Nous montrons qu'il est possible d'élaguer le graphe de tâches selon le processus qui le déroule, ce qui permet au modèle de programmation STF de passer à l'échelle sur une centaine de nœuds de calcul et théoriquement, dans notre cas, au-delà du million de nœuds de calcul.

2.2.1 Surcoûts de la réplication du graphe de tâches

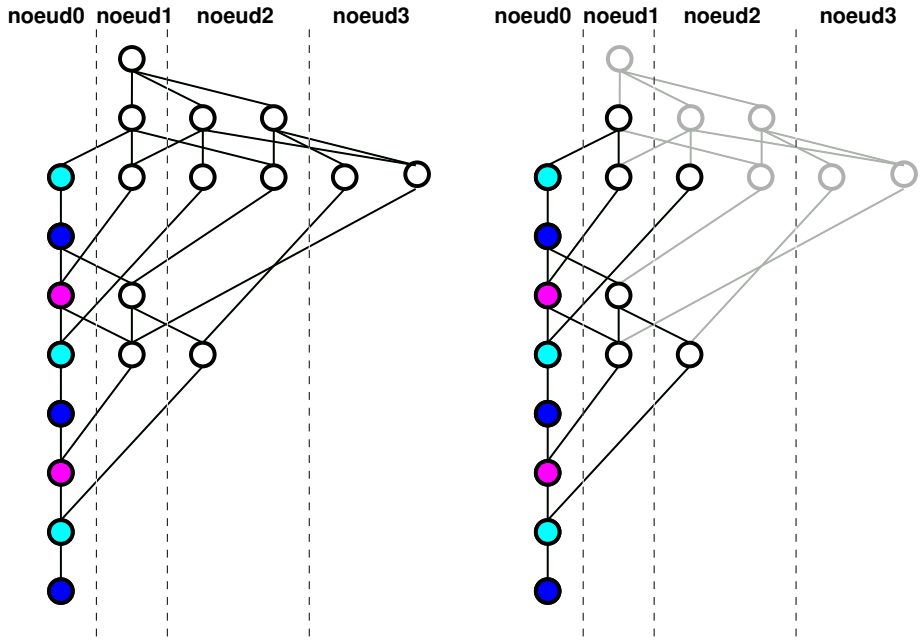
Comme présenté en Section 1.2.2, plusieurs paradigmes existent pour tirer parti du modèle de programmation STF pour le calcul distribué. Dans le paradigme maître-esclave, le maître prend la décision de répartition du travail et la transmet à ses esclaves. Bien que ce paradigme ait l'avantage d'être relativement simple et de permettre de répartir la charge facilement, le maître devient un point de contention non négligeable qui peut fortement limiter le passage à l'échelle de ce modèle.

Un autre paradigme consiste à répliquer le déroulement du graphe de tâches dans chaque processus. En combinant ce modèle distribué au modèle STF, qui stipule que chaque tâche est soumise séquentiellement, il est possible de garantir que, même si la soumission du graphe de tâches n'est pas synchronisée entre les processus, toutes les décisions concernant la distribution du calcul sont prises **à la même étape de la soumission séquentielle du graphe de tâches** sur l'ensemble des processus. Grâce à cette propriété du modèle STF, la cohérence distribuée du graphe de tâches entre les processus est assurée sans qu'ils n'aient à échanger de messages de contrôle, ce qui permet un excellent passage à l'échelle de ce modèle.

Ce modèle par réplication du graphe de tâches est celui qui a été choisi pour implémenter le support implicite du calcul distribué de StarPU dans StarPU-MPI (cf. Section 1.3.2). Ce modèle n'est cependant pas exempt de tout défaut : il stipule en effet qu'il faut dérouler tout le graphe de tâches sur tous les processus, même si seulement une sous-partie de ce graphe est exécutée sur celui-ci. C'est pourquoi YarKhan et al. soutiennent dans [72] que la combinaison de ce modèle distribué avec la soumission séquentielle des tâches du modèle STF ne permet pas son passage à l'échelle. Nous montrons toutefois

dans la section suivante qu'il est possible de passer outre cette limitation en retirant les tâches non pertinentes du graphe de tâches grâce à un élagage intelligent de ce graphe en fonction du processus qui le déroule.

2.2.2 Contribution : retirer les tâches non pertinentes



(a) Graphe de tâches non élagué (b) Graphe de tâches élagué

FIGURE 2.1 – Exemple d'élagage du graphe de tâches sur le processus MPI 0 pour une factorisation de Cholesky d'une matrice 4x4 blocs sur 4 processus. Les tâches élaguées sont grisées.

Dans le modèle par réplication du graphe de tâches, la condition nécessaire pour obtenir la cohérence distribuée du graphe de tâches entre les processus est que chacun d'entre eux analyse les dépendances entrantes, locales et sortantes des tâches qu'ils doivent exécuter localement (selon la distribution du calcul) afin de déterminer les actions à effectuer. Ainsi, le processus qui possède le plus de données en écriture (cf. Section 1.3.2) d'une tâche va recevoir toutes les données qu'il ne possède pas et exécuter la tâche. Un processus qui possède une ou plusieurs données en lecture de la tâche va envoyer ses données au processus qui doit exécuter la tâche. Un processus qui ne possède aucune des données de la tâche n'est impliqué ni dans l'exécution de la tâche, ni dans les communications nécessaires à l'exécution de la tâche. Il est donc possible d'*élaguer* la tâche sur ce processus.

La FIGURE 2.1 montre l'élagage du graphe de tâches de notre application sur le processus MPI 0 pour une matrice 4x4 blocs sur 4 processus. Les tâches

grisées n'ont pas besoin d'être soumises sur le processus MPI 0 car il ne possède aucune des données impliquées dans le calcul de ces tâches.

2.2.3 Résultats

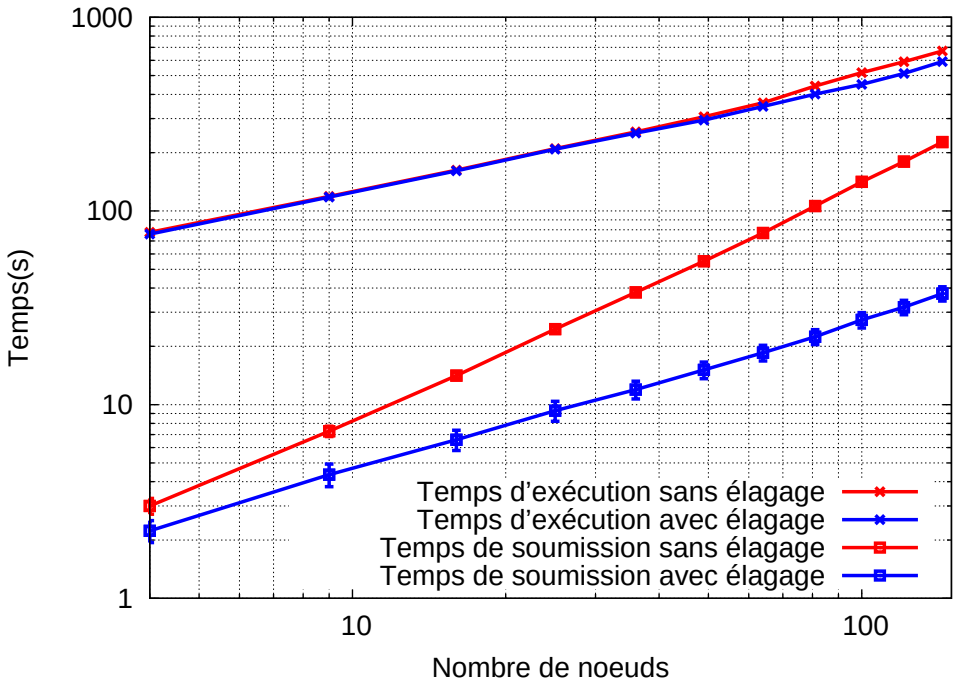


FIGURE 2.2 – Impact de l'élagage du graphe de tâches sur les temps de soumission et d'exécution en fonction du nombre de nœuds, à quantité de calcul par nœud constant. Les axes X et Y sont en échelle logarithmique.

La FIGURE 2.2 présente une étude comparative des temps de soumission et d'exécution de la configuration n°1 de notre application (cf. TABLE 2.3) en fonction de si l'application élague le graphe de tâches ou non, sur des grappes de 1 à 144 nœuds de calcul de la machine TERA-100 Hybrid (cf. TABLE 2.1), avec un processus MPI par nœud. Les tailles de matrices sont choisies pour conserver la même quantité de travail par processus en augmentant le nombre de processus (scalabilité *faible*).

Cette figure montre que, sans élagage du graphe de tâches, le temps d'exécution se dégrade lorsque le temps de soumission devient de l'ordre de grandeur du temps d'exécution. Ainsi, la soumission des tâches limite les performances de l'application dès 80 nœuds de calcul. Par contre, nous observons que l'élagage du graphe de tâches permet de réduire fortement le temps de soumission de ce graphe, ce qui permet d'obtenir les pleines performances de notre application jusqu'à 144 nœuds de calcul.

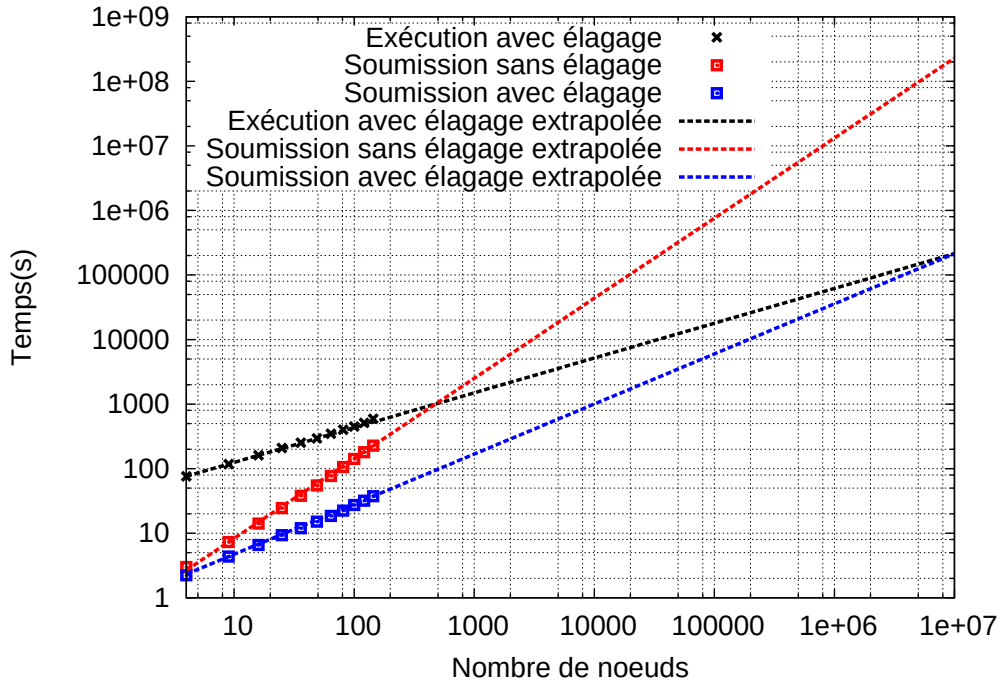


FIGURE 2.3 – Extrapolation de la FIGURE 2.2 jusqu'à 10 millions de nœuds de calcul.

Pour étudier la portée de cette contribution pour des grappes de machines plus grandes, nous avons extrapolé sur la FIGURE 2.3 le graphe précédent jusqu'à 10 million de nœuds de calcul. La première observation est que le temps de soumission sans élagage du graphe de tâches rattraperait le temps d'exécution pour des grappes de l'ordre de 500 nœuds de calcul. À ce stade, l'exécution deviendrait séquentielle car la soumission des tâches serait plus lente que leur exécution, ce qui concorde avec les observations présentées dans [72]. La seconde observation est qu'avec l'élagage du graphe de tâches, le temps de soumission ne rattraperait le temps d'exécution que pour des grappes de l'ordre de **10 millions** de nœuds de calcul, et le temps de soumission deviendrait du même ordre que le temps d'exécution à partir de grappes de 10 000 nœuds de calcul. Cela prouve que l'élagage du graphe de tâches diminue suffisamment le temps de soumission du graphe de tâches pour permettre le passage à l'échelle de cette application sur bien plus de nœuds de calcul que le nombre dont nous avons disposé pour cette étude.

2.2.4 Discussion

La technique d'élagage du graphe de tâches que nous propons est valide pour tous les types de calcul tant que la vérification de l'élagage d'une tâche se fait par rapport aux données accédées par cette tâche et que la distribution des

données sur les nœuds de calcul ne change pas pour les tâches déjà soumises au support d'exécution. De plus, cet élagage est optimisé car les tâches soumises localement après élagage impliquent toutes au moins une communication de données ou l'exécution de la tâche, et sont donc nécessaires pour conserver la cohérence distribuée du graphe de tâches.

Il est possible d'aller plus loin dans l'élagage du graphe de tâches car seules les tâches qui doivent être exécutées localement ont réellement besoin d'être soumises. Pour ne plus soumettre les tâches permettant d'inférer les communications, l'application peut remplacer la soumission de ces tâches par un préchargement du cache de communications de StarPU-MPI en explicitant ces communications dans l'application. Une technique pour automatiser cet élagage supplémentaire consiste à tirer parti de la représentation algébrique du graphe de tâches pour élaguer les tâches permettant d'inférer les communications et générer automatiquement les communications. Il existe une méthode présentée dans [24] permettant de convertir un graphe de tâches construit séquentiellement en représentation algébrique, puis de faire l'opération inverse pour obtenir un graphe de tâches complètement élagué ainsi que les communications à effectuer.

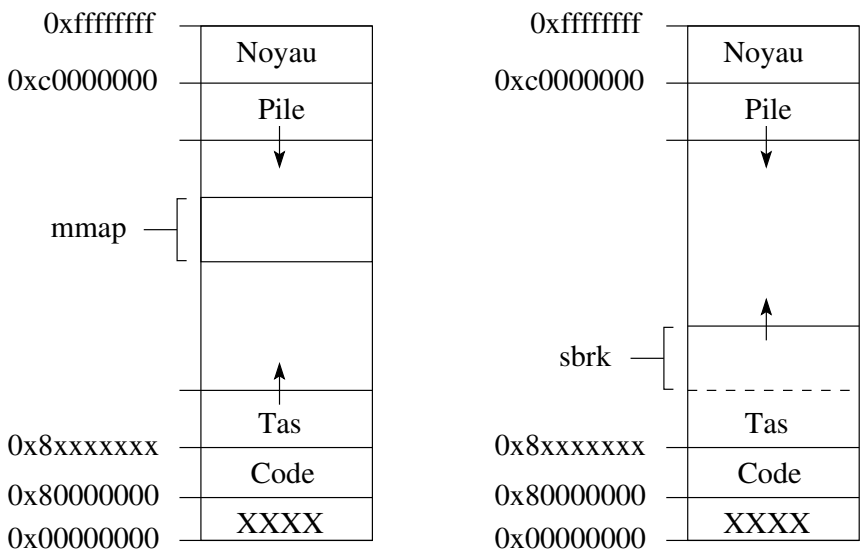
Si le temps de soumission du graphe de tâches élagué prend malgré tout trop de temps et limite les performances de l'application, il peut alors être nécessaire d'augmenter la granularité des tâches afin de réduire le nombre de tâches à soumettre. Des travaux sont en cours dans StarPU pour reconsidérer dynamiquement la granularité des tâches de l'application.

2.3 Inconvénients des appels système pour l'allocation de mémoire

Les besoins en simulation du CEA nécessitent de résoudre le système $Ax = B$ pour des matrices A de taille supérieure au million d'inconnues. Les travaux précédant cette thèse s'étant limités à des matrices de taille inférieures à 100 000 inconnues, nous avons étudié le comportement de StarPU lorsque la taille de matrice augmente. Nous avons alors observé des problèmes soit de performance, soit d'encombrement mémoire important, en fonction de l'appel système allocateur de mémoire utilisé par l'allocateur standard *malloc* de la GNU libc (glibc), comme présenté en Figure 2.5. Cette section explique le problème rencontré avec chaque stratégie d'allocation mémoire et propose d'utiliser un cache d'allocation pour les contourner.

2.3.1 Comportement des allocations de mémoire

Le système d'exploitation Linux, comme beaucoup d'autres systèmes de type Unix, propose deux appels systèmes pour allouer des pages physiques dans



(a) *mmap* (b) *sbrk*

FIGURE 2.4 – Positionnement dans l'espace d'adressage d'un processus Linux des zones mémoire allouées par les appels système *mmap* et *sbrk*.

un processus, appelés *mmap* et *sbrk*. Ces appels systèmes sont ensuite utilisés par des bibliothèques en espace utilisateur pour proposer des allocateurs de mémoire standard, comme la routine *malloc* de la bibliothèque standard GNU libc (glibc). Le fonctionnement de l'allocation de mémoire avec ces appels système est présenté sur la FIGURE 2.4. L'appel système *mmap* projette des pages physiques dans l'espace d'adressage virtuel du processus pour les rendre accessibles par celui-ci. Cet espace peut être projeté n'importe où entre le bas de la pile et le haut du tas de l'espace d'adressage du processus. L'appel système *sbrk* permet de déplacer le pointeur indiquant le haut du tas de l'espace d'adressage virtuel du processus. Si le pointeur est déplacé vers le haut, l'appel système réserve le nombre de pages physiques nécessaires et les projette en haut du tas de l'espace d'adressage virtuel du processus. À l'inverse, si le pointeur est déplacé vers le bas, l'appel système rend les pages physiques qui ne sont plus utilisées au système d'exploitation. Comme il n'est possible de déallouer que la mémoire située en haut du tas avec cet appel système, des trous se forment dans le tas lorsque de la mémoire qui n'est pas en haut du tas est libérée : le tas se *fragmente*. Pour limiter cette fragmentation, l'allocateur de mémoire *malloc* implémente des stratégies de réutilisation des trous du tas en fonction de la taille des données à allouer.

La FIGURE 2.5 montre le pic d'encombrement mémoire du nœud de calcul le plus chargé en mémoire (haut) et les performances (bas) de la configuration n°1 de notre application (cf. TABLE 2.3) sur 36 nœuds de calcul de la machine TERA-100 Hybrid (cf. TABLE 2.1) en fonction de la taille de la matrice calculée

pour plusieurs stratégies d'allocation de mémoire. Les GPUs des nœuds de calcul ne sont pas utilisés dans cette expérience.

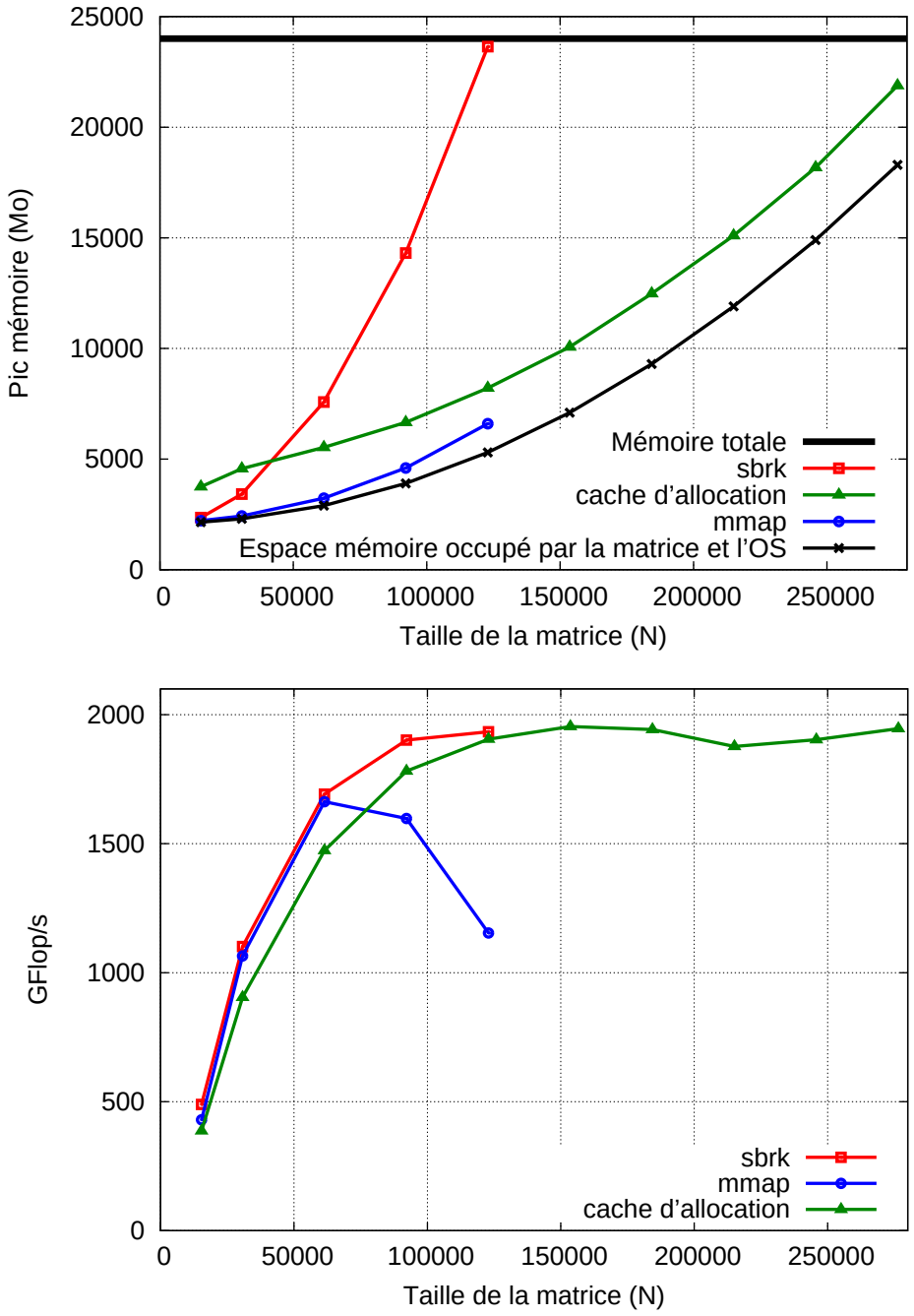


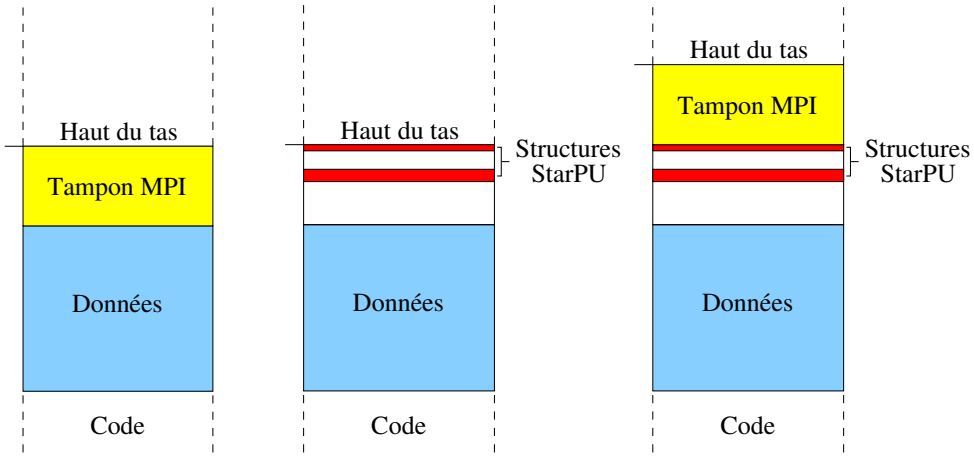
FIGURE 2.5 – Pic d'empreinte mémoire (haut) et performances (bas) avec chaque stratégie d'allocation.

Coût des déallocations avec *munmap*

Lorsque l'allocateur standard alloue sa mémoire avec l'appel système *mmap*, les performances de notre application s'effondrent pour des tailles de matrices supérieures à 64 000. Ce problème est dû au coût de l'appel à la routine *munmap* pour libérér de la mémoire. En effet, un appel à *munmap* pour libérer des pages physiques projetées dans l'espace d'adressage d'un processus nécessite de synchroniser tous les cœurs de calcul avec le nouvel état de l'espace d'adressage. Cette synchronisation provoque un effondrement des performances lors du passage à l'échelle car StarPU effectue un grand nombre de petites allocations et surtout déallocations, pour les structures de tâches et de données notamment.

Forte fragmentation du tas avec *sbrk*

Lorsque l'allocateur standard alloue sa mémoire avec l'appel système *sbrk*, la consommation mémoire explose à cause d'une forte fragmentation du tas. En effet, l'allocateur autorise les petites allocations de StarPU (notamment les structures de tâches ou de données) à polluer les zones mémoires allouées précédemment pour des grosses données (dans notre cas les tampons de réception MPI). Cela oblige l'allocateur standard *malloc* à augmenter la taille du tas



(a) 1er gros tampon MPI (b) Petites données StarPU (c) 2e gros tampon MPI

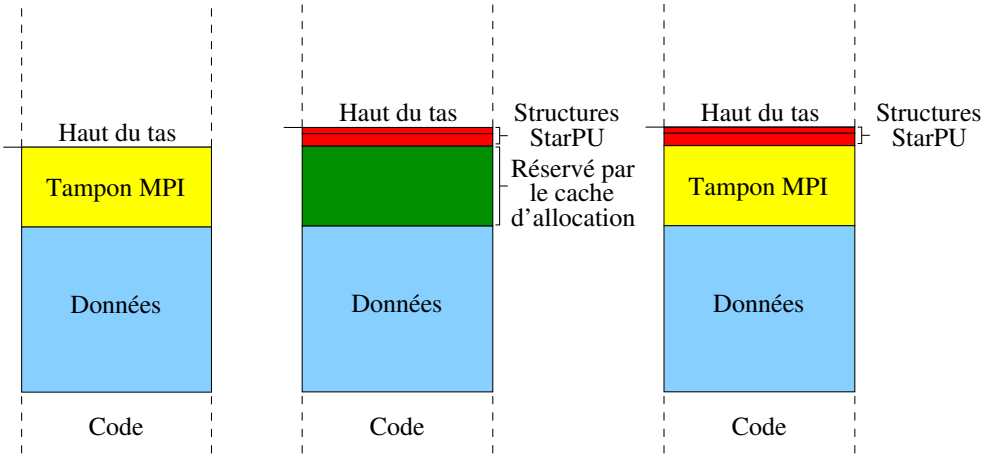
FIGURE 2.6 – Fragmentation du tas avec *sbrk*.

par un appel à l'appel système *sbrk* à chaque réception de tampon MPI pour réserver une zone contigüe de mémoire dans le tas permettant l'allocation du tampon, comme le montre la FIGURE 2.6. De plus, toutes les allocations et déallocations mémoire des tampons MPI de notre application se font dans le même ordre. Comme la dernière zone mémoire allouée est déallouée en dernier, l'allocateur ne peut pas diminuer la taille du tas du processus avant la

fin de l'exécution. Ces deux effets combinés provoquent un débordement de la mémoire à cause d'une fragmentation très importante du tas.

2.3.2 Contribution : utilisation d'un cache d'allocation mémoire

Pour contourner ces problèmes sans réécrire ou modifier les allocateurs standards, nous proposons de ne pas libérer les zones de mémoire allouées par StarPU et de les réutiliser pour d'autres allocations. Notre contribution étend le cache d'allocations de StarPU, utilisé au départ pour gérer la réutilisation des zones mémoire dédiées aux transferts avec les GPUs, afin de lui faire gérer toutes les allocations effectuées par StarPU. Lorsque de la mémoire allouée par StarPU est libérée, elle est désormais stockée dans le cache d'allocations au lieu d'être libérée. Quand StarPU alloue de la mémoire, il regarde dans le cache d'allocations si un espace mémoire de la même taille que la donnée en cours d'allocation est réutilisable. Si oui, il utilise cet espace pour allouer la donnée, sinon il alloue la mémoire classiquement avec un appel à l'allocateur standard *malloc* de la glibc.



(a) 1er gros tampon MPI (b) Petites données StarPU (c) 2e gros tampon MPI

FIGURE 2.7 – Fragmentation du tas avec le cache d'allocation.

Les courbes *cache d'allocation* de la FIGURE 2.5 montrent que le cache d'allocations permet à notre application de passer à l'échelle jusqu'à des matrices de l'ordre de 300 000 inconnues tant en termes de performances qu'en termes de consommation mémoire. Comme le cache d'allocations ne libère pas les données allouées avant la dé-initialisation de StarPU, il corrige de par son fonctionnement les problèmes de performance observés lorsque l'allocateur standard utilise *mmap* pour allouer sa mémoire car il empêche les synchronisations entre les cœurs de calcul dues aux déallocations avec *munmap* durant

l'exécution de l'application. Le cache d'allocations permet également de limiter la fragmentation du tas due à la stratégie de l'allocateur standard lorsqu'il utilise l'appel système *sbrk*, comme le montre la FIGURE 2.7, car il gère désormais correctement la réallocation de la zone de mémoire du tampon MPI précédemment libérée pour allouer le nouveau tampon MPI. Le cache d'allocations permet donc à l'application d'utiliser l'allocateur de mémoire de son choix sans avoir à se préoccuper de l'appel système utilisé en interne.

2.3.3 Discussion

Il est nécessaire de rappeler que l'utilisation d'un cache d'allocations peut provoquer des débordements de mémoire si ce dernier grossit au point de remplir entièrement la mémoire des nœuds de calcul. Une solution existant dans StarPU pour contourner ce problème consiste à transférer une partie des données sur des mémoires *externes*, comme un disque dur par exemple. L'utilisation de ces mémoires pour le calcul reste ardu car le temps de transfert des données vers et depuis ces mémoires est souvent bien supérieur (de plusieurs ordres de grandeur) au temps de transfert des données entre la mémoire principale d'une machine et son processeur. Il peut donc être difficile de recouvrir efficacement ces transferts de données par du calcul.

Un autre problème concerne la soumission très rapide des tâches. En effet, comme le cache d'allocations est consulté à la soumission des tâches mais n'est peuplé qu'à leur terminaison, si la soumission des tâches est beaucoup plus rapide que leur exécution (ce qui est le cas de notre application), le cache d'allocations est en fait très peu réutilisé car la plupart des tâches ont déjà été soumises avant même que le cache d'allocations ne se remplisse. La section suivante discute de ce problème de distance entre les fronts de soumission et d'exécution des tâches et propose une solution pour réguler le flot de soumission de tâches.

2.4 Contrôler le flot de soumission des tâches

Soumettre les tâches le plus tôt possible au support d'exécution lui permet de prendre des décisions à l'avance, comme inférer et poster les communications entre nœuds de calcul pour avoir un bon recouvrement des communication par du calcul par exemple. Cette soumission agressive du graphe de tâches peut cependant avoir des effets indésirables, notamment au niveau de l'utilisation des ressources des machines. Dans le cas de notre application, c'est la surréservation des ressources de mémoire qui provoque des débordements de la mémoire des nœuds de calcul, comme le montre le graphe du haut de la FIGURE 2.9. Nous présentons dans cette section une méthode pour maîtriser le flot de soumission de tâches en fonction de l'utilisation des ressources des

machines.

2.4.1 Rationalisation de l'utilisation des ressources

Pour éviter cette surutilisation des ressources, nous voulons que le support d'exécution puisse contrôler l'attribution des ressources aux tâches. Ces ressources étant accédées de manière concurrente par des tâches ayant des dépendances entre elles, de mauvais choix d'attributions peuvent conduire à un blocage total de l'application. Ce problème d'interblocage lié à l'accès simultané à des ressources concurrentes [32] a été largement étudié dans la littérature, qui distingue trois classes de solutions pour y remédier. La *récupération* depuis un interblocage consiste à enregistrer régulièrement des états valides de l'exécution. Si un interblocage survient, le dernier état valide enregistré est restauré et l'exécution est rejouée avec un nouveau choix d'allocation de ressources. L'*évitement* des interblocages consiste à déléguer à un tiers éclairé l'allocation des ressources disponibles selon l'algorithme du banquier proposé par [36]. Enfin, la *prévention* des interblocages consiste à empêcher les demandeurs de ressources de réserver plus que les ressources disponibles en faisant attendre les demandeurs jusqu'à ce que les ressources demandées soient disponibles.

Dans le cadre de notre application, nous ne souhaitons pas faire de *récupération* depuis un interblocage car nous voulons saturer la mémoire des machines avec les données du calcul, et les machines n'ont pas de mémoire externe comme des disques dur. Il nous est donc difficile de stocker le ou les points de sauvegarde nécessaires pour permettre le retour en arrière dans l'exécution sans augmenter le nombre de nœuds de calcul ou réduire la taille du cas traité. Il est possible de faire de l'*évitement* des interblocages dans StarPU, mais cela nécessite une modification importante du cœur de StarPU. Par contre, nous pouvons faire de la *prévention* des interblocages simplement dans StarPU en renforçant le contrat du support d'exécution à l'application qui soumet des tâches. StarPU peut garantir à l'application que toutes les ressources nécessaires à l'exécution d'une tâche soient réservées à la soumission de la tâche, et bloquer la soumission des tâches si les ressources nécessaires à son exécution ne sont pas encore disponibles.

2.4.2 Contribution : maîtriser le flot de soumission de tâches

Pour pouvoir arrêter le flot de soumission de tâches sans provoquer d'interblocages dus aux dépendances entre tâches, le support d'exécution doit pouvoir introduire un ordre séquentiel à la soumission des tâches pour garantir que, lors de l'arrêt de la soumission des tâches, toutes les tâches déjà acceptées en soumission par le support d'exécution peuvent s'exécuter. Grâce à cet ordre sé-

quentiel, le support d'exécution dispose également de la connaissance de l'état d'utilisation *au pire* des ressources durant l'exécution parallèle, c'est-à-dire si l'intégralité des ressources réservées par le support d'exécution pour exécuter les tâches acceptées en soumission sont consommées. Il est donc possible de maîtriser l'utilisation des ressources grâce au contrôle du flot de soumission des tâches sans imposer d'ordre séquentiel à l'exécution parallèle. Comme le modèle de programmation STF, par conception, permet la soumission séquentielle du graphe de tâches, tout support d'exécution qui implémente le modèle STF (comme StarPU) introduit donc nativement cet ordre séquentiel.

Pour maîtriser l'utilisation d'une ressource dans ce type de support d'exécution, le principe est de calculer à la soumission de chaque tâche la quantité de cette ressource qui doit être réservée par le support d'exécution pour garantir l'exécution correcte de la tâche. Si les ressources sont disponibles, le support d'exécution enregistre la réservation dans un gestionnaire de ressources et accepte la tâche en soumission ; sinon, il bloque la soumission de tâches. Pendant que la soumission des tâches est bloquée, l'exécution des tâches acceptées en soumission se poursuit et libère les ressources réservées à leur terminaison. Lorsqu'une quantité suffisante de la ressource est disponible, le support d'exécution débloque alors la soumission de tâches et accepte la tâche en attente de soumission. Cette régulation du flot de soumission de tâches permet ainsi d'adapter dynamiquement la distance entre le front de soumission et le front d'exécution des tâches en fonction des ressources consommées.

2.4.3 Contribution : politiques de contrôle du flot de soumission de tâches

Avec StarPU, l'application peut volontairement contraindre la soumission des tâches grâce à la fonction `starpu_task_wait_for_n_submitted`. Un appel à cette routine fait attendre le thread appelant jusqu'à ce que le nombre de tâches soumises à StarPU descende sous le seuil qui lui est donné en paramètre. Bien que cette routine permette au programmeur de contrôler le flot de soumission de tâches, il reste difficile pour le programmeur de choisir quand et comment contrôler le flot de soumission de tâches pour contrôler l'utilisation des ressources des machines sans avoir une forte expertise dans le domaine des supports exécutifs, du calcul parallèle et du calcul distribué. Pour répondre à ce problème, nous avonc implémenté dans StarPU deux politiques permettant de contrôler le flot de soumission de tâches. La première régule le nombre de tâches que StarPU accepte en soumission, et la seconde contrôle son encombrement de mémoire.

Contrôle du nombre de tâches soumises

Pour implémenter cette politique, trois actions sont nécessaires : enregistrer le nombre de tâches soumises, bloquer la soumission de tâches lorsque le nombre

de tâches soumises dépasse un seuil maximal, et la débloquer lorsque le nombre de tâches descend sous un seuil minimal. Pour ce faire, nous avons étendu le gestionnaire de tâches interne de StarPU, qui décompte le nombre de tâches acceptées en soumission par StarPU, pour qu'il déclenche automatiquement le blocage et le déblocage de du flot de soumission de tâches. Ainsi, lorsqu'une tâche est soumise à StarPU par l'application, l'incrémentation du compteur de tâches va bloquer la soumission si le nombre de tâche soumises à StarPU dépasse le seuil maximal. De même, à la terminaison de chaque tâche, la décrémentation du compteur va alors débloquer automatiquement la soumission de tâches si le nombre de tâches soumises à StarPU descend sous le seuil minimal. Les seuils de blocage et déblocage de la soumission de tâches de cette politique sont définis par l'utilisateur à travers les variables d'environnement `STARPU_LIMIT_MAX_SUBMITTED_TASKS` et `STARPU_LIMIT_MIN_SUBMITTED_TASKS`.

Contrôle de l'encombrement mémoire

Pour implémenter cette politique, trois actions sont nécessaires : enregistrer la quantité de mémoire nécessaire à l'exécution d'une tâche, bloquer la soumission de tâches lorsque la quantité de mémoire allouée dépasse un seuil maximal, et la débloquer lorsque la quantité de mémoire allouée descend sous un seuil minimal. Comme StarPU dispose déjà d'un gestionnaire de mémoire qui enregistre les allocations de mémoire internes au support d'exécution, notre contribution étend ce gestionnaire pour lui permettre d'enregistrer également les données automatiquement allouées par StarPU via les interfaces de données utilisateur. Nous avons implémenté de nouvelles routines `starpu_memory_allocate` et `starpu_memory_deallocate` qui permettent d'avertir le gestionnaire de mémoire de StarPU qu'une quantité de données vient d'être allouée ou libérée, respectivement. Ces routines sont appelées par les interfaces de données StarPU dans leurs méthodes `allocate` et `free`, à côté des allocations et déallocations de mémoire. Grâce à ce décompte de la mémoire allouée dans les interfaces de données, le gestionnaire de mémoire de StarPU calcule à la soumission de chaque tâche les ressources mémoire nécessaires pour exécuter cette tâche en appelant la méthode `allocate` des interfaces de toutes les données de la tâche, et bloque la soumission des tâches lorsque la quantité de mémoire allouée atteint le seuil maximal. De même, lors de la terminaison des tâches, StarPU est averti de la libération des ressources mémoire par les appels aux routines `starpu_memory_deallocate` situées dans les méthodes `free` des interfaces des données et débloque la soumission des tâches si la quantité de mémoire allouée descend sous le seuil minimal. Les seuils de blocage et de déblocage de la soumission des tâches de cette politique sont données par l'utilisateur à travers les variables d'environnement `STARPU_MEMORY_THRESHOLD_SLEEP/WAKEUP`.

2.4.4 Résultats

Pour améliorer le passage à l'échelle de notre application, nous cherchons à obtenir le meilleur compromis possible entre l'allocation agressive des ressources mémoire des machines et les performances de notre application. Pour cela, nous étudions le comportement des deux politiques de contrôle du flot de soumission de tâches proposées précédemment afin de déterminer celle qui se prête le mieux à notre objectif.

Contrôle par le nombre de tâches

La Figure 2.8 montre les performances (haut) et l'empreinte mémoire (bas) de la configuration n°1 (cf. Table 2.3) de notre application sur 144 nœuds de la machine TERA-100 Hybrid (cf. Table 2.1) en fonction de la valeur du seuil de blocage de la soumission de tâches pour cette politique. Le seuil de déblocage de la soumission des tâches est fixé à 80% du seuil de blocage afin de limiter le coût de blocage et de déblocage de la soumission des tâches. Ces graphes montrent que la valeur de ce seuil de blocage influence grandement les performances et l'empreinte mémoire de notre application. Si le seuil est bas, l'empreinte mémoire est faible car le cache d'allocations est petit et les données sont très réutilisées, mais il n'y a pas assez de tâches soumises pour permettre de nourrir efficacement les unités de calcul et recouvrir les temps de communication par du calcul. Si le seuil est haut, les performances sont bonnes grâce au bon recouvrement des communications par du calcul, mais l'empreinte mémoire augmente car le cache d'allocations grossit pour stocker simultanément les données de toutes les tâches soumises. Si l'utilisateur veut saturer la mémoire des machines qu'elle utilise, il faut adapter la valeur du seuil pour éviter de déborder de la mémoire. Comme la granularité des tâches influe sur la taille des données présentes dans le cache d'allocation, le choix de la valeur du seuil doit donc également dépendre du paramétrage effectué par l'application : dans notre cas, la taille de bloc utilisée et le nombre de nœuds de calcul impliqués. Bien qu'il soit possible d'obtenir un bon compromis entre performances et encombrement mémoire pour notre application avec cette politique, sa dépendance au paramétrage de l'utilisateur complique l'automatisation du choix du seuil.

Contrôle par la mémoire

La Figure 2.9 montre l'évolution de l'encombrement mémoire du processus le plus chargé en mémoire sans (haut) et avec (bas) le contrôle du flot de soumission de tâches par la mémoire durant la factorisation d'une matrice de 205 000 inconnues. Ce test utilise la configuration n°2 (cf. Table 2.3) de notre application, et a été exécuté sur 16 nœuds de la machine TERA-100 Parallele (cf. Table 2.1) avec 4 processus MPI par nœud de calcul (un par nœud NUMA). Les seuils de blocage et de déblocage de la politique de contrôle

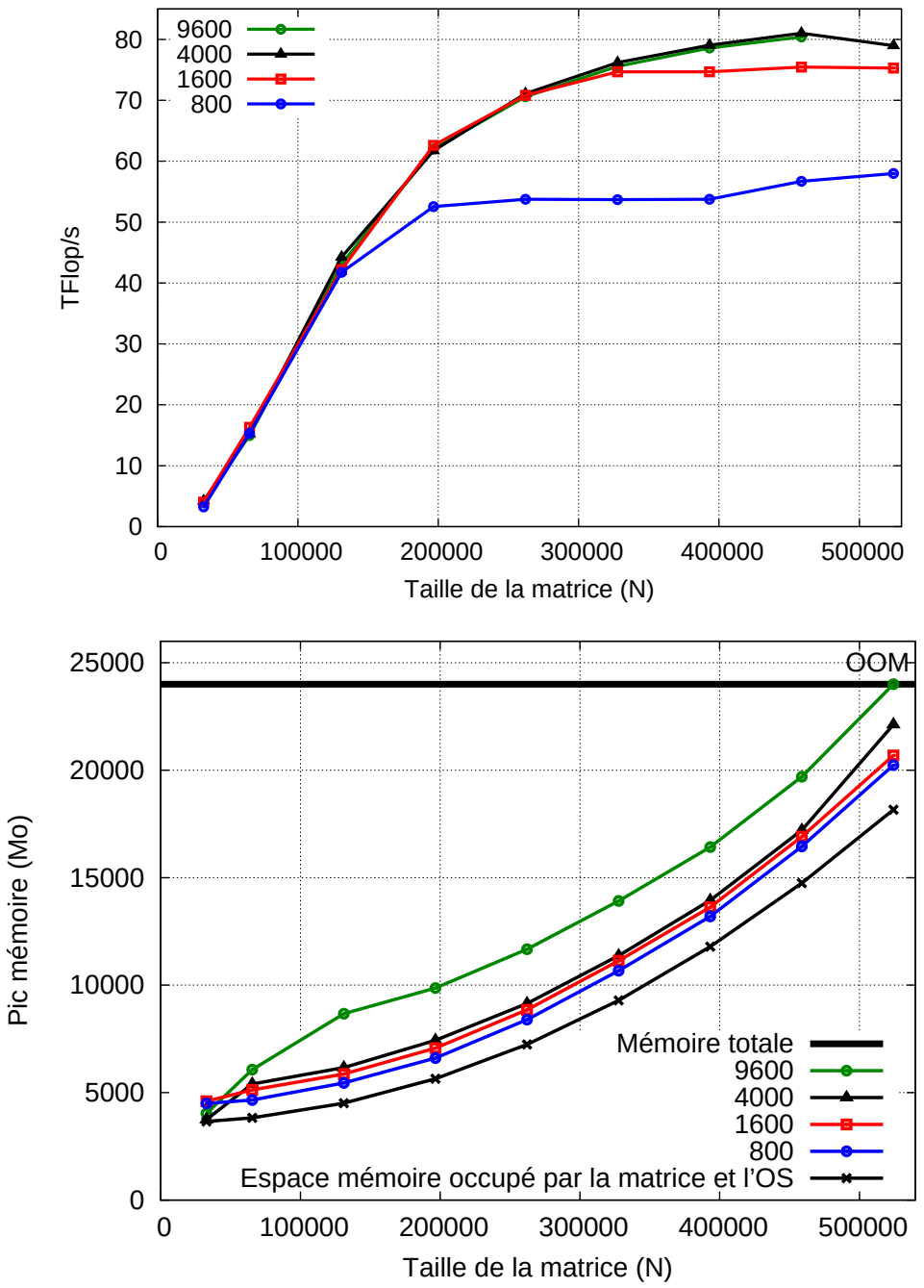


FIGURE 2.8 – Performances (haut) et empreinte mémoire (bas) pour divers seuils de blocage de la soumission de tâches sur 144 nœuds de calcul hybrides (1152 CPUs et 288 GPUs). Le seuil de déblocage est fixé à 80% du seuil de blocage.

mémoire sont définis respectivement à 28 et 24 Go.

Si la soumission des tâches n'est pas contrôlée (haut), la mémoire virtuelle

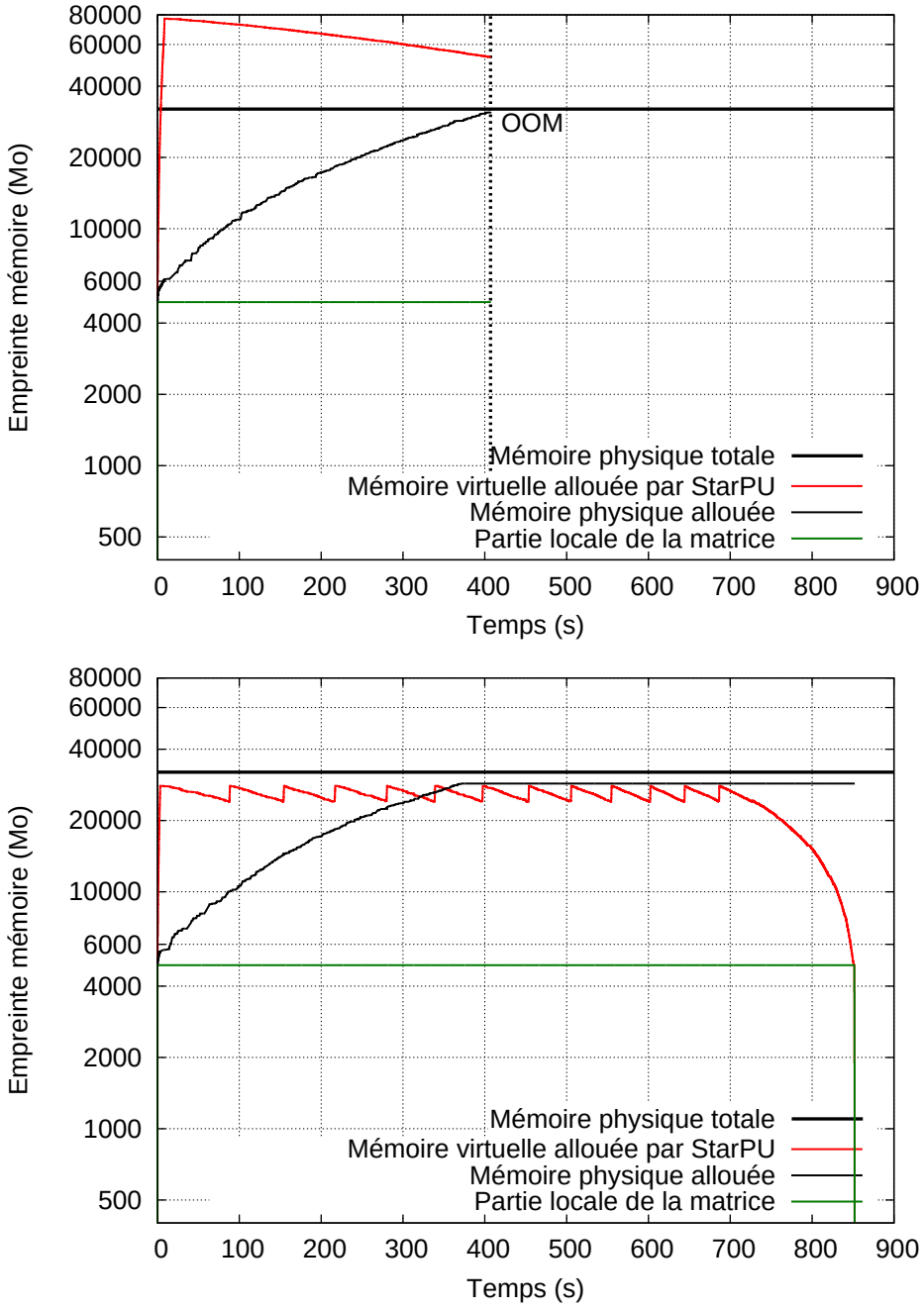


FIGURE 2.9 – Empreinte mémoire du processus le plus chargé en mémoire sans (haut) et avec (bas) contrôle mémoire. Les seuils de blocage et de déblocage avec contrôle mémoire sont définis respectivement à 28 et 24 Go.

allouée par StarPU dépasse rapidement la mémoire disponible sur la machine car tous les tampons de réception MPI sont alloués à la soumission de tâches, qui est faite trop en avance par rapport à l'exécution des tâches et ne permet

pas la réutilisation des données via le cache d'allocations. Au fur et à mesure des réceptions de tampons MPI, la quantité de mémoire physique allouée sur le nœud de calcul augmente jusqu'à provoquer l'arrêt de l'application par débordement de la mémoire (OOM : Out of Memory). Avec le contrôle mémoire (bas), la mémoire virtuelle allouée par StarPU ne dépasse pas le seuil maximal de réservation mémoire donné à StarPU car seuls les tampons de réception MPI qui ont pu être réservés par StarPU avant de bloquer la soumission de tâches sont alloués. Quand la soumission de tâches reprend, de nouvelles réceptions MPI sont postées et les tampons MPI des réceptions précédentes, désormais stockés dans le cache d'allocation, sont réutilisés. De cette manière, la mémoire physique allouée par le processus augmente jusqu'à arriver à ce seuil maximal qu'elle ne dépasse pas, car aucune allocation supplémentaire n'est effectuée par StarPU qui ne fait que réutiliser les données du cache d'allocation.

Cette politique de contrôle du flot de soumission de tâches par l'encombrement mémoire garantit l'exécution de l'application dans les limites des ressources mémoire disponibles pour StarPU, et permet d'obtenir un bon compromis entre performance et encombrement mémoire pour notre application.

2.4.5 Discussion

Nous avons vu dans la section précédente qu'il n'est pas simple de choisir la valeur du seuil de nombre de tâches qui offre le meilleur compromis entre performances et encombrement mémoire. La mémoire n'est cependant pas la seule ressource qu'il est souhaitable de contrôler dans un support d'exécution à base de tâches. Le coût des ordonnanceurs de tâches qui implémentent une heuristique ayant une complexité en calcul non polynomiale en nombre de tâches, comme l'heuristique HEFT présentée en Section 1.3.1, peut devenir non négligeable lorsque l'application soumet un très grand nombre de petites tâches au support d'exécution, comme c'est le cas pour ScalFMM [9] par exemple. Il est alors intéressant de réguler le nombre de tâches soumises au support d'exécution pour maîtriser ce surcoût en ressources de calcul. Ces deux politiques n'ont en fait pas pour objectif de répondre au même problème, et sont complémentaires. Nous avons surtout utilisé le contrôle par l'encombrement mémoire car les tâches de notre application sont grosses et nous cherchons un bon compromis entre performance et consommation mémoire, mais il est possible d'utiliser ces deux politiques en même temps dans une application et de les laisser collaborer pour contrôler le flot de soumission de tâches.

Il est important de noter que le contrôle du flot de soumission de tâches par l'encombrement mémoire ne garantit l'exécution de l'application que dans les limites des ressources mémoire *virtuelles* disponibles pour StarPU. La mémoire physique allouée par le processus peut dépasser le seuil maximum donné à StarPU car le décompte de la mémoire allouée par le gestionnaire de mémoire ne prend pas en compte la fragmentation du tas. Il est possible de contour-

ner ce problème en interdisant à l'allocateur de mémoire d'utiliser le tas pour allouer les données de l'application. Par exemple, la bibliothèque OpenMPI que nous avons utilisé pour nos expériences piège les appels à la fonction *malloc* pour utiliser son propre allocateur de mémoire qui n'alloue de la mémoire qu'avec l'appel système *mmap*. Pour prendre en compte la fragmentation du tas, une solution est de périodiquement demander des statistiques à l'allocateur de mémoire utilisé, et plus particulièrement le ratio entre la quantité de mémoire allouée et la quantité de mémoire physiquement utilisée. Il suffit alors de multiplier la mémoire décomptée par le gestionnaire de mémoire de StarPU par ce ratio pour obtenir l'encombrement mémoire réel. Les allocateurs mémoire standards ne permettent pas actuellement d'accéder à ces informations, et nous ne souhaitons pas embarquer un allocateur mémoire dans StarPU pour les raisons citées en Section 2.3.2. Nous pensons donc contribuer sur ce point à la glibc notamment.

Nous avons présenté dans cette section une méthode de *prévention* des interblocages lié à l'accès simultané à des ressources concurrentes qui fonctionne pour notre application. Cette méthode ne fonctionne cependant pas pour toutes les applications de simulation scientifique. Beaucoup d'applications découpent leur simulation en trois phases : une phase d'assemblage des données où toutes les données sont allouées puis initialisées, une phase de calcul sur ces données, puis une phase de nettoyage où toutes les données sont déallouées. Si la taille de l'ensemble des données assemblées dépasse la mémoire totale de la machine, notre méthode va bloquer la soumission de tâches pendant l'assemblage des données ; les tâches de calcul et de nettoyage ne seront jamais soumises et l'application ne va pas terminer. Une solution applicative à ce problème, notamment utilisée par QR_Mumps [10], consiste à surcontraindre l'ordre de soumission des tâches grâce aux informations connues par l'application en ajoutant des dépendances supplémentaires au graphe de tâches pour exécuter les tâches d'assemblage, de calcul et de nettoyage dans un ordre qui permet d'éviter de déborder de la mémoire. Il est possible d'implémenter une solution transparente pour le programmeur d'applications dans le support d'exécution, mais il est pour cela nécessaire de passer du modèle de *prévention* des interblocages au modèle d'*évitement* des interblocages. Au lieu de bloquer la soumission de tâches depuis l'application et d'allouer les données à la soumission des tâches, il est possible d'ajouter comme critère pour rendre une tâche prête que la quantité de mémoire nécessaire pour allouer toutes les données de la tâche doit avoir été réservée auprès du gestionnaire de mémoire de StarPU, et reporter les allocations de mémoire à ce moment là. Le gestionnaire de mémoire peut alors implémenter l'algorithme du banquier proposé par [36] pour choisir l'ordre d'allocation des ressources aux tâches afin de respecter la consigne mémoire donnée par l'utilisateur. Cette méthode permet de soumettre l'intégralité des tâches de l'application (assemblage, calcul et nettoyage) et de faire attendre les tâches d'assemblage qui ne peuvent pas réserver de la mé-

moire auprès du gestionnaire de mémoire tout en laissant les tâches de calcul et de nettoyage s'exécuter sur les données qui ont pu être allouées, ce qui permet à l'application de progresser.

2.5 Évaluation générale du passage à l'échelle

Maintenant que nous avons détaillé les différents points bloquants pour le passage à l'échelle de StarPU et comment nous les avons résolus, nous présentons dans cette section les performances de notre solveur d'algèbre linéaire dense Chameleon et nous les comparons avec celles des références de l'état de l'art ScaLAPACK et DPLASMA.

2.5.1 Contexte expérimental

La bibliothèque DPLASMA, développée par l'UTK, est l'implémentation de référence de l'algèbre linéaire dense sur support d'exécution à base de tâches. Elle se base sur le support d'exécution PaRSEC qui implémente le modèle PTG, là où Chameleon se base sur le support d'exécution StarPU qui implémente le modèle STF. La bibliothèque ScaLAPACK, développée par l'UTK également, est l'implémentation MPI de référence pour l'algèbre linéaire dense sur architecture distribuée. Elle travaille uniquement sur des matrices découpées en panneaux, là où DPLASMA et Chameleon travaillent sur des matrices découpées en blocs. L'ordonnancement de ScaLAPACK est purement statique, là où DPLASMA et Chameleon reposent sur des support d'exécution pour effectuer l'ordonnancement dynamique des tâches. Les trois bibliothèques utilisent une distribution statique 2D-bloc cyclique des données sur les nœuds de calcul MPI. Les expériences présentées dans cette section ont été réalisées avec la configuration n°1 (cf. Table 2.3) de notre application sur 144 nœuds de la machine TERA-100 Hybrid (cf. Table 2.1). Deux configurations des machines sont comparées : la configuration *homogène*, où seuls les CPUs sont utilisés pour faire le calcul, et la configuration *hétérogène* où toutes les unités de calcul (GPUs inclus) sont utilisées pour faire le calcul.

Le paramétrage de la factorisation de Cholesky utilisée et des supports d'exécution (pour les bibliothèques qui en utilisent) en fonction de chaque configuration de machines est présenté dans les Tables 2.4 et 2.5 respectivement. Ce paramétrage a été choisi pour obtenir les meilleures performances possibles de chaque bibliothèque d'algèbre linéaire dense sur les machines cibles. PaRSEC et StarPU utilisent leurs implémentations respectives des ordonnanceurs hiérarchiques avec priorités, appelées Priority Based Queue (*PBQ*) et *prio* respectivement. Dans la configuration hétérogène, StarPU utilise l'ordonnanceur *dmdas*, une variante dynamique de l'algorithme HEFT, et PaRSEC utilise une extension de *PBQ* qui permet de déporter les calculs sur

Table 2.4 – Réglage des paramètres pour chaque bibliothèque d'algèbre linéaire dense et pour chaque configuration de machine. La taille de bloc représente la taille de tuile pour Chameleon et DPLASMA, et la largeur du panneau pour ScaLAPACK.

Modèle/Bibliothèque	Homogène		Hétérogène	
	Bloc	Ordonnanceur	Bloc	Ordonnanceur
STF/Chameleon	320	*prio*	512	*dmdas*
PTG/DPLASMA	320	*PBQ*	320	*PBQ*
MPI/ScaLAPACK	64	statique	-	-

Table 2.5 – Réglages des implémentations pour chaque support d'exécution.

Modèle/Bibliothèque (support d'exécution)	Noyaux ordonnançables sur GPUs	GPU multi-streaming	Politique de comm. MPI	Punaisage mémoire OpenMPI
STF/Chameleon (StarPU)	Tous	Non	Point-à-point	Oui
PTG/DPLASMA (PaRSEC)	GEMM uniquement	Oui	Collectives	Non

les GPUs de manière gloutonne. De plus, la bibliothèque DPLASMA utilise une taille de tuilage plus petite (320) que la bibliothèque Chameleon (512) car le support d'exécution PaRSEC utilise le multi-streaming GPU pour exécuter plusieurs noyaux de calcul sur un seul GPU de manière concurrente. StarPU, quant à lui, est capable exécuter tous les noyaux de calcul de la factorisation de Cholesky sur les GPUs. PaRSEC, par contre, choisit de n'exécuter que les noyaux de calcul GEMM sur les GPUs, car ce sont les noyaux de calcul les plus intensifs en termes de calcul. Pour les communications entre nœuds MPI, StarPU est capable de tirer parti du punaisage mémoire OpenMPI pour accélérer ses communications point-à-point là où PaRSEC perd en performances avec cette optimisation. À l'inverse, PaRSEC est capable d'extraire les schémas de diffusion de la factorisation de Cholesky grâce à sa représentation algébrique du graphe de tâches (PTG) et d'effectuer des communications collectives, là où StarPU n'effectue que des communications point-à-point.

2.5.2 Résultats et comparaison avec l'état de l'art

La Figure 2.10 montre les performances des différentes bibliothèques d'algèbre linéaire dense distribuée dans les configurations *homogènes* (courbes en

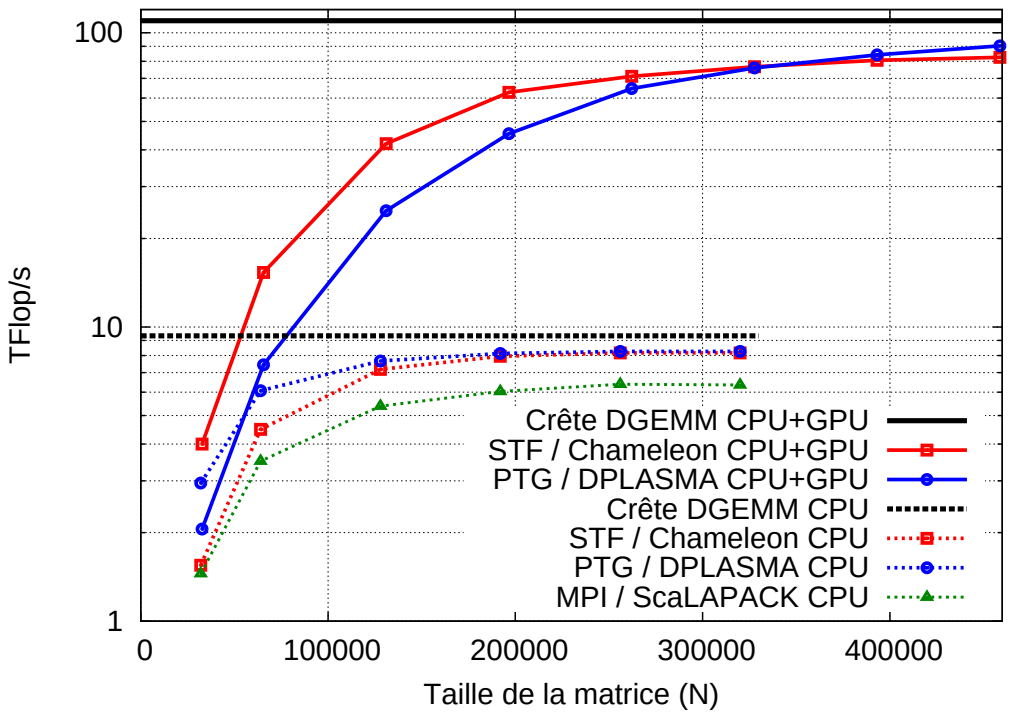


FIGURE 2.10 – Performances des factorisations de Cholesky de ScaLAPACK, DPLASMA et Chameleon sur 144 nœuds de calcul (1152 cœurs CPU et 288 GPUs au total). L'axe des Y est en échelle logarithmique.

trait pointillé) et *hétérogènes* (courbes en trait plein).

Nous observons sur cette figure que le modèle STF est compétitif avec les modèles PTG et MPI. De plus, les principales différences de performance entre Chameleon (dont le support d'exécution implémente le modèle STF) et DPLASMA (dont le support d'exécution implémente le modèle PTG) sont principalement dues aux fonctionnalités des supports d'exécution sous-jacents (StarPU et PaRSEC, respectivement), mais pas aux modèles de programmation utilisés.

Configuration *homogène*

Chameleon et DPLASMA surpassent ScaLAPACK car ces bibliothèques sont capables d'exploiter pleinement le découpage en tuiles de la factorisation de Cholesky, là où ScaLAPACK ne travaille que sur des panneaux de la matrice. Chameleon et DPLASMA exhibent les mêmes performances asymptotiques, mais DPLASMA est plus efficace pour les petites matrices. Ceci s'explique par le fait que StarPU est optimisé pour les architectures hétérogènes, là où PaRSEC a originalement été développé pour exploiter des architectures homogènes. Dans ce cas particulier, il faudrait que l'ordonnanceur *prio* de StarPU

puisse gérer l'ordonnancement des tâches en fonction de la localité des données en plus des priorités, comme le fait l'ordonnanceur *PBQ* de PaRSEC.

Configuration *hétérogène*
Chameleon surpasse DPLASMA pour les petites matrices (jusqu'à 320 000 inconnues) pour deux raisons : d'abord, le punaisage mémoire OpenMPI accélère grandement les communications point-à-point de StarPU, ce qui permet de débloquer plus rapidement du parallélisme pour nourrir les GPUs avec du calcul. Ensuite, l'heuristique d'ordonnancement HEFT de StarPU permet de prendre de meilleures décisions d'ordonnancement au démarrage de la factorisation, là où la politique gloutonne de l'ordonnanceur de PaRSEC oblige la déportation des premiers GEMM sur les GPUs en dépit du coût de transfert des données. Pour des matrices plus grandes que 320 000 inconnues, StarPU comme DPLASMA affichent des performances proches de la borne théorique du GEMM (calculée comme la somme de la performance du GEMM sur CPU et du GEMM sur GPU multiplié par le nombre total d'unités de calcul de chaque type). DPLASMA est légèrement plus performant que Chameleon pour les grandes matrices grâce au multi-streaming GPU supporté par PaRSEC qui permet d'obtenir un meilleur compromis entre parallélisme et granularité, mais également grâce aux communications collectives MPI qui permettent de réduire considérablement la latence des diffusions de la factorisation de Cholesky (par exemple, la diffusion des blocs diagonaux aux panneaux) par rapport aux communications purement point-à-point de StarPU-MPI.

2.5.3 Discussion

Pour résoudre le problème de latence excessive des communications point-à-point lors des diffusions, il est nécessaire que StarPU-MPI permette de faire des communications collectives. Par contre, il ne faut pas que cela remette en cause le modèle d'exécution distribué et les choix d'implémentations effectués. Une solution simple consiste à demander à l'application de donner explicitement les diffusions de son application à StarPU-MPI, mais ce n'est pas satisfaisant car cela implique de changer le code de l'application. Automatiser les diffusions est plus complexe car deux problèmes se posent : celui de la détection distribuée de la diffusion et celui de la diffusion sans synchronisation globale.

Pour la détection, nous ne voulons pas que tous les nœuds de calcul aient à détecter le schéma de diffusion et à soumettre une communication collective MPI car cela remettrait en cause l'élagage du graphe de tâches, alors que seul le nœud initiateur a effectivement besoin de détecter la diffusion. Une méthode automatique et distribuée pour détecter les diffusions consiste à retarder la soumission des requêtes de communications MPI jusqu'à ce que toutes les tâches d'une même *génération* du graphe de tâches soient soumises. Lorsqu'une génération complète a été soumise, il est possible de détecter la présence d'un

schéma de communication collective entre cette génération et la précédente. Par contre, cette détection nécessite que le graphe de tâches soit soumis *en largeur d'abord*, là où beaucoup d'applications soumettent le graphe de tâches *en profondeur d'abord*.

Une fois le schéma de diffusion détecté par le nœud initiateur, il faut que ce nœud puisse informer les nœuds récepteurs de la nature du schéma de diffusion qui a été choisi (chaîne, arbre binaire ou binomial, etc.). Une manière d'implémenter cette technique dans StarPU-MPI est de permettre aux enveloppes envoyées avant chaque communication de données de contenir une information de diffusion. Cette information contiendrait la vue locale du schéma de diffusion distribué pour le nœud récepteur, et lui permet de savoir quel nœud va lui envoyer la donnée de la diffusion (pas nécessairement le nœud initiateur) et à quels autres nœuds il doit la propager, sans jamais avoir connaissance du schéma global de diffusion.

2.6 Intégration dans la chaîne de simulation logicielle 3D

Pour utiliser le solveur Chameleon dans le code de simulation du CEA, il est nécessaire que les parties de ce code qui effectuant la résolution du système $Ax = B$ appellent à la place des routines de la bibliothèque Chameleon. La chaîne logicielle 3D étant structurée de sorte à séparer clairement le solveur numérique des parties métier de la simulation, nous nous concentrons sur la substitution des deux étapes de la résolution du système, à savoir la factorisation de la matrice et la descente-remontée, par leurs équivalents dans Chameleon.

Factorisation de la matrice A

La matrice A à traiter étant complexe symétrique non hermitienne, nous avons d'abord développé une adaptation du noyau de calcul POTRF pour ce type de matrices car ce calcul n'existe actuellement pas dans le standard LAPACK. Ce noyau, que nous avons nommé `ZSYTRF_LLt`, a été ajouté à la bibliothèque Chameleon et est disponible pour tous les utilisateurs. Nous avons ensuite substitué à la fonction d'appel de la factorisation de matrice du code existant une fonction qui effectue ces trois opérations : donner à Chameleon l'accès à la matrice à factoriser A précédemment assemblée par la chaîne logicielle à travers l'initialisation d'un descripteur de matrice de Chameleon ; donner à Chameleon la fonction de répartition des données sur les nœuds de calcul ; et enfin, appeler la routine de calcul de la factorisation de Cholesky sur matrice complexe symétrique non hermitienne en double précision : `PZSYTRF_LLt`.

Résolution du système par descente-remontée
La méthode d'implémentation est identique à celle de la factorisation de matrice, c'est-à-dire substituer l'appel de la fonction de descente-remontée du code existant par une fonction qui effectue les trois opérations suivantes : déclarer la matrice factorisée A et la matrice de second membres B (dans notre application, un panneau) à Chameleon via des descripteurs de matrice ; donner à Chameleon la fonction de répartition des données sur les nœuds de calcul ; et enfin, appeler la routine de calcul de la descente-remontée depuis une matrice symétrique non hermitienne sur un panneau d'éléments complexes double précision : ZSYTRS. L'ajout notable de la descente-remontée par rapport à la factorisation de matrice est que nous avons forcé l'exécution des tâches sur les nœuds qui possèdent les blocs diagonaux de la matrice. En effet, la descente-remontée du code existant effectue les calculs sur ces nœuds, et la suite de la chaîne logicielle attend que les données produites par la descente-remontée soient présentes sur ces nœuds pour continuer le calcul. Nous avons donc calqué le comportement de notre descente-remontée avec celle du code existant afin que ces données se trouvent à l'endroit attendu par la suite de la chaîne logicielle sans nécessiter de nouvelles communications de données.

Discussion
Nous avons constaté durant cette intégration que la bibliothèque Chameleon facilite considérablement l'intégration de solveurs d'algèbre linéaire sur supports d'exécution dans des grands codes de simulation tel que celui du CEA. En effet, l'abstraction de la machine offerte par Chameleon est tel que l'utilisateur n'a pas à se préoccuper du pilotage du support d'exécution sous-jacent pour obtenir des performances. De plus, les performances du solveur Chameleon sont comparables au solveur numérique développé par le CEA pour ses besoins. Cette intégration réussie soutient donc l'idée que les supports d'exécution facilitent la programmation et l'utilisation des machines parallèles pour les utilisateurs.

Synthèse

Comme précisé dans la Section 1.1.2 du Chapitre 1, il existe deux axes de réponse pour améliorer le temps de restitution des simulations numériques. Dans ce chapitre, nous avons considéré les évolutions des techniques informatiques, notamment en termes de parallélisme et d'exploitation efficace des machines de calcul, grâce à l'intégration et à l'utilisation d'un solveur d'algèbre linéaire dense porté sur support d'exécution à base de tâches dans la chaîne de simulation logicielle 3D du CEA. Nous avons montré qu'il est possible avec un support d'exécution à base de tâches soumises séquentiellement d'obtenir un passage à l'échelle de la factorisation de Cholesky dense tuilée de

la bibliothèque Chameleon, qui utilise le support d'exécution StarPU, comparable aux références de l'état de l'art et au solveur développé au CEA sur 144 nœuds de la machine TERA-100 Hybrid du CEA (soit 1152 cœurs CPU et 288 accélérateurs GPU).

Pour cela, nous avons proposé les contributions suivantes : un élagage intelligent et optimal du graphe de tâches en fonction de la projection des données sur les nœuds de calcul et des données accédées par les tâches qui permet de faire passer à l'échelle le modèle de réplication du graphe de tâches sur les nœuds de calcul avec le modèle STF sur 144 nœuds de calcul du CEA, et théoriquement au-delà du million de nœuds de calcul ; un contrôle des allocations de mémoire de StarPU grâce à un cache d'allocations pour contourner le problème du coût prohibitif des déallocations de mémoire des allocateurs standards lorsqu'ils utilisent l'appel système *mmap*, et de la fragmentation importante du tas lorsqu'ils utilisent l'appel système *sbrk* ; finalement, un mécanisme de contrôle du flot de soumission de tâches pour limiter les ressources consommées par le support d'exécution, et deux politiques pour contrôler le flot de soumission de tâches en fonction du nombre de tâches soumises au support d'exécution et de l'encombrement mémoire des allocations enregistrées auprès du support d'exécution.

Pour répondre au besoin de simulations 3D de plus en plus fines avec le code du CEA, il est également nécessaire d'étudier les évolutions des techniques numériques. Cette thèse n'a pas pour objet d'explorer cet axe, mais d'adapter les techniques informatiques que nous proposons pour permettre le passage à l'échelle des applications utilisant ces nouveaux algorithmes. Dans le cadre de la chaîne de simulation logicielle 3D du CEA, l'évolution des techniques algorithmiques s'est traduite par l'utilisation de techniques de compression de type Block Low-Rank (BLR) sur les matrices à traiter. Les évolutions associées des techniques informatiques pour permettre le passage à l'échelle du solveur BLR font l'objet du chapitre suivant.

Chapitre 3

Adapter le contrôle de l'encombrement mémoire aux données de taille évolutive

"In medio stat virtus, non non c'est tout à fait exact la vérité est dans le juste milieu. Par contre ça n'a rien à voir avec la conversation."

– Loth, *Kaamelott Livre V, Les Nouveaux Clans*

Résumé du chapitre **70**

3.1 Compression de matrices issues d'équations intégrales **71**

3.1.1 Matrices de rang faible 71

3.1.2 Solveur linéaire avec compression 72

3.2 Contribution : implémentation d'un solveur pour matrices compressées avec StarPU **78**

3.2.1 Une représentation StarPU des matrices compressées 78

3.2.2 Intégration des noyaux de calcul pour matrices compressées dans Chameleon 79

3.3 Contribution : gestion des données de taille variable avec StarPU **82**

3.3.1 Estimer l'évolution de la taille des données 82

3.3.2 Mise en œuvre dans le solveur pour matrices compressées 83

3.4 Précision des estimateurs et performances **83**

3.4.1 Cas du solveur pour matrices compressées 84

3.5 Discussion **86**

Résumé du chapitre

Pour aller plus loin dans le passage à l'échelle du solveur linéaire, le CEA propose de faire évoluer les méthodes numériques utilisées par le solveur pour tirer parti de techniques de compression de matrices issues d'équations intégrales, que nous présentons en Section 3.1. La difficulté de cette méthode numérique réside dans le fait que le taux de compression des données du problème change pendant les phases de calcul du solveur. Nous présentons en Section 3.2 une nouvelle interface de données StarPU permettant de décrire des matrices compressibles. Nous montrons que cette interface permet d'intégrer les noyaux de calcul spécifiques aux matrices compressées développés au CEA dans Chameleon. Nous montrons également que cette intégration facilite la transition du solveur pour matrices denses vers ce nouveau solveur pour matrices compressibles.

Toutes les contributions présentées dans le chapitre 2 sont immédiatement utilisables pour optimiser les performances de ce nouveau solveur, à l'exception notable du contrôle de l'encombrement mémoire présenté en Section 2.4. En effet, pour que StarPU puisse garantir l'exécution du programme dans les limites mémoire définies par l'utilisateur, il doit connaître la taille des données lors de la soumission des tâches pour pouvoir refuser d'accepter de nouvelles tâches en soumission lorsque sa vue de la mémoire est pleine. Afin que StarPU puisse anticiper l'évolution de la taille des données dès la soumission des tâches, nous proposons en Section 3.3 que StarPU prenne en compte les variations de la taille des données grâce à des estimateurs spécifiés par l'application. Nous montrons en Section 3.4 que si ces estimateurs sont peu précis, la nécessité pour StarPU de garantir l'exécution correcte du programme peut restreindre excessivement le parallélisme disponible. Nous présentons ensuite une solution pragmatique basée sur la connaissance empirique de l'évolution des données de ce solveur qui permet de débloquer du parallélisme, mais au prix de concessions sur la garantie d'exécution du programme dans la limite de la mémoire disponible pour StarPU.

3.1 Compression de matrices issues d'équations intégrales

Dans cette section, nous montrons qu'il est possible de compresser certaines parties des matrices issues d'équations intégrales pour réduire considérablement le coût en calcul et en mémoire de la résolution du système linéaire, tout en contrôlant l'erreur numérique introduite par cette compression. Nous présentons les évolutions effectuées par le CEA dans le code de simulation pour tirer parti de la compression de matrices, et nous montrons sur un grand cas de calcul l'intérêt de la méthode pour les calculs 3D effectués par le code du CEA.

3.1.1 Matrices de rang faible

Il est possible de définir le rang r d'une matrice A de $\mathbb{C}^{m\times n}$ comme la taille de son image :

$$r = dim(Im(A)) \text{ , où } Im(A) = \{y \in \mathbb{C}^m, \exists x \in \mathbb{C}^n, y = Ax\} \tag{3.1}$$

Afin de calculer en pratique le rang d'une matrice on utilise la décomposition en valeurs singulières (SVD [1]). Cette décomposition peut être réalisée à l'aide du noyau `ZGESVD` défini dans la bibliothèque BLAS.

La SVD permet de décomposer A en $\mathcal{U}\Sigma\mathcal{V}^H$. Σ est une matrice diagonale composée des valeurs singulières de la matrice A. $\mathcal{U}$ et $\mathcal{V}$ sont deux matrices unitaires de taille $m \times p$ et $p \times n$, avec $p = \min(m, n)$.

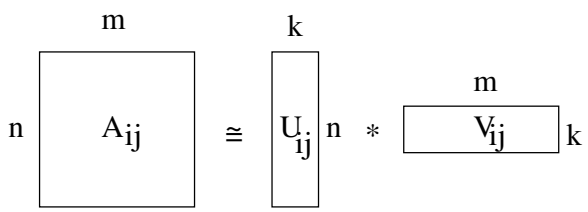


FIGURE 3.1 – Une tuile A_{ij} de rang faible admet une représentation approchée $U_{ij}V_{ij}^t$.

Une fois les valeurs singulières obtenues, on les classe dans l'ordre décroissant. Si la valeur normalisée par rapport à la première valeur singulière de la valeur singulière numéro k est plus petite qu'un certain seuil ε_{svd}, on considère alors que k est le rang de la matrice.

La méthode *Adaptive Cross Approximation* (ACA) avec pivot partiel initialement développée par Mario Bebendorf dans [20] est une méthode algébrique qui permet d'approcher une matrice régulière de rang faible (dont le rang est

1. *Singular Value Decomposition*

```
rank = size(S)
do k=1, size(S)
  if (S(k)/S(1) < epsilon_svd) then
    rank = k
    exit
  end if
end do
```

Listing 3.1 – Recherche du rang par la SVD. S contient toutes les valeurs singulières classées par ordre décroissant.

inférieur à sa taille) en n'utilisant que quelques éléments de la matrice originale. Par conséquent, comparée à d'autres techniques de compression comme la SVD, la méthode ACA est beaucoup moins coûteuse.

À l'aide de la méthode ACA [73] on décompose une matrice régulière A (dans $\mathbb{C}^{m\times n}$) en $A = U \times V$ avec U dans $\mathbb{C}^{m\times r}$ et V dans $\mathbb{C}^{r\times n}$. Cette décomposition est soumise à l'erreur ε ($\|.\|$ désignant la norme de Frobenius sur les matrices, sauf mention contraire) :

$$\varepsilon = \frac{\|A - U \times V\|}{\|A\|} = \frac{\|A - \tilde{A}\|}{\|A\|} = \frac{\|R\|}{\|A\|} \tag{3.2}$$

On définit $\tilde{A} = UV$ comme étant la matrice approchant A et $R = A - \tilde{A}$ la matrice d'erreur associée.

La décomposition réalisée permet de limiter le stockage d'une matrice de taille $m \times n$ à deux matrices de tailles $m \times r$ et $r \times n$, soit un coût mémoire en $O(r(m+n))$. Si $r << \min(m, n)$, on diminue alors grandement la place mémoire occupée pour le stockage de A. Pour exemple, la plupart des matrices A traitées par le code de simulation 3D du CEA ont un taux de compression supérieur à 90%.

Par la suite on réutilise le *rank fraction* (RF) défini dans [63] afin de mesurer l'impact de cette décomposition. Ce « rapport de compression » décrit l'espace occupé par la décomposition, comparé à celui qu'occuperait la matrice pleine.

$$RF = \frac{r(m+n)}{m \times n} \tag{3.3}$$

3.1.2 Solveur linéaire avec compression

Recherche de tuiles de rang faible

Il n'est pas possible d'appliquer la méthode ACA sur l'ensemble de la matrice d'impédance A car celle-ci n'est pas régulière du fait de la singularité du noyau

de Green. Découper la matrice A en tuiles a donc un avantage dans cette méthode car cela permet d'isoler les tuiles diagonales qui ne sont pas régulières.

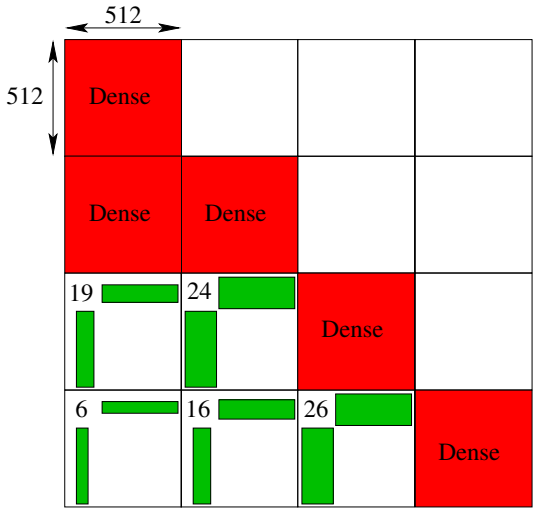


FIGURE 3.2 – Exemple de matrice tuilée compressible. Les nombres correspondent au rang des tuiles compressées.

Nous cherchons à appliquer la méthode ACA sur des tuiles de rang faible (tel que celui-ci soit largement inférieur à leur taille), afin de diminuer à la fois le stockage des tuiles et le coût de l'algorithme ACA.

Or, un terme A_{ij} de la matrice d'impédance A représente l'interaction entre le ddl i et le ddl j. Comme ce terme dépend principalement de la fonction de Green, il varie en $1/d$, avec d la distance entre le ddl i et le ddl j. Le regroupement spatial des ddls en domaines permet alors de créer des tuiles qui représentent l'interaction de l'ensemble des ddls d'un domaine sur un autre. Les interactions entre deux domaines éloignés sont donc représentées par des tuiles de rang faible dans la matrice A, comme le montre la FIGURE 3.2. À l'inverse, les tuiles diagonales ont un rang proche de leur taille car elles représentent les interactions d'un domaine sur lui-même : pour ces tuiles, le noyau de Green est irrégulier. Il est donc trop coûteux d'appliquer la méthode ACA dessus. En fonction de la géométrie de l'objet, il est possible que certaines tuiles extra-diagonales représentent également des interactions entre des domaines proches spatialement, et ne soient donc pas compressibles.

Pour résumer, le regroupement spatial des ddls permet d'obtenir des tuiles de rang faible en dehors de la diagonale. Ces tuiles peuvent alors être compressées par la méthode ACA.

Partitionnement avec un pavage : *cobblestone*

Pour le groupement spatial des inconnues, Shaeffer propose dans [63] une mé-

thode de partitionnement basée sur un pavage : la technique dite de *cobblestone*. Dans ce découpage il est possible de fixer le nombre de ddls que l'on veut placer par domaine, comme le montre la FIGURE 3.3.

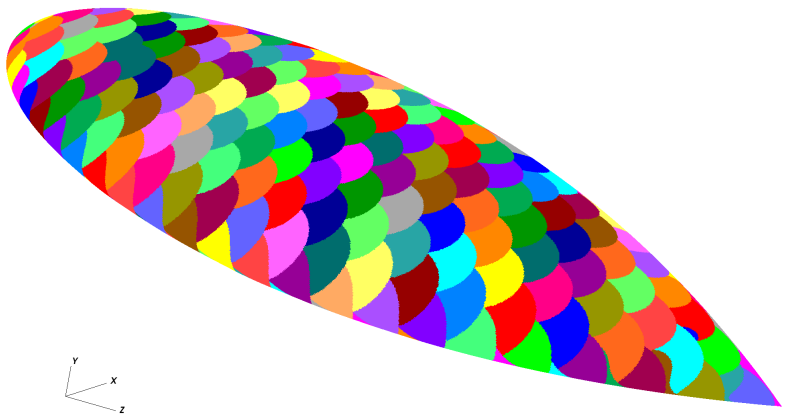


FIGURE 3.3 – Partitionnement d'une amande du cas test académique NASA Almond de 929 280 ddls avec une taille de domaine de 3 000 ddls. On observe bien la structure de type pavage sur la face de l'amande.

De part la construction du partitionnement, on remarque dans la FIGURE 3.4 que plus on s'éloigne de la diagonale dans la matrice plus les domaines mis en jeu sont éloignés.

Afin de s'en assurer dans le code, on considère que deux domaines i et j sont éloignés lorsqu'ils vérifient la relation suivante :

$$\frac{distance(i,j)}{taille\ du\ domaine} \geq \delta \tag{3.4}$$

où δ est un paramètre d'entrée que l'on choisit entre 0 et 1. La distance entre les deux domaines est calculée comme étant la distance entre leurs centres de gravité.

Factorisation de Cholesky tuilée avec la compression ACA

Lorsqu'on souhaite intégrer la compression ACA établie lors de l'assemblage dans la factorisation de Cholesky, il est nécessaire de tenir compte de la structure des tuiles compressées dans les calculs de l'algorithme :

1. Les tuiles diagonales ne sont jamais compressées, il n'y a pas de modification pour leur mise à jour. On fait toujours appel à `ZSYTRF_LLt`.
2. Lors de la mise à jour du panel, si la tuile A_{ik} est compressée, on effectue la mise à jour suivante :

$$U_{ik}^{new} \times V_{ik}^{new} = U_{ik}^{old} \times V_{ik}^{old} \times A_{kk}^{-1} \tag{3.5}$$

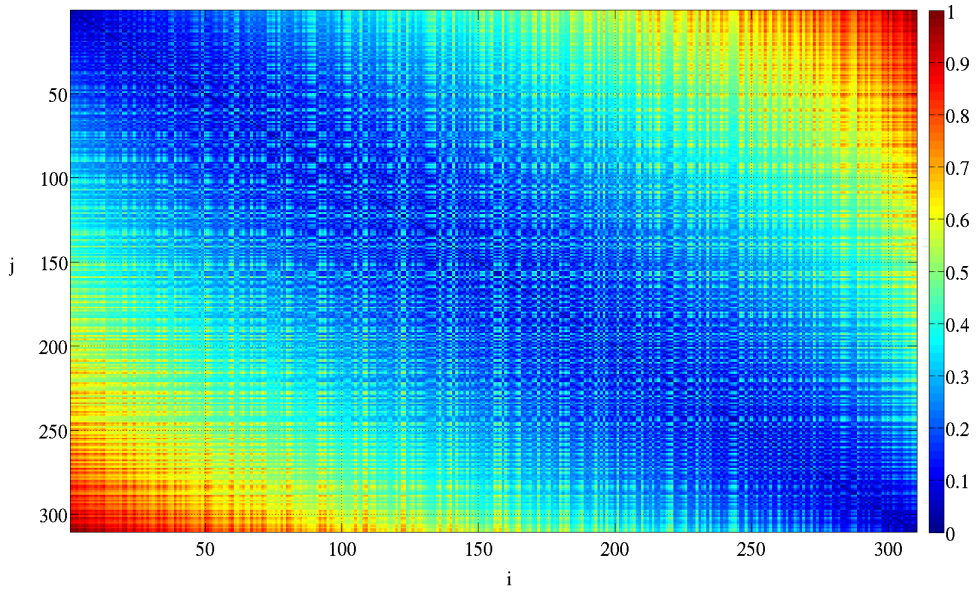


FIGURE 3.4 – Représentation de la distance entre le domaine i et le domaine j, normalisée par rapport à la taille du domaine : $d = \frac{distance(i,j)}{taille\ du\ domaine}$ dans le cas du partitionnement de l'amande présenté FIGURE 3.3.

Cette mise à jour est réalisée en écrivant :

$$\begin{cases} U_{ik}^{new} & = U_{ik}^{old} \\ V_{ik}^{new} & = V_{ik}^{old} \times A_{kk}^{-1} \end{cases} \tag{3.6}$$

3. Pour la mise à jour du reste de la matrice, deux familles de cas sont à considérer :
   (a) La tuile A_{ij} n'est pas compressée, quatre sous-cas apparaissent en fonction de la compression de A_{ik} et A_{jk}. Le produit $A_{ik} \times A_{jk}^t$ est effectué de manière à limiter le nombre d'opérations et la taille des tampons temporaires nécessaires aux calculs (puisqu'à cette étape on ne peut pas écraser les tuiles A_{ik} ou A_{jk}) :
      i. A_{ik} et A_{jk} non compressibles : on effectue classiquement le produit $A_{ik} \times A_{jk}^t$
      ii. A_{ik} non compressible, A_{jk} compressible : $(A_{ik} \times V_{jk}^t) \times U_{jk}^t$
      iii. A_{ik} compressible, A_{jk} non compressible : $U_{ik} \times (V_{ik} \times A_{jk}^t)$
      iv. A_{ik} compressible, A_{jk} compressible : $U_{ik} \times (V_{ik} \times V_{jk}^t) \times U_{jk}^t$. Ce dernier produit n'a pas de sens privilégié à priori, puisqu'on ne sait pas si c'est la tuile ik ou la tuile jk qui a été la plus compressée

(b) Ou alors la tuile A_{ij} a été compressée et on souhaite effectuer la mise à jour suivante :

$$U_{ij}^{new} \times V_{ij}^{new} = U_{ij}^{old} \times V_{ij}^{old} - A_{ik} \times A_{jk}^{t} \tag{3.7}$$

dans laquelle A_{ik} et A_{jk} peuvent aussi être compressées.

Bien que dans la littérature ([18, 43]) cette étape de recompression s'effectue à l'aide d'une décomposition QR suivie d'une SVD, le CEA a choisi ici d'appliquer de nouveau l'algorithme ACA sur le terme de droite. On recherche ainsi une ligne ou une colonne de ce terme, sans le calculer explicitement, à l'aide des propriétés suivantes :

- La ligne p du produit $C \times D$ est calculé par : $C(p,:) \times D$
- Et la colonne q du produit $C \times D$ par : $C \times D(:,q)$

Ces calculs de lignes et de colonnes s'effectuent eux aussi de manière à limiter le nombre d'opérations. Ainsi, lorsqu'on cherche à obtenir la colonne q du produit $U_{ik} \times (V_{ik} \times A_{jk}^{t})$ (cas (a) iii.), on la calcule de la manière suivante :

$$\left[U_{ik} \times (V_{ik} \times A_{jk}^{t})\right](:,q) = U_{ik} \times [V_{ik} \times A_{jk}(q,:)] \tag{3.8}$$

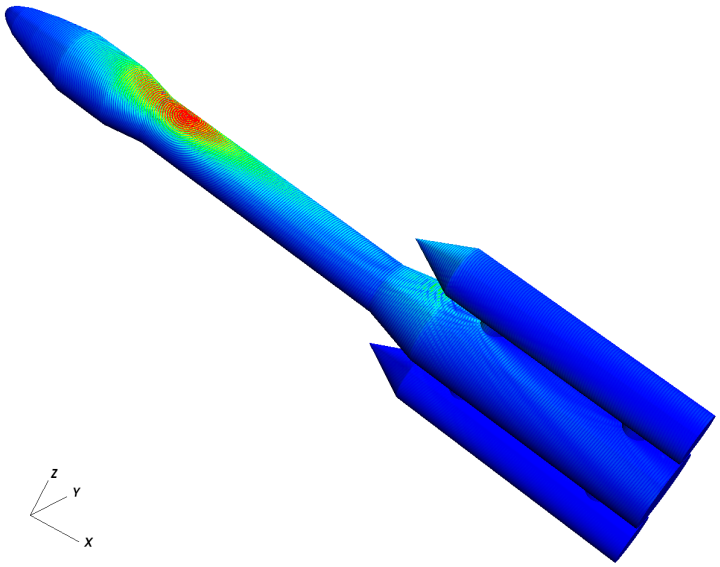
Une seconde application de l'algorithme ACA conduit à un nouveau couple U_{ij}, V_{ij} qui représente la compression de la tuile ij de la matrice L. Cette nouvelle compression ne se fait pas nécessairement avec le même rang que la compression sur A : ce rang peut augmenter ou diminuer, mais la tuile ij ne peut pas grossir au point de redevenir dense.

Intérêt de la méthode

La Figure 3.5 présente une comparaison des temps de calcul et de l'occupation mémoire des solveurs dense et avec compression du CEA sur le cas du lanceur CNES à 1 650 875 inconnues du colloque EM ISAE [5] sur la machine TERA-100 Parallele (cf. Table 2.1). L'erreur ε de la méthode ACA, telle que définie en (3.2), est fixée à 10^{-4}.

Nous remarquons tout d'abord que la méthode ACA permet de compresser la matrice à 93,6%, ce qui permet de lancer la simulation de ce type de cas sur un nombre de cœurs bien plus faible qu'avec le solveur dense. De plus, cette méthode permet de réduire considérablement le temps de calcul de ce grand cas, avec ici un gain en temps de l'ordre de 60.

Cette méthode est donc tout à fait adaptée pour simuler des grands cas 3D qui étaient auparavant inaccessibles avec un solveur dense. Nous souhaitons donc porter ce solveur pour matrices compressées sur StarPU pour profiter des bonnes propriétés de ce support d'exécution pour optimiser le calcul.



Solveur (nombre de cœurs)	Taille en mémoire	Taux de compression	Temps de factorisation (Heures CPU)
Dense (13 626)	40,6 To	0%	305 870
Compressé (1 640)	2,6 To	93,6%	5 082

FIGURE 3.5 – Comparaison des temps de calcul et de l'occupation mémoire des solveurs dense et compressé pour le cas du lanceur CNES à 1 650 875 inconnues du colloque ISAE.

Gérer des données de taille variable avec StarPU

La difficulté principale de l'implémentation de cette méthode réside dans le fait que les données peuvent changer de taille dynamiquement. Il faut donc que l'application décrive ses données de taille évolutive à StarPU pour qu'il puisse les manipuler. Nous présentons en Section 3.2 une nouvelle interface de données permettant au programme de décrire des matrices compressibles, i.e compressées ou non, à StarPU. Nous montrons ensuite comment nous avons utilisé cette interface pour intégrer les noyaux de calcul spécifiques aux matrices compressées développés par le CEA dans Chameleon. Nous détaillons en particulier le mécanisme permettant aux tâches de mise à jour de la matrice de modifier la taille des données en plus de modifier leur contenu.

Un autre problème de cette méthode est que l'utilisateur n'est pas en mesure de prédire exactement à quel point les données de la matrice A vont grossir pendant le calcul, ce qui peut provoquer des débordements de mémoire. La seule information qu'il est en mesure de fournir à StarPU est qu'une tuile

compressée ne peut pas grossir au point de redevenir dense. Nous avons vu en Section 2.4 que StarPU peut garantir que la quantité de mémoire consommée par StarPU ne dépasse pas la limite mémoire donnée par l'utilisateur grâce au contrôle mémoire. Or, pour que ce contrôle fonctionne, il est nécessaire que StarPU connaisse la taille des données à la soumission des tâches, ce qui n'est pas possible dans le cas de ce solveur. Nous montrons dans la Section 3.3 qu'anticiper la variation de la taille des données dès la soumission des tâches permet de conserver la garantie d'exécution dans les limites de la mémoire du contrôle mémoire de StarPU.

De par sa complexité, ce solveur pour matrices compressées n'a été implémenté que pour des machines homogènes CPU. Nous n'utiliserons donc pas d'accélérateurs GPUs pour nos expériences de ce chapitre.

3.2 Contribution : implémentation d'un solveur pour matrices compressées avec StarPU

3.2.1 Une représentation StarPU des matrices compressées

La première étape pour implémenter ce solveur est donc de permettre à StarPU de gérer des tuiles de matrice de taille variable. Comme décrit en Section 1.3, un programme peut décrire la structure de ses données à StarPU à travers des interfaces de données, dont StarPU se sert pour automatiser et optimiser les transferts de données entre les différentes mémoires des machines. Parmi les interfaces de données StarPU préexistantes, on peut citer les vecteurs par exemple, ou encore les matrices denses dont le solveur Chameleon se sert. Cette dernière interface ne permet de décrire que des matrices denses de taille fixe. Nous proposons donc d'étendre cette interface pour pouvoir décrire des matrices compressées.

Pour ce faire, StarPU met à disposition du programmeur une API lui permettant de décrire sa propre interface de données et de l'enregistrer auprès de StarPU. Nous nous sommes servis de cette fonctionnalité pour implémenter dans Chameleon une extension de l'interface pour matrices denses de StarPU. Cette nouvelle interface permet à l'utilisateur de décrire des matrices denses *compressibles*, c'est-à-dire des matrices qui peuvent être soit pleines, soit compressées.

Pour intégrer la connaissance du rang de la matrice compressible dans la gestion des données de l'interface, nous avons redéfini les quatre méthodes de l'interface de données présentées en Figure 3.6.

```
struct starpu_data_interface_ops {
  ssize_t allocate ();
  ssize_t free ();
  int pack_data(void **ptr, size_t *count);
  int unpack_data(void *ptr, size_t count);
  ...
};
```

FIGURE 3.6 – Méthodes de l'interface de données StarPU que nous avons redéfinies pour l'interface des matrices compressibles.

Les méthodes `allocate` et `free` sont utilisées par StarPU pour allouer et libérer une donnée. Dans ces méthodes, nous avons introduit la gestion du format compressé des données : si la tuile est dense, une seule donnée est allouée ; si la tuile est compressée, deux données sont allouées pour stocker les deux matrices U et V^t.

Dans le cadre du transfert de données entre processus, les méthodes `pack` et `unpack` sont utilisées respectivement pour compacter et dé-compacter les données dans et depuis un tampon de données contigu. Ici, nous avons ajouté dans la méthode `pack` un nouvel encodage du tampon de données afin que le récepteur puisse retrouver le format de compression des données dans la méthode `unpack`.

3.2.2 Intégration des noyaux de calcul pour matrices compressées dans Chameleon

La structure du solveur pour matrices compressées étant similaire à celle du solveur pour matrices denses, l'intégration des noyaux de calcul dans Chameleon consiste juste, structurellement, à substituer les noyaux de calcul dense par leur équivalent pour matrices compressées. La difficulté principale de cette intégration se situe dans les accès aux données. En effet, certains noyaux de calcul ont besoin de modifier directement la taille des données accédées en écriture pour refléter le changement du taux de compression de données, et donc de réallouer les données U et V^t.

```
void (*cpu_func) (void *buffers[], void *cl_arg);
```

FIGURE 3.7 – Prototype d'une fonction de codelet StarPU.

StarPU requiert que les fonctions des codelets (i.e. les tâches), présentées en Section 1.3.1, aient le prototype présenté en FIGURE 3.7. Dans ce prototype, les données accédées par la tâche sont représentées par un tableau `buffers[]`

contenant les **interfaces** des données de la tâche. Ainsi, les tâches StarPU peuvent accéder directement aux interfaces de leurs données. Ainsi, la tâche de mise à jour peut directement accéder à l'interface de la donnée produite pour modifier le rang de la matrice, et pour ré-allouer correctement les données en appelant les méthodes `allocate` et `free` de l'interface.

Intégration du solveur dans le code de simulation 3D

Le solveur étant désormais implémenté dans Chameleon et fonctionnel, nous souhaitons l'utiliser dans le code de simulation en lieu et place du solveur dense dont nous avons présenté l'intégration en Section 2.6. Pour ce faire, il est nécessaire d'adapter l'enregistrement des données du programme présenté en Section 1.3.1 afin d'utiliser l'interface de matrices compressibles. La routine d'enregistrement des données, présentée dans l'algorithme 9, a une structure similaire à son homologue pour matrices denses, mais requiert en plus que le programme spécifie le rang de la matrice compressée. Si le rang spécifié à l'enregistrement est 0, la matrice est alors considérée comme dense.

Algorithme 9 : Enregistrement des tuiles compressées de la matrice A.

```
for (m = 0 ; m < N ; m++) do
  for (n = 0 ; n < N ; n++) do
    compressed_matrix_data_register (&Ahandles[m][n],
      A[m][n], rank, ...) ;
  end for
end for
```

Une fois les données enregistrées, il suffit alors de substituer l'appel de la routine `PZSYTRF`, qui invoque le solveur dense, par la routine `PZSYTRF_BLR`, qui invoque le solveur pour matrices compressibles, pour utiliser ce nouveau solveur dans l'application. Cette facilité de substitution des tâches du solveur dense par les tâches du solveur pour matrices compressées est un point fort de cette implémentation sur Chameleon, ce qui tend à soutenir l'idée que les supports d'exécution à base de tâches tels que StarPU facilitent le développement et l'intégration de nouveaux algorithmes dans des grands codes de calcul.

Comparaison des performances avec le solveur du CEA

La Figure 3.8 présente une comparaison des temps de factorisation du solveur pour matrices compressées développé au CEA avec notre solveur porté sur le support d'exécution StarPU, pour un cas de 750 000 inconnues, de 512 à 2592 cœurs de la machine TERA-100 Parallele (cf. Table 2.1), soit 16 à 81 nœuds de calcul. Cette figure confirme que le surcoût lié à l'utilisation de StarPU est limité lors du passage à l'échelle, et qu'il permet au solveur Chameleon d'exhiber des performances compétitives par rapport au solveur optimisé du CEA grâce aux optimisations de performance effectuées par StarPU.

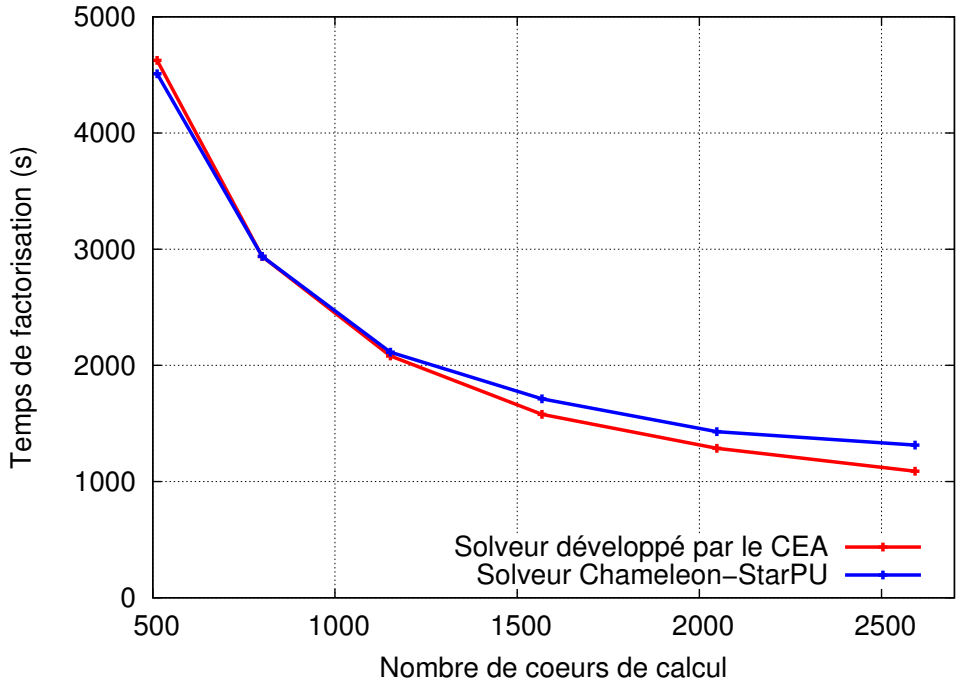


FIGURE 3.8 – Temps d'exécution du solveur pour matrices compressées de Chameleon et du solveur développé par le CEA en fonction du nombre de cœurs de calcul pour un cas de 750 000 inconnues.

Grossissement des données : un risque de débordement mémoire

Dans le cas de la FIGURE 3.8, nous n'avons volontairement pas tenté de saturer la mémoire des nœuds de calcul. En effet, si nous tentons d'utiliser toute la mémoire disponible pour le calcul, la variation imprévisible de la taille des données peut provoquer des débordements de mémoire qu'il est difficile d'anticiper. La FIGURE 3.9 présente l'encombrement mémoire du processus le plus chargé en mémoire durant l'exécution du solveur pour matrices compressées pour un cas de 450 000 inconnues sur 200 cœurs de la machine TERA-100 Hybrid (cf. TABLE 2.1), soit 25 nœurs de calcul. Comme précisé en Section 3.1, les GPUs des machines ne sont pas utilisés pour cette expérience.

Nous remarquons ici le même problème d'allocation agressive de mémoire pour les tampons de réception MPI que celui constaté en Section 2.4, mais que cela ne suffit pas pour provoquer de débordement de mémoire pour ce cas. Le débordement de mémoire est ici dû au grossissement des données locales de la matrice, modélisé sur cette figure par le delta entre la taille *initiale* et la taille *courante* des données de la matrice.

Nous présentons dans la section suivante une extension du contrôle de l'encombrement mémoire pour les données de taille variable. Nous montrons qu'il est possible d'utiliser des estimations de la variation de taille des données à la soumission des tâches pour prédire l'évolution de leur taille à l'exécution.

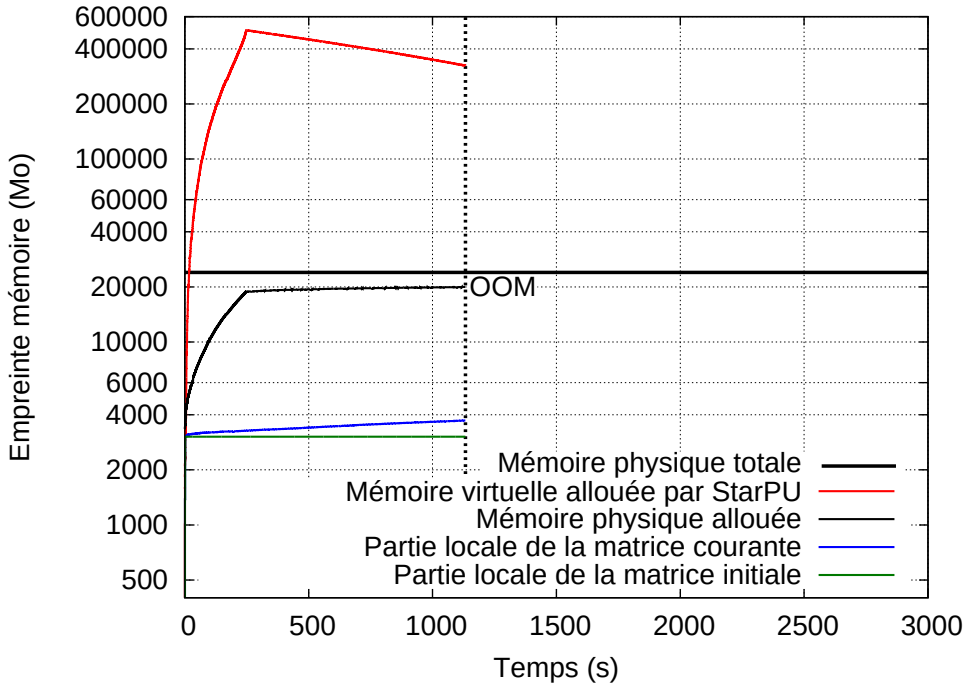


FIGURE 3.9 – Encombrement mémoire du processus le plus chargé en mémoire durant l'exécution d'un cas de 450 000 inconnues sur 200 cœurs de calcul

3.3 Contribution : gestion des données de taille variable avec StarPU

3.3.1 Estimer l'évolution de la taille des données

Pour chaque tâche soumise, StarPU va recevoir de l'application une évaluation de la taille que chaque donnée accédée en écriture, i.e dont la taille peut changer durant le calcul, aura à la terminaison de la tâche. StarPU va alors tenir à jour une estimation de la mémoire consommée par les tâches acceptées en soumission. À partir de cette estimation, StarPU est alors capable de bloquer la soumission de tâches lorsque toute la mémoire disponible a été réservée, comme présenté précédemment en Section 2.4. À la terminaison des tâches, la différence entre la taille réelle des données et leur taille estimée étant désormais connue, StarPU peut alors corriger les estimations et éventuellement accepter de nouvelles tâches en soumission.

Pour le cas du calcul distribué, la réception de données de taille variable pose également problème : comme le processus récepteur n'a pas de moyen de connaître la taille des données à recevoir lors de la soumission de la tâche, StarPU doit aussi évaluer la taille de ces données. Une fois les données reçues

et leur taille connue, comme à la terminaison des tâches, StarPU est alors en mesure de corriger les estimations et éventuellement de débloquer la soumission de tâches.

3.3.2 Mise en œuvre dans le solveur pour matrices compressées

Nous proposons que StarPU utilise des estimateurs et des correcteurs pour respectivement évaluer la taille des données à la soumission des tâches et corriger l'estimation lorsque la taille des données est connue. Ces estimateurs et correcteurs utilisent les routines `starpu_memory_allocate` et `starpu_memory_deallocate` pour avertir le gestionnaire de mémoire de StarPU du grossissement et de la réduction de la taille des données, respectivement.

Pour le cas de ce solveur, StarPU doit recevoir une estimation de la variation de taille des données des tâches de mise à jour de la matrice, qui sont celles qui modifient le taux de compression des tuiles, ainsi que des tampons de réception MPI. En effet, les données étant assemblées de manière distribuée sur les différents processus, chaque processus ne connaît que la taille des données qu'il a assemblées. La taille des données à recevoir n'est donc pas connue à la soumission des tâches.

Les correcteurs d'estimations sont eux appelés quand les tailles de données sont connues : pour les tâches de mise à jour, le correcteur est appelé juste avant la fin de chaque tâche ; pour les données provenant d'autres processus, le correcteur est appelé dans la méthode `unpack` de l'interface de données pour corriger l'estimation une fois le rang de la donnée reçue extrait du tampon de réception.

Ce mécanisme fonctionne pour les applications qui peuvent estimer de manière suffisamment approchée l'évolution de la taille de leurs données. Nous montrons en section suivante que les estimateurs du solveur pour matrices compressibles provoquent en fait une surconstriction du flot de soumission de tâches et une perte importante de performances car ils sont trop peu précis pour l'application ciblée. Nous proposons une solution pragmatique permettant de libérer du parallélisme dans ce solveur au prix de concessions sur la garantie d'exécution dans les limites de la mémoire disponible.

3.4 Précision des estimateurs et performances

Comme StarPU arrête d'accepter des tâches en soumission lorsque toute la mémoire dont il dispose a été réservée, StarPU peut s'arrêter très rapidement d'accepter des tâches en soumission si les estimateurs surévaluent fortement la taille des données.

Comme nous l'avons vu en FIGURE 2.8, le nombre de tâches acceptées en soumission par StarPU a une influence directe sur les performances du programme. Cette surestimation de la taille des données a donc pour conséquence directe de limiter très fortement les performances de l'application. Il existe un lien entre la précision des estimateurs et les performances de l'application : plus les estimateurs sont précis, plus le support d'exécution accepte de tâches en soumission, et plus le parallélisme disponible est important. À l'inverse, moins les estimateurs sont précis, plus le support d'exécution étrangle le flot de soumission de tâches, et plus le parallélisme disponible est faible.

3.4.1 Cas du solveur pour matrices compressées

Notre solveur pour matrices compressées est une représentation extrême de ce problème car StarPU va surestimer l'encombrement de mémoire du taux de compression des tuiles de la matrice, soit de plus de 90% pour la plupart des cas traités par le code de simulation du CEA. Les utilisateurs savent toutefois qu'en pratique les données de la matrice ne grossiront jamais au point de redevenir dense, même s'ils ne savent pas prédire leur grossissement.

Nous nous servons de cette information pour faire une concession sur la garantie d'exécution dans les limites de la mémoire de StarPU afin de libérer du parallélisme pour l'application ciblée. Pour ce faire, nous décidons de ne pas contrôler la variation de taille des données *locales* à un processus, car l'utilisateur sait que le grossissement des données ne sera pas très important. Il est par contre absolument nécessaire de conserver l'estimation de la taille des données provenant d'autres processus dans le cadre du calcul distribué car StarPU ne peut pas connaître la taille des données assemblées par d'autres processus, comme discuté précédemment en Section 3.3.2.

Nous montrons plus en détails le fonctionnement de ce contrôle mémoire *partiel* sur la FIGURE 3.10, qui présente l'encombrement de mémoire du processus le plus chargé en mémoire durant l'exécution du calcul sur le même cas que présenté précédemment en FIGURE 3.9. Cette figure montre que notre contrôle partiel de la mémoire fonctionne en pratique pour limiter la consommation de mémoire, notamment grâce au contrôle de l'encombrement de mémoire des tampons de réception. Nous remarquons aussi que le grossissement des données de la matrice locale, toujours représentée sur le graphe comme la différence entre la partie *locale* et la partie *courante* de la matrice, est bien contenu dans les limites de la mémoire virtuellement allouée par StarPU. Cela est dû au fait que la surestimation de la taille des données à recevoir d'autres processus est telle qu'en pratique, elle suffit en général largement pour recouvrir le grossissement des données locales.

Nous souhaitons également confirmer que cette solution permet de libérer suffisamment de parallélisme pour ne pas limiter les performances du solveur. Nous présentons en FIGURE 3.11 une comparaison des temps CPU, c'est-à-

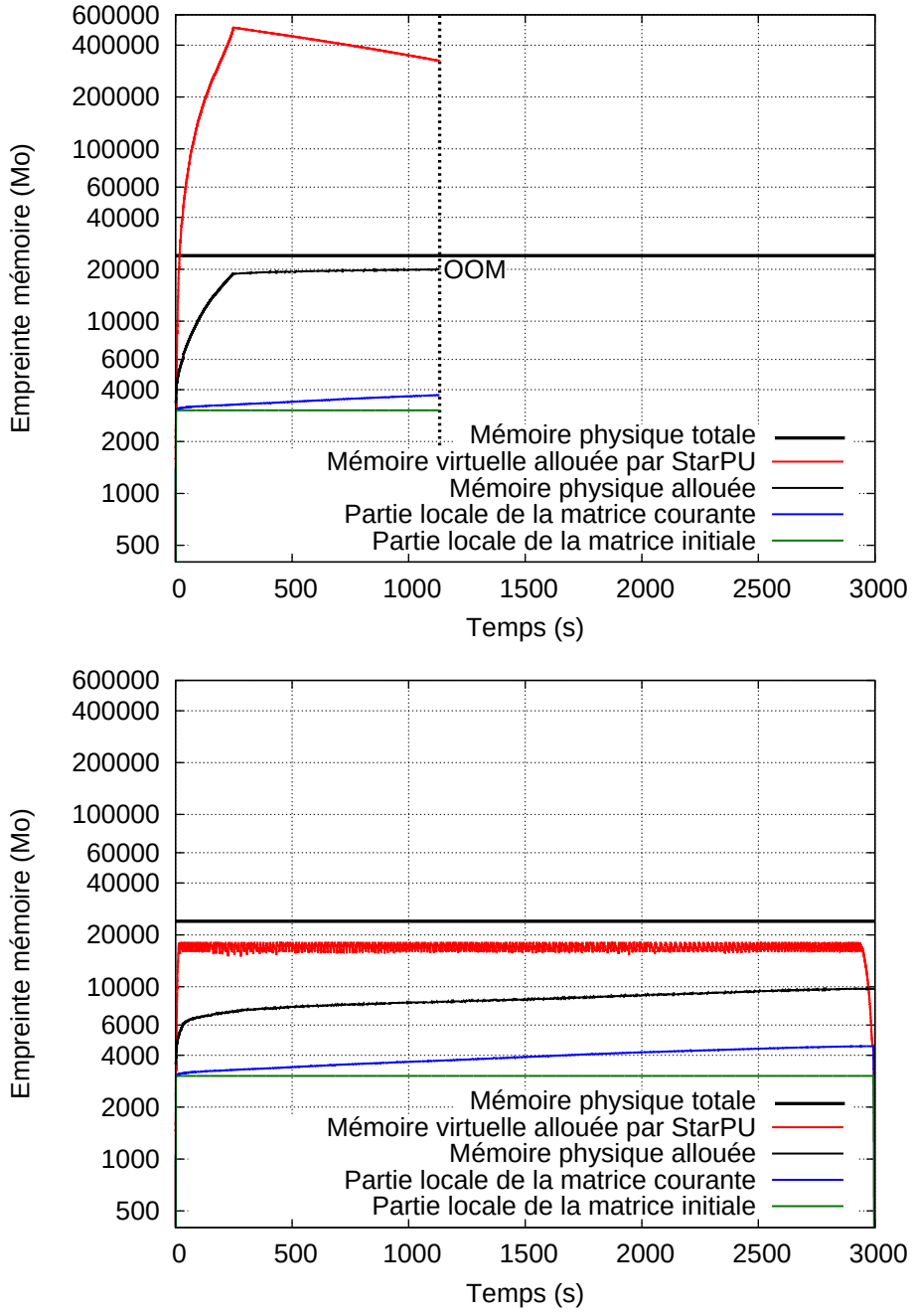


FIGURE 3.10 – Encombrement mémoire du processus le plus chargé en mémoire durant l'exécution d'un cas de 450 000 inconnues sur 25 nœuds de calcul sans (haut) et avec contrôle *partiel* de l'encombrement mémoire (bas).

dure du temps de calcul consommé sur les machines, en fonction du nombre de nœuds de calcul utilisés, avec et sans contrôle mémoire sur un cas de 450 000 inconnues.

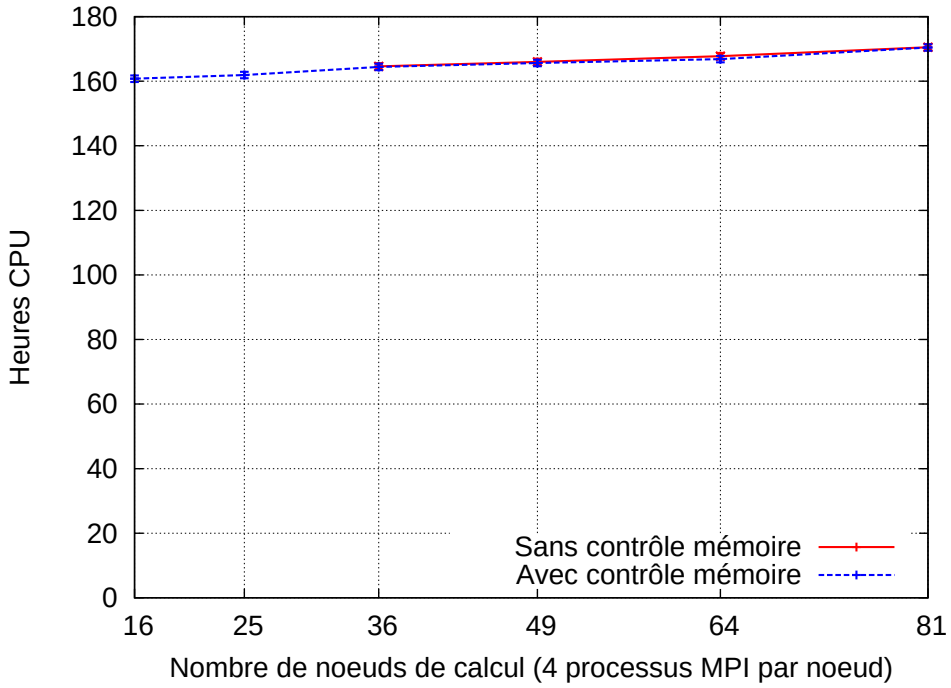


FIGURE 3.11 – Temps de calcul consommé sur les nœuds, représenté en heures CPU, en fonction du nombre de nœuds impliqués dans le calcul pour un cas de 450 000 inconnues.

Cette figure met en lumière deux propriétés importantes de ce contrôle de mémoire relaxé : tout d'abord, pour des nombres de nœuds de calcul allant de 36 à 81, les performances des deux solutions sont comparables. Ce contrôle mémoire libère donc effectivement suffisamment de parallélisme pour permettre au solveur de passer à l'échelle de manière satisfaisante. Ensuite, nous remarquons que seul le solveur avec contrôle mémoire peut exécuter ce cas de 450 000 inconnues sur 16 et 25 nœuds de calcul, tout en réduisant le temps CPU consommé. Cela montre que, malgré le relâchement de la contrainte mémoire, ce solveur peut s'exécuter sans erreur sur un nombre de nœuds inférieur à la version sans contrôle mémoire. Ces deux observations supportent donc cette approche pragmatique pour le passage à l'échelle de ce solveur.

3.5 Discussion

Un problème se pose si l'utilisateur a mal estimé le grossissement des données de son application au point qu'elles finissent par occuper toute la mémoire disponible pour StarPU. Selon la politique de contrôle par l'encombrement mémoire, StarPU n'est alors plus en mesure d'accepter la moindre tâche en soumission. Pour ne pas bloquer, StarPU va alors tenter de finir le calcul en

exécutant les tâches une par une pour minimiser la consommation de mémoire, et va prévenir l'utilisateur de cet état de fait pour l'inciter à utiliser plus de nœuds de calcul pour la prochaine exécution de son application.

Une solution simple pour obtenir plus de mémoire est d'utiliser les mémoires lentes des machines comme les disques dur, mais pour les mêmes raisons que précédemment discuté en Section 2.3.3, les utiliser pour le calcul pose le problème du temps d'accès à ces mémoires lentes, qui peut considérablement diminuer les performances du programme.

Une autre solution est que StarPU demande dynamiquement des ressources mémoire supplémentaires auprès de l'ordonnanceur de jobs du calculateur lorsqu'il détecte qu'il n'a bientôt plus assez de mémoire disponible. L'ordonnanceur de jobs peut alors bloquer l'exécution de l'application en attente des ressources mémoire demandées, puis la débloquer lorsqu'elles sont disponibles.

Chapitre 4

Combiner facilement différentes heuristiques d'ordonnancement

"Avec ma mère, faut que tout soit carré."
– Arthur, *Kaamelott Livre I, La visite d'Ygerne*

Résumé du chapitre **89**
4.1 Limites des ordonnanceurs monolithiques **90**
4.2 Contribution : les ordonnanceurs modulaires . . . **91**
4.2.1 Principes des ordonnanceurs modulaires 92
4.2.2 Progression des tâches dans l'ordonnanceur 93
4.2.3 Des composants d'ordonnancement 94
4.2.4 Interface des composants : demandes actives et passives 94
4.3 Résultats **97**
4.3.1 Dimensionnement des réservoirs 97
4.3.2 Comportement des ordonnanceurs 99
4.3.3 Performances 101
4.4 Discussion **101**

Résumé du chapitre

Nous avons étudié dans les chapitres précédents le passage à l'échelle de notre application en mémoire distribuée, mais il est également important également d'optimiser les performances sur un seul nœud de calcul. Nous proposons dans ce chapitre de nous intéresser au passage à l'échelle des ordonnanceurs de tâches de StarPU. Comme présenté en Section 1.3.1, l'ordonnancement de tâches est un problème NP-complet et des heuristiques plus ou moins complexes existent pour effectuer cet ordonnancement. Nous

ne proposons pas dans ce chapitre une nouvelle heuristique d'ordonnancement : nous entamons une réflexion sur la structure des ordonnanceurs, et nous montrons qu'une formalisation modulaire des ordonnanceurs de tâches permet de faciliter la combinaison de plusieurs heuristiques au sein d'un même ordonnanceur. Nous montrons dans des travaux préliminaires que ces ordonnanceurs modulaires affichent des performances comparables à leurs équivalents monolithiques, et qu'ils facilitent l'élaboration d'ordonnanceurs évolués dans StarPU.

4.1 Limites des ordonnanceurs monolithiques

Les ordonnanceurs de tâches servent à décider pour chaque tâche l'unité de calcul qui va l'exécuter. Dans StarPU, les tâches sont poussées (*Push*) dans l'ordonnanceur uniquement lorsqu'elles sont prêtes, i.e que toutes leurs dépendances de tâches sont satisfaites et que leurs données sont disponibles. Lorsqu'une unité de calcul est disponible pour exécuter une tâche, le Worker StarPU associé à cette unité de calcul récupère (*Pull*) une tâche depuis l'ordonnanceur, puis l'exécute. Ne pousser que les tâches prêtes vers l'ordonnanceur permet de ne pas le surcharger avec des tâches qui ne sont pas encore prêtes à être exécutées et d'éviter de prendre des décisions d'ordonnancement trop à l'avance. Par contre, cela diminue la clairvoyance de l'ordonnanceur et peut dégrader la qualité de l'ordonnancement car il ne peut prendre des décisions que sur les tâches dont il a connaissance : il ne sait pas si des tâches nécessitant un traitement prioritaire vont être poussées plus tard ou non. Un autre problème de clairvoyance de la plupart des ordonnanceurs actuels de StarPU est qu'ils prennent la décision d'ordonnancement d'une tâche sur une unité de calcul lorsqu'elle est poussée dans l'ordonnanceur. Ce fonctionnement leur permet notamment de précharger les données sur les unités de calcul (comme les GPUs par exemple), mais cela peut dégrader la qualité de l'ordonnancement effectué car l'ordonnancement est parfois décidé beaucoup de temps avant l'exécution réelle de la tâche, ce qui empêche souvent de gérer correctement les priorités.

Comme StarPU permet à l'utilisateur d'implémenter ses propres stratégies d'ordonnancement, certaines applications qui utilisent StarPU, comme ScalFMM [9] implémentent leur propre ordonnanceur pour récupérer cette clairvoyance perdue en injectant des informations spécifiques à l'application dans l'ordonnanceur. Pour résoudre ces problèmes de clairvoyance sans obliger les applications à écrire leur propre ordonnanceur, une solution consiste à prendre la décision d'ordonnancement non pas quand les tâches sont poussées dans l'ordonnanceur (au *Push*), mais quand les Workers StarPU demandent des tâches depuis l'ordonnanceur (au *Pull*). De cette manière, les décisions

d'ordonnancement sont prises de manière paresseuse, uniquement lorsque les unités de calcul réclament du travail. Des ordonnanceurs de tâches ayant cette propriété d'ordonnancement paresseux peuvent prendre de meilleures décisions d'ordonnancement. Ils ont connaissance de l'intégralité des tâches déjà poussées par StarPU dans l'ordonnanceur et peuvent choisir quelle tâche ordonnancer, là où les ordonnanceurs actuels de StarPU ont l'obligation d'ordonnancer immédiatement toute tâche poussée. De fait, leur clairvoyance est fortement améliorée.

Une autre propriété que nous souhaitons avoir dans nos ordonnanceurs de tâches est de pouvoir combiner facilement plusieurs heuristiques d'ordonnancement. Comme présenté en Section 1.3.1, certaines heuristiques d'ordonnancement (comme HEFT) produisent de bons ordonnancements sur des architectures hétérogènes mais sont coûteuses à calculer, et à l'inverse d'autres heuristiques d'ordonnancement (comme Eager) sont très peu coûteuses mais ne produisent de bons ordonnancements que sur des architectures homogènes. Actuellement, la majorité des ordonnanceurs de StarPU n'implémentent qu'une seule heuristique d'ordonnancement : ils sont dits *monolithiques*. Combiner plusieurs ordonnanceurs monolithiques entre eux est possible mais difficile, car chaque ordonnanceur implémente à sa façon les éléments de base nécessaires à l'ordonnancement : listes de tâches, modules de préchargement des données, etc.

Pour satisfaire ces deux propriétés, nous proposons une formalisation de la structure d'un ordonnanceur permettant de distinguer les composantes de l'ordonnancement nécessaires pour satisfaire ces propriétés, et de les exprimer en éléments simples qui peuvent s'assembler pour créer des ordonnanceurs totalement modulaires.

4.2 Contribution : les ordonnanceurs modulaires

Cette idée de modulariser les ordonnanceurs a déjà été explorée dans le domaine de l'ordonnancement système, où des efforts pour permettre à un processus en espace utilisateur de décrire des ordonnanceurs et de les faire utiliser par le système d'exploitation ont été étudiés [40]. Les travaux les plus avancés dans ce domaine sont ceux autour de Bossa [52, 17], qui propose un langage dédié (DSL) permettant à un programme en espace utilisateur d'implémenter un ordonnanceur dans ce langage dédié, qui est ensuite généré sous forme d'un assemblage de composants puis utilisé par le système d'exploitation. Notre contribution est, à notre connaissance, le premier effort pour transposer ces travaux aux ordonnanceurs de tâches pour les supports d'exécution.

4.2.1 Principes des ordonnanceurs modulaires

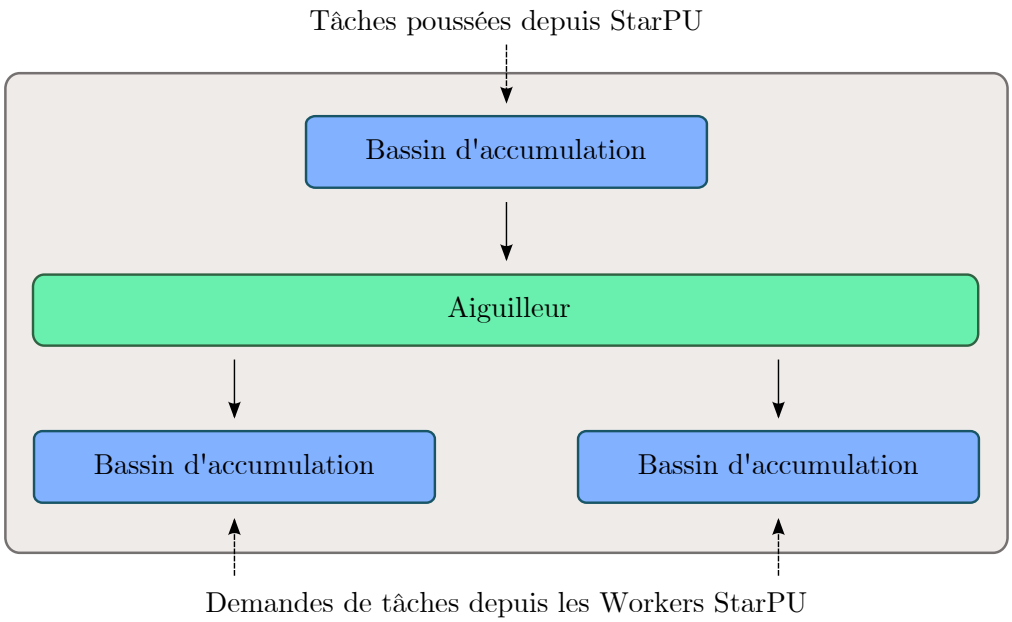


FIGURE 4.1 – Un exemple d'ordonnanceur modulaire.

Le concept clé de notre contribution repose sur le constat que les ordonnanceurs de tâches peuvent s'exprimer en éléments simples, chacun représentant une composante spécifique de l'ordonnanceur de tâches, et qu'il est possible de les assembler pour former un ordonnanceur de tâches fonctionnel. Les composantes de base nécessaires aux ordonnanceurs de tâches pour disposer des propriétés souhaitées de clairvoyance et de combinaison facile sont les suivantes : d'abord, pour permettre l'ordonnancement paresseux des tâches, il faut un élément capable de stocker des tâches temporairement en attendant la demande d'ordonnancement : nous l'appelons un *bassin d'accumulation*. Ensuite, l'ordonnanceur doit prendre des décisions d'ordonnancement pour les tâches, selon une ou plusieurs heuristiques. Ces décisions aiguillent chaque tâche vers l'unité de calcul qui va l'exécuter : c'est pourquoi nous appelons ces éléments des *aiguilleurs*. Enfin, tous les *effets de bord* de l'ordonnanceur comme le préchargement des données sur les unités de calcul ou le choix de la meilleure implémentation d'une tâche sont gérés dans des éléments qui leur sont spécifiques. Maintenant que nous avons nos briques de base, il faut déterminer comment les assembler pour obtenir un ordonnanceur fonctionnel. En pratique, les ordonnanceurs de tâches de StarPU disposent d'un unique point d'entrée *en haut* de l'ordonnanceur, par lequel StarPU pousse les tâches dans l'ordonnanceur. Ils disposent par contre de plusieurs points de sortie *en bas*, qui correspondent aux Workers StarPU qui récupèrent des tâches depuis l'or-

donnanceur pour les exécuter. Un arbre d'ordonnancement, où les tâches sont poussées à la racine de l'arbre et prélevées à ses feuilles, correspondrait parfaitement à la configuration des points d'entrée et de sortie des ordonnanceurs de tâches de StarPU. Nous proposons donc des ordonnanceurs modulaires à structure arborescente, où les nœuds sont les éléments simples de notre ordonnanceur et les arêtes sont les liens permettant de faire progresser les tâches d'un élément à un autre. Un exemple d'ordonnanceur modulaire est présenté en FIGURE 4.1.

4.2.2 Progression des tâches dans l'ordonnanceur

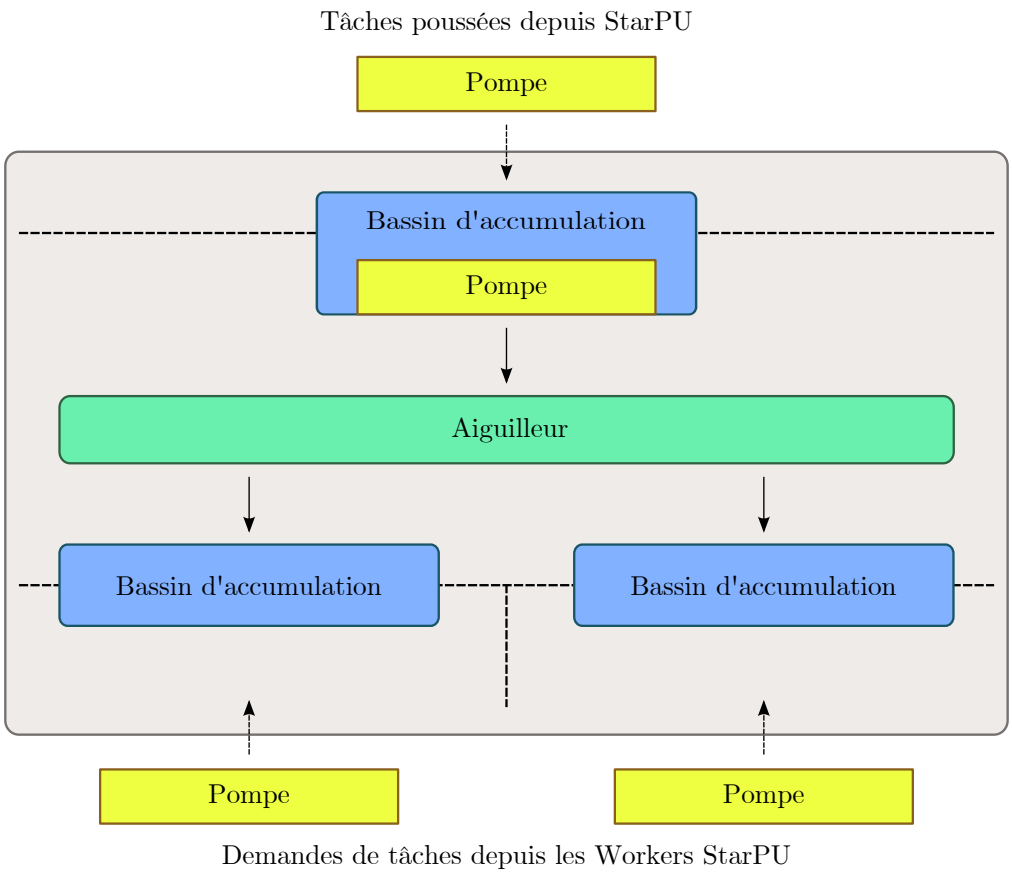


FIGURE 4.2 – Ordonnanceur modulaire de la FIGURE 4.1 avec les pompes. Les traits pointillés délimitent les zones d'ordonnancement de l'ordonnanceur modulaire.

Dans nos ordonnanceurs modulaires, nous nommons les moteurs qui déclenchent la progression des tâches dans l'ordonnanceur des *pompes*. Les tâches poussées par StarPU dans l'ordonnanceur ainsi que les demandes de tâches des

Workers StarPU provoquent la progression des tâches dans l'ordonnanceur : ce sont donc des pompes. Ces pompes *natives* ne sont cependant pas suffisantes pour assurer la progression des tâches dans l'arbre d'ordonnancement, car les tâches peuvent être stockées dans des bassins d'accumulation pour permettre l'ordonnancement paresseux. Les bassins d'accumulation délimitent ce que nous appelons des *zones d'ordonnancement*, comme le montre la Figure 4.2, car ils délimitent des zones dans lesquelles les tâches progressent dans l'arbre sans être arrêtées, jusqu'à ce qu'elles soient stockées dans un bassin d'accumulation. Pour assurer la progression des tâches de bassin en bassin, il faut et il suffit donc d'assurer la présence d'une pompe par zone d'ordonnancement.

4.2.3 Des composants d'ordonnancement

Le modèle d'ordonnanceurs modulaires que nous proposons est basé sur la programmation par composants [66]. Dans ce modèle, les fonctions sont encapsulées dans les composants, et les communications entre composants du même type sont régies par une interface unifiée.

Nous avons implémenté différentes classes de composants d'ordonnancement, chacun représentant une des composantes simples dont nous avons besoin pour que les ordonnanceurs modulaires satisfassent les propriétés souhaitées de clairvoyance et de composabilité des heuristiques d'ordonnancement. Les composants *réservoir* implémentent les bassins d'accumulation : ils stockent les tâches qui sont poussées vers eux dans une liste locale au composant, et les transmet à d'autres composants sur demande. Les composants *d'aiguillage* implémentent les aiguilleurs, et par conséquent les heuristiques d'ordonnancement. Les composants *à effets de bord* implémentent les opérations à effet de bord nécessaires au bon fonctionnement de l'ordonnance modulaire, comme le choix de la meilleure implémentation ou le préchargement des données, par exemple. Finalement, les composants *Worker* sont une catégorie particulière de composants à effets de bord. Ils représentent les workers StarPU dans le modèle des ordonnanceurs modulaires et permettent d'effectuer les conversions de requêtes nécessaires pour assurer la compatibilité des ordonnanceurs modulaires avec les workers StarPU.

4.2.4 Interface des composants : demandes actives et passives

Les composants d'ordonnancement communiquent entre eux via une interface unifiée permettant ainsi l'assemblage facile de composants, comme présenté sur la Figure 4.3.

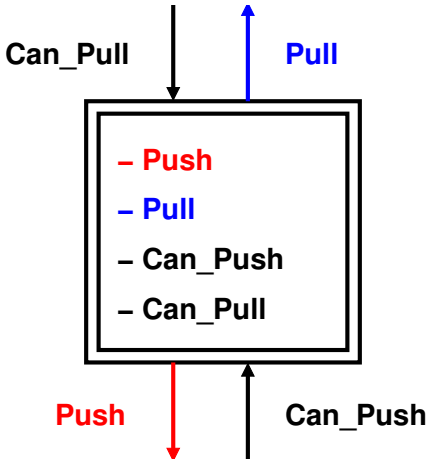
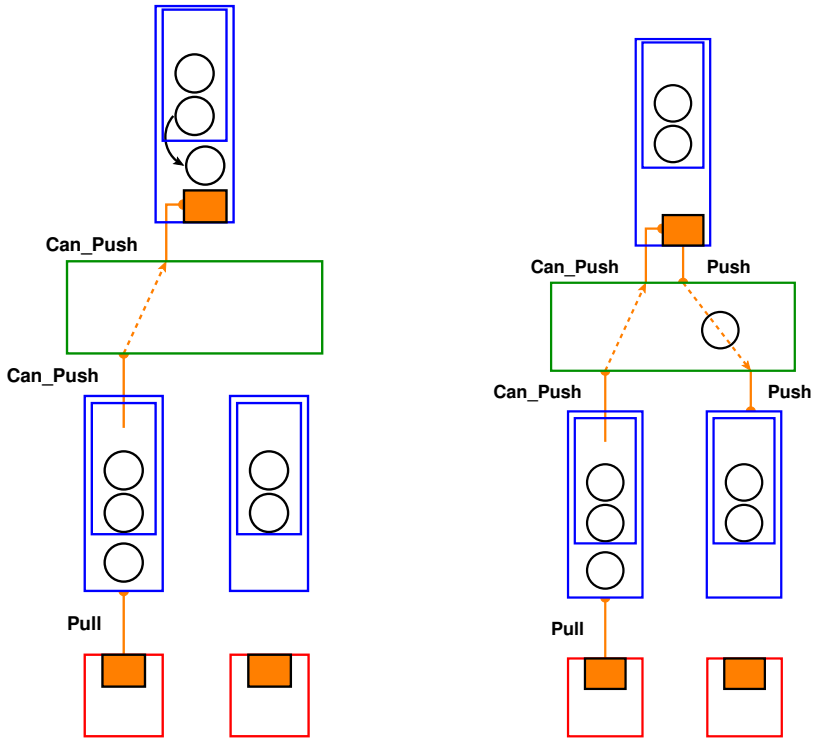


FIGURE 4.3 – Interface d'un composant d'ordonnancement.

L'interface est divisée en deux parties : les demandes *actives* et les demandes *passives*. Une demande active consiste à transférer une tâche d'un composant à un autre dans l'arbre d'ordonnancement. La méthode `Push` permet à un composant de pousser une tâche vers un de ses fils dans l'arbre d'ordonnancement, et `Pull` permet à un composant de récupérer une tâche depuis son père dans l'arbre d'ordonnancement. Bien que ces deux méthodes permettent effectivement de faire progresser les tâches dans l'ordonnanceur, elles ne satisfont pas les propriétés souhaitées pour nos ordonnanceurs. En effet, si les Workers StarPU essayent de réclamer une tâche à un composant réservoir vide de tâches via un appel à la méthode `Pull`, ce réservoir ne va pas avoir d'autre choix que de propager cette demande jusqu'à un réservoir de plus haut niveau afin d'obtenir une tâche pour le Worker StarPU. Si un composant d'aiguillage se trouve entre ces deux réservoirs dans l'arbre d'ordonnancement, il est possible que la tâche ne soit pas aiguillée vers le réservoir qui en fait la demande lors de son ordonnancement. Et même si c'est le cas, les transferts de données ne pourront être effectués en avance de phase, ce qui va dégrader les performances de l'application. Le problème mis en avant par ce scénario est que l'ordonnancement peut être trop paresseux. Il est nécessaire que les tâches progressent d'une zone d'ordonnancement à une autre de manière régulière, en réponse à des événements qui se produisent dans l'ordonnanceur.

Les demandes passives permettent à un composant de notifier les autres composants de l'arbre qu'un événement permettant de faire progresser des tâches s'est produit dans ce composant. Dans notre interface de composants d'ordonnancement, ce sont les méthodes `Can_Push` et `Can_Pull` qui permettent ces notifications : elles servent respectivement à notifier au composant parent qu'il lui est possible de pousser des tâches et de notifier aux composants fils qu'il leur est possible de réclamer des tâches. Les demandes passives nous permettent ainsi d'implémenter des pompes dans certains composants : actuelle-

ment, les deux composants qui implémentent une pompe dans notre modèle d'ordonnanceurs sont les composants réservoirs et les composants Worker.



(a) Montée du `Can_Push` (b) Redescente du `Push`

FIGURE 4.4 – Transformations des demandes *passives* en demandes *actives* : exemple du `Can_Push`

La FIGURE 4.4 montre avec l'exemple de la transformation du `Can_Push` en `Push` dans un composant réservoir comment une demande passive peut activer la pompe d'un composant. Lorsqu'une tâche est prélevée par un composant Worker dans un composant réservoir par un appel à la méthode `Pull` du réservoir (FIGURE 4.4a), de la place est libérée dans le réservoir : il n'est plus plein. Cet événement provoque la notification au composant parent qu'il est possible de pousser des tâches vers ce réservoir par une demande passive `Can_Push`. Cet appel se propage jusqu'à un composant capable de traiter cette demande, dans ce cas un autre composant réservoir. Lorsque cette demande arrive au niveau du composant réservoir, le composant réservoir active sa pompe qui transforme cette demande en une ou plusieurs demandes actives `Push`, pour faire progresser des tâches vers la zone d'ordonnancement suivante. Si un composant d'aiguillage se trouve dans la zone d'ordonnancement, il est possible que le composant réservoir demandeur ne soit pas rempli (FIGURE 4.4b). La pompe continue alors de pousser des tâches jusqu'à ce que la dernière tâche poussée par la pompe lui revienne, ce qui signifie que le réservoir vers lequel

cette tâche doit être poussée est déjà plein. Grâce à ce mécanisme, les décisions d'ordonnancement sont prises de manière paresseuse, mais pas trop, ce qui permet le bon fonctionnement des effets de bord de l'ordonnanceur ainsi qu'une bonne clairvoyance des heuristiques d'ordonnancement.

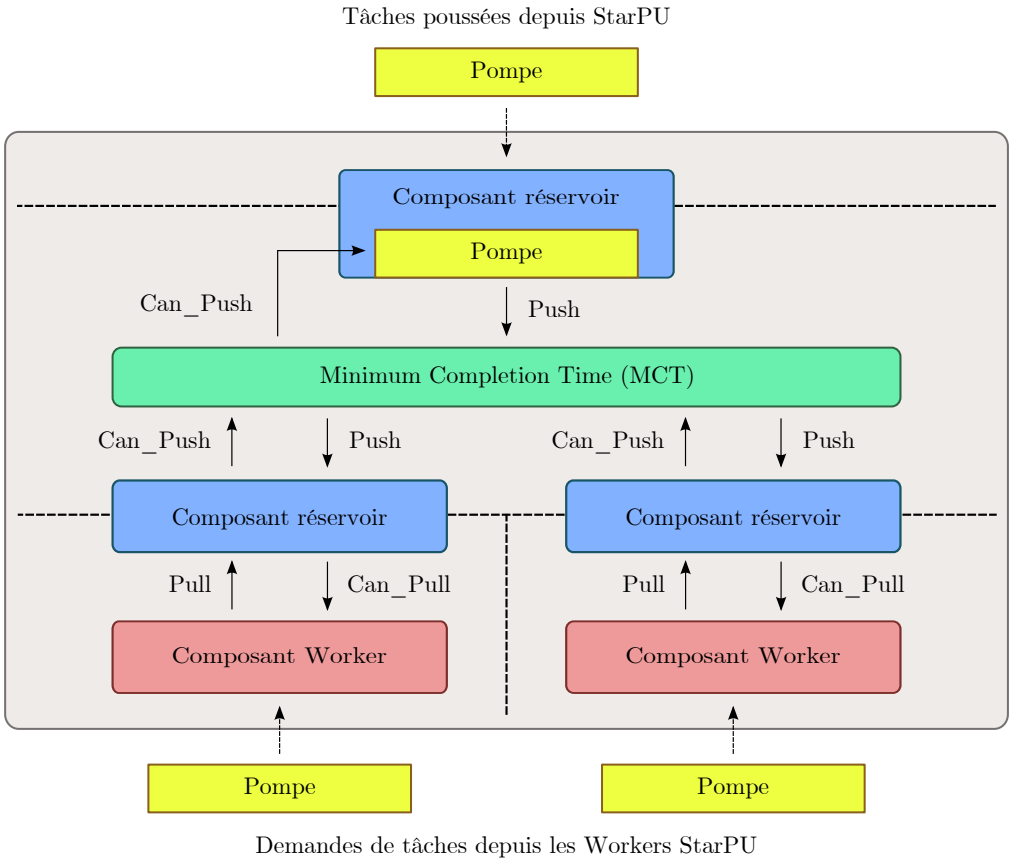


FIGURE 4.5 – Ordonnanceur modulaire *modular-heft*

La FIGURE 4.5 spécifie l'ordonnanceur vu précédemment en FIGURE 4.1 et forme un assemblage valide de composants qui implémente la même heuristique que l'ordonnanceur monolithique *dmdas* de StarPU : c'est l'ordonnanceur modulaire *modular-heft*. Nous présentons dans la Section 4.3 une étude des performances et du comportement de ces deux ordonnanceurs afin de valider notre modèle d'ordonnanceurs modulaires.

4.3 Résultats

4.3.1 Dimensionnement des réservoirs

L'une des propriétés souhaitées pour les ordonnanceurs modulaires est la clair-

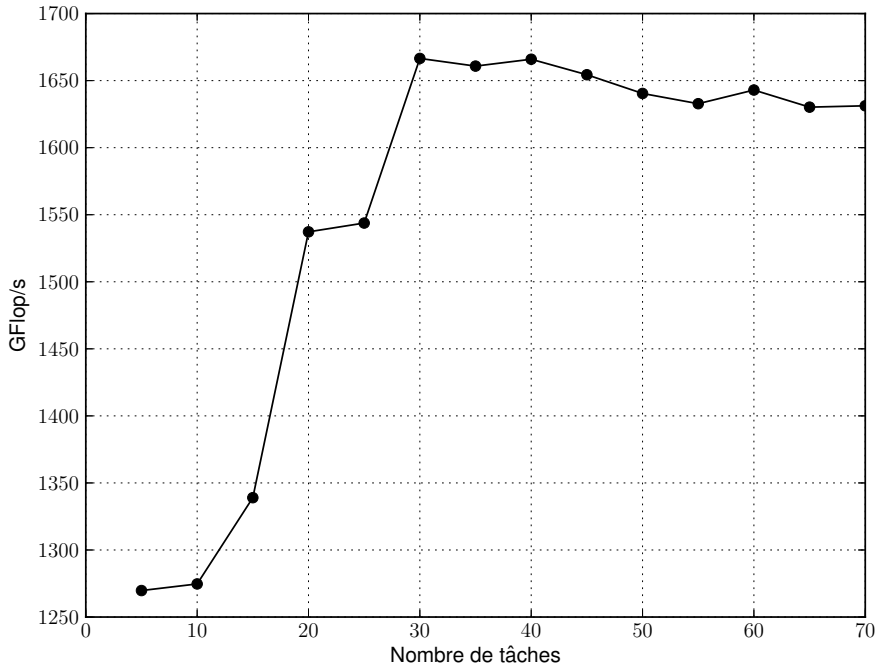


FIGURE 4.6 – Influence du dimensionnement des réservoirs de plus bas niveau de l'ordonnanceur *modular-heft* sur les performances de notre application pour une matrice de 48 000 inconnues.

voyance par l'ordonnancement paresseux des tâches, mais pas trop pour permettre aux effets de bord de l'ordonnanceur de fonctionner correctement. Comme ce sont les réservoirs de plus bas niveau qui déclenchent l'activation des pompes des réservoirs de plus haut niveau, la paresse d'un ordonnanceur modulaire est directement liée à la taille de ses réservoirs de plus bas niveau : plus ils sont petits, plus il faut les remplir souvent et moins les effets de bord, notamment le préchargement des données, fonctionnent correctement mais la clairvoyance est meilleure car les tâches sont ordonnancées au plus proche de leur exécution réelle ; plus ils sont gros, plus les effets de bord fonctionnent correctement mais la clairvoyance est réduite car les tâches sont ordonnancées plus à l'avance, ce qui peut dégrader la qualité de l'ordonnancement.

C'est exactement ce qu'on observe sur la FIGURE 4.6, qui présente une étude de l'influence du dimensionnement des réservoirs de plus bas niveau de l'ordonnanceur modulaire *modular-heft* sur les performances de la configuration n°3 (cf. TABLE 2.3) de notre application sur 1 nœud de la machine Mirage (cf. TABLE 2.1) pour une matrice de 48 000 éléments. De 5 à 15 tâches, les performances sont très faibles car l'ordonnanceur n'a pas le temps de précharger les données sur les unités de calcul, notamment les GPUs, avant que les tâches

ne soient récupérées par les Workers StarPU. Les performances augmentent à partir de 20 tâches grâce au bon fonctionnement du préchargement des données. Les performances augmentent encore à partir de 30 tâches car, sur cette machine, le facteur de gain de vitesse sur GPU par rapport aux CPUs du noyau de calcul dominant de notre application (GEMM) est de 30. L'ordonnanceur peut donc, à partir de cette valeur de la taille de réservoir, correctement distribuer ces tâches aux unités de calcul. Au-delà de 30 tâches, les performances diminuent car des réservoirs de plus bas niveau trop gros provoquent l'ordonnancement des tâches vers ces réservoirs trop de temps avant leur exécution réelle, et fait perdre de la clairvoyance à l'ordonnanceur. Nous avons donc fixé la taille des réservoirs de plus bas niveau à 30 tâches afin d'obtenir les meilleures performances de notre ordonnanceur modulaire.

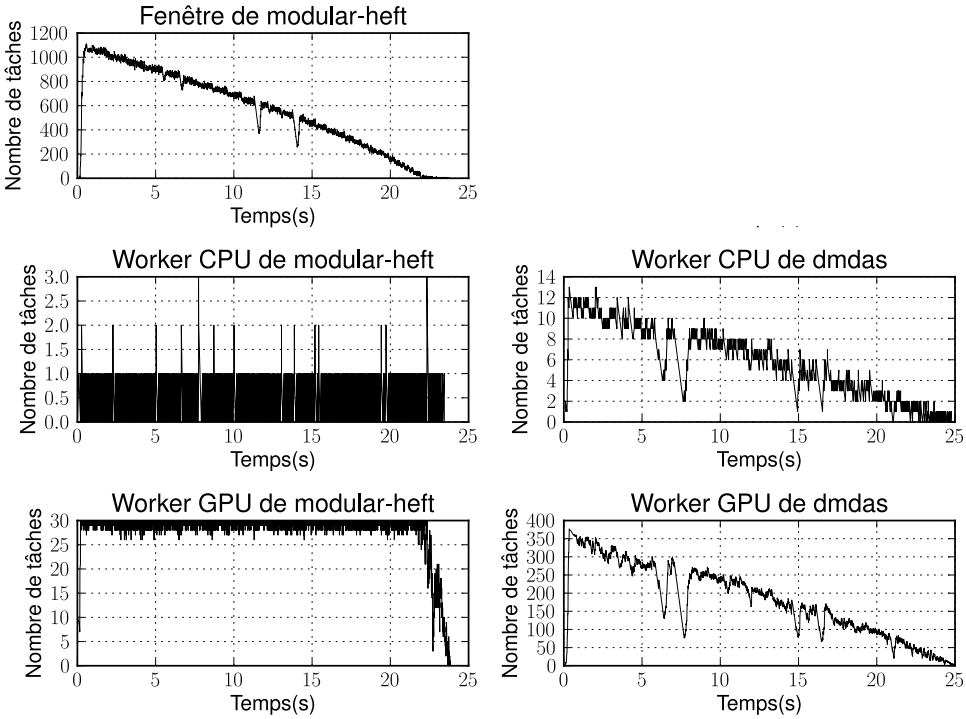


FIGURE 4.7 – État des files de tâches des ordonnanceurs *modular-heft* (gauche) et *dmdas* (droite) durant l'exécution.

4.3.2 Comportement des ordonnanceurs

La FIGURE 4.7 présente l'état des files de tâches de l'ordonnanceur *modular-heft* (à gauche) et de l'ordonnanceur *dmdas* (à droite) durant l'exécution de notre application sur le même cas que la figure précédente. Nous remarquons sur ces

graphes que l'ordonnanceur *dmdas* de StarPU surcharge fortement les GPUs, avec 380 tâches par GPUs, par rapport aux CPUs, qui n'ont pas plus d'une dizaine de tâches chacun, ce qui reflète bien l'affinité forte qu'ont les tâches de la factorisation de Cholesky avec les GPUs (notamment le GEMM). Nous remarquons également de fortes variations du nombre de tâches prêtes par moments dans les files de tâches de l'ordonnanceur *dmdas*. Ces variations sont dues au fait que les tâches sont ordonnancées dès qu'elles sont poussées dans l'ordonnanceur, ce qui provoque des problèmes de priorisation des tâches prioritaires. En effet, le déblocage du chemin critique par l'exécution des tâches prioritaires est retardé à cause du comportement de l'ordonnanceur, ce qui provoque ces chutes du nombre de tâches prêtes. Pour l'ordonnanceur modulaire *modular-heft*, nous remarquons premièrement qu'il surcharge les GPUs par rapport aux CPUs de la même manière que l'ordonnanceur *dmdas*, avec un facteur 30 entre nombre de tâches sur GPUs et sur CPUs. Par contre, les variations du nombre de tâches prêtes sont fortement lissées par l'ordonnanceur modulaire *modular-heft* grâce à la présence d'un deuxième étage de composants réservoirs dans cet ordonnanceur. Cet étage, appelé fenêtre d'ordonnancement dans la théorie de l'ordonnancement et sur la Figure 4.7, permet l'ordonnancement paresseux des tâches et améliore la gestion des priorités des tâches, ce qui permet de débloquer plus rapidement le chemin critique de la factorisation et de lisser les variations du nombre de tâches prêtes durant l'exécution.

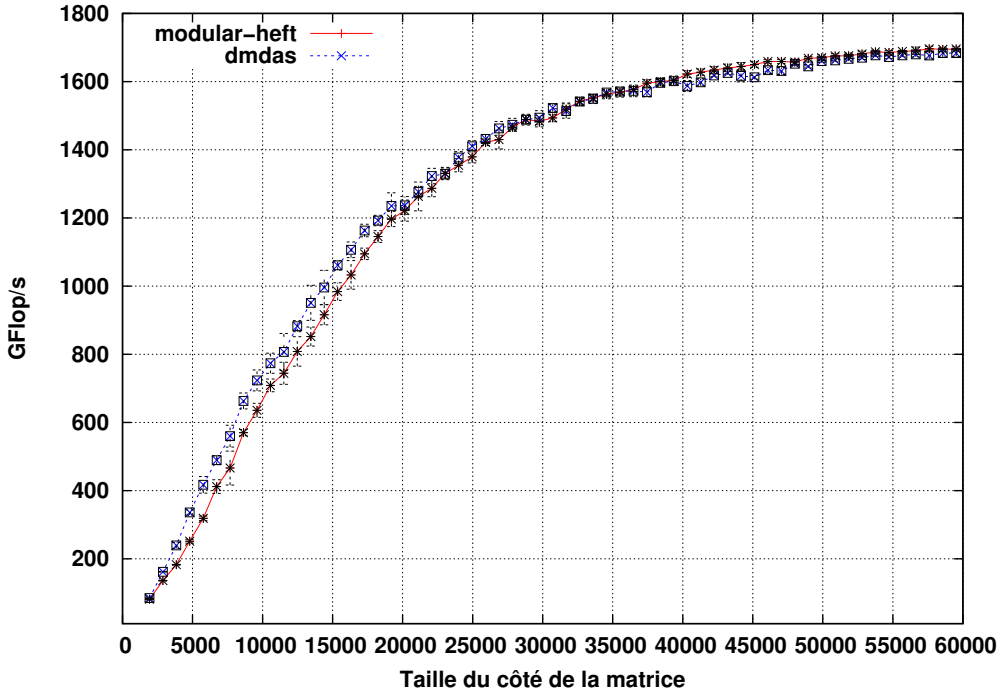


Figure 4.8 – Performances des ordonnanceurs modular-heft et dmdas.

4.3.3 Performances

La Figure 4.8 présente une comparaison des performances des ordonnanceurs *modular-heft* et *dmdas* pour la configuration n°4 (cf. Table 2.3) de notre application sur 1 nœud de la machine Mirage (cf. Table 2.1). Les performances des deux ordonnanceurs sont comparables pour des matrices de taille intermédiaire et grande, ce qui confirme que nos ordonnanceurs modulaires ont des performances comparables aux ordonnanceurs monolithiques lors du passage à l'échelle. Par contre, pour les petites matrices, l'ordonnanceur modulaire *modular-heft* est moins performant que *dmdas*. Ceci est dû au surcoût de la formalisation par composants des ordonnanceurs modulaires par rapport aux ordonnanceurs monolithiques. En effet, chaque progression de tâche dans un ordonnanceur modulaire se traduit concrètement par plusieurs échanges de tâches entre composants, notamment par l'effet des demandes passives (comme le `Can_Push` vu précédemment) qui provoquent la descente des tâches vers les feuilles de l'arbre d'ordonnancement. En revanche, les ordonnanceurs monolithiques réduisent au maximum le nombre de traitements pour optimiser l'ordonnancement. Ce surcoût des ordonnanceurs modulaires est notable pour les petites matrices car le ratio entre le surcoût de l'ordonnanceur et la quantité de calcul n'est pas négligeable. Par contre, pour les grandes matrices, le surcoût de l'ordonnanceur étant négligeable face à la quantité de calcul, notre ordonnanceur modulaire a des performances comparables à son équivalent monolithique.

4.4 Discussion

Pour limiter le surcoût des ordonnanceurs modulaires par rapport à leurs homologues monolithiques, il est possible de réduire la fréquence d'émission de demandes passives en mettant en place un mécanisme d'*hysteresis* : au lieu de demander le remplissage d'un réservoir dès qu'une tâche y est prélevée, il est possible de ne le faire que périodiquement, ou de ne le déclencher que lorsque le nombre de tâches dans le réservoir descend sous un seuil de remplissage par exemple. De ce seuil de remplissage dépendra le compromis choisi par l'utilisateur, comme pour le choix de la taille des réservoirs, entre la paresse nécessaire pour obtenir une bonne clairvoyance de l'ordonnanceur et l'ordonnancement suffisamment à l'avance pour permettre aux effets de bord, notamment le préchargement des données sur les unités de calcul, de s'effectuer correctement.

La paresse n'est pas la seule propriété nécessaire pour obtenir une bonne clairvoyance des ordonnanceurs de tâches. Dans StarPU, les tâches sont poussées une par une dans l'ordonnanceur. Cette méthode de transmission des tâches à l'ordonnanceur est problématique car un ordonnanceur trop agressif va chercher à ordonnancer immédiatement cette tâche sans considérer celles

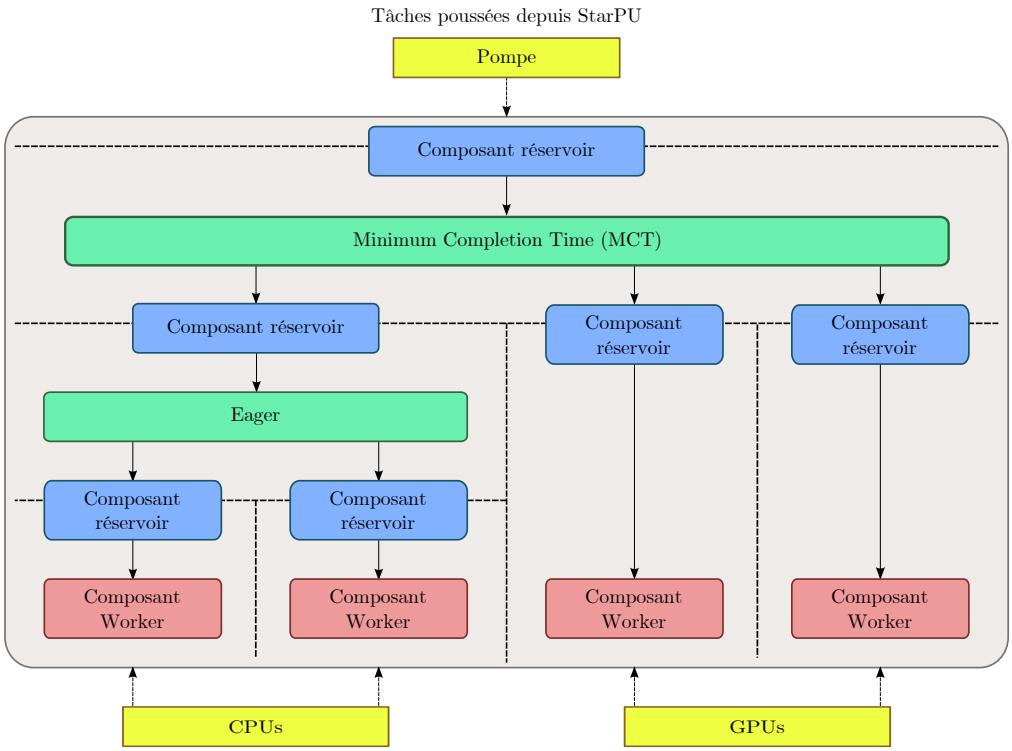


FIGURE 4.9 – Un exemple d'ordonnanceur modulaire qui couple plusieurs heuristiques d'ordonnancement : *hierarchical-heft*

qui vont être poussées après dans l'ordonnanceur. Il faut donc dans l'ordonnanceur un système de stockage de tâches qui n'autorise leur ordonnancement que lorsqu'un certain nombre de tâches sont déjà présentes dans l'ordonnanceur. Dans le modèle des ordonnanceurs modulaires, une solution simple consiste à implémenter une spécialisation du composant réservoir qui refuse de pousser des tâches vers ses fils tant qu'il ne contient pas un certain nombre de tâches. Ce nouveau composant ferait ainsi effet de bouchon de stockage de tâches, ce qui permet à l'ordonnanceur de considérer les tâches par paquets et de retrouver ainsi la clairvoyance perdue. Pour permettre au programmeur d'implémenter de nouveaux composants d'ordonnancement (comme ce composant *bouchon* par exemple), nous fournissons dans StarPU un squelette de composant d'ordonnancement. De base, ce squelette implémente un composant vide qui ne fait que transmettre les demandes *actives* et *passives* à son parent et à ses fils dans l'arbre de l'ordonnanceur modulaire. Créer un nouveau composant d'ordonnancement consiste donc à respécifier tout ou partie des méthodes de l'interface du squelette, selon les fonctionnalités souhaitée du composant. Par exemple, les composants réservoir ne respécifient que les méthodes `Push` et `Can_Push` de l'interface des composants d'ordonnancement et réutilisent les autres méthodes telles quelles. À l'inverse, les composants Worker ne respécifient que les méthodes `Pull` et `Can_Pull` de l'interface.

L'objectif originel des ordonnanceurs modulaires est de permettre l'étude d'ordonnanceurs qui couplent plusieurs heuristiques d'ordonnancement, et les ordonnanceurs modulaires ouvrent cette possibilité. Un exemple d'ordonnanceur que nous souhaitons étudier est l'ordonnanceur modulaire *hierarchical-heft*, présenté en FIGURE 4.9. Cet ordonnanceur implémente une heuristique similaire aux ordonnanceurs *modular-heft* et *dmdas*, mais sa structure lui permet d'exploiter le meilleur des chaque heuristique d'ordonnancement. Le coût de l'heuristique *MCT* est réduit car cette heuristique n'a à choisir d'ordonnancer les tâches qu'en fonction de l'architecture des ressources sous-jacentes (i.e CPU ou GPU). Pour l'ordonnancement entre ressources de calcul homogènes, l'heuristique simple *eager*, qui passe très bien à l'échelle, est utilisée. Cet exemple montre les possibilités offertes par notre contribution pour combiner facilement différentes heuristiques d'ordonnancement et permettre l'exploration de nouveaux ordonnanceurs de tâches.

Conclusion et perspectives

"Il faut pas respirer la compote, ça fait tousser."
– Kadoc, *Kaamelott Livre V, Le Destitué*

Les simulations de phénomènes physiques traitant des phénomènes de plus en plus complexes et de plus en plus grands en taille, les besoins associés en calcul informatique nécessitent de faire évoluer les solutions matérielles et logicielles pour permettre leur passage à l'échelle. Pour répondre à ce besoin, la complexité des supercalculateurs modernes ne fait qu'augmenter, avec désormais des machines disposant d'unités de calcul hétérogènes, des mémoires séparées qui doivent être utilisées de concert pour le calcul, etc. Actuellement, l'adaptation des optimisations d'un code de calcul pour tirer parti d'un nouveau matériel prend à peu près autant de temps que la durée de vie de ce matériel dans un supercalculateur.

Une solution à ce problème consiste à déléguer les aspects non portables du calcul à un support d'exécution. Les supports d'exécution exposent à l'application un modèle de programmation parallèle qui abstrait certains aspects de la machine au programmeur. À partir des informations spécifiées par l'application, le support d'exécution va automatiser et optimiser la gestion de ces aspects afin d'assurer à l'application une portabilité de ses performances.

Contributions

Nous nous sommes proposés dans cette thèse d'étudier le passage à l'échelle du support d'exécution à base de tâches StarPU en l'utilisant pour optimiser le solveur d'algèbre linéaire dense d'un code de simulation du CEA qui traite des cas de plusieurs millions d'inconnues.

Comme ces grands cas nécessitent beaucoup de mémoire pour s'exécuter, nous avons étudié en premier le passage à l'échelle de ce solveur en nombre de nœuds de calcul. Nous avons tout d'abord réduit le surcoût de la réplication de la soumission des tâches sur tous les processus en proposant une méthode d'élagage du graphe de tâches en fonction du processus qui le déroule. Nous montrons que cet élagage réduit considérablement le temps de soumission des tâches, et que la soumission des tâches ne limite désormais les performances du programme qu'à l'échelle de centaines de milliers de processus. Nous avons également identifié que l'utilisation des appels système allocateurs de mémoire

de Linux qui est faite par les allocateurs standards peut limiter fortement les performances du solveur et augmenter considérablement sa consommation de mémoire. Ces informations étant par contre connues de StarPU, nous proposons de gérer la réutilisation des données directement dans StarPU grâce à un cache d'allocations. Cette stratégie permet de contourner ces problèmes sans nécessiter de changements dans la stratégie d'allocation de mémoire du programme qui utilise StarPU. Le dernier point que nous avons étudié est lié à la soumission des tâches : les tâches soumises par l'application à StarPU doivent être acceptées en soumission le plus rapidement possible afin que StarPU puisse prendre de bonnes décisions d'ordonnancement et recouvrir efficacement les transferts de données par du calcul. Dans le cadre du calcul distribué, cette soumission anticipée du graphe de tâches peut poser des problèmes de surconsommation de mémoire de StarPU, qui a besoin d'allouer dès la soumission des tâches les tampons temporaires de réception des données, et donc de débordements mémoire. L'une des propriétés du modèle de programmation STF qu'implémente StarPU est que toute tâche acceptée en soumission peut être exécutée car toutes ses dépendances portent par construction sur des taĉhes précédemment soumises. Il est donc possible d'arrêter la soumission de tâches sans provoquer de blocage de l'exécution. Nous utilisons cette propriété pour proposer un mécanisme permettant de contrôler l'utilisation des ressources de calcul grâce au contrôle du flot de soumission de tâches. À travers l'étude de deux politiques de contrôle des ressources, en nombre de tâches et en mémoire, nous montrons que le contrôle de l'encombrement mémoire permet d'obtenir un bon compromis entre l'allocation anticipée de mémoire et les performances de l'application tout en garantissant l'absence de débordements de mémoire. L'ensemble de ces contributions ont permis d'exécuter ce solveur d'algèbre linéaire dense avec succès jusqu'à 144 nœuds de calcul hybrides du supercalculateur TERA-100 du CEA-DAM, soit 1 152 cœurs et 288 GPUs, et jusqu'à 3 000 cœurs de calcul sans accélérateurs GPUs. Ses performances sont de plus comparables aux références du domaine ainsi qu'au solveur développé par le CEA pour ses besoins.

Nous avons ensuite porté un solveur linéaire complexe qui traite des matrices compressées sur StarPU afin de faciliter son utilisation et son optimisation dans le code de simulation. Toutes les contributions précédentes peuvent immédiatement servir pour optimiser ce solveur, à l'exception notable du contrôle de l'encombrement mémoire car la taille des matrices compressées évolue durant l'exécution du solveur. Nous avons étendu le contrôle de mémoire de StarPU pour lui permettre de gérer les données de taille variable tout en continuant de garantir que l'exécution ne dépassera pas des limites de la mémoire disponible pour StarPU. Pour ce faire, nous introduisons des estimateurs de l'évolution des données, qui sont appelés à la soumission des tâches pour avertir StarPU que la taille de certaines données de la tâche peut changer au moment de son exécution. La taille des données est ensuite corrigée

à la terminaison des tâches, quand leur taille est connue. Le fonctionnement est similaire pour la réception de données depuis d'autres processus, qui sont de taille inconnue, et dont la taille peut être estimée à la soumission des tâches, et corrigée au moment de leur réception. Bien que cette solution fonctionne pour le solveur pour matrices compressées, il limite fortement ses performances car l'estimateur fourni par l'utilisateur est extrêmement peu précis : la seule information disponible est que la matrice ne peut pas grossir au point de redevenir dense, ce qui conduit à une surestimation de la taille des données de l'ordre du taux de compression de la matrice, qui dépasse généralement 90% pour la plupart des cas de simulation du CEA, et donc une surconstriction du flot de soumission de tâches. Nous proposons de faire une concession sur la garantie d'exécution dans les limites de la mémoire en n'estimant pas le grossissement des données locales, car l'utilisateur sait empiriquement qu'il sera faible, afin de débloquer du parallélisme supplémentaire. Nous montrons que cette solution est satisfaisante pour les cas d'utilisation du CEA, et qu'elle permet de débloquer suffisamment de parallélisme pour avoir des performances comparables au solveur pour matrices compressées développé par le CEA jusqu'à 3 000 cœurs de calcul.

Nous avons également étudié dans cette thèse le passage à l'échelle à l'intérieur d'un nœud. Dans ce contexte, nous avons identifié les ordonnanceurs de tâches comme la principale cause de surcoût du support d'exécution lorsque le nombre de cœurs de calcul au sein d'un nœud augmente. Notre contribution à ce problème ne consiste pas à proposer une nouvelle heuristique d'ordonnancement capable de passer à l'échelle tout en prenant de bonnes décisions, mais à proposer une formalisation modulaire de la structure des ordonnanceurs de tâches permettant de faciliter le développement et la combinaison d'heuristiques d'ordonnancement au sein d'un même ordonnanceur. Nous montrons à travers une étude comparative des ordonnanceurs modulaires avec leurs équivalents monolithiques que le surcoût lié à l'utilisation d'un tel ordonnanceur est négligeable, et que cette structuration des ordonnanceurs ouvre la voie vers l'élaboration d'ordonnanceurs mixtes, où plusieurs heuristiques d'ordonnancement sont utilisées de concert en fonction de l'architecture de la machine.

La programmation par tâches, vers l'exascale et au-delà

Pour atteindre l'exascale, la communauté du calcul haute performance annonce un profond changement de paradigme lié à une nouvelle évolution des architectures matérielles. Nous sommes en effet arrivés à la limite de ce qu'il est possible de faire en termes de consommation énergétique pour les supercalculateurs [64]. Les nouveaux matériels doivent donc permettre d'augmenter la puissance de calcul sans consommer plus d'énergie. Ainsi, la multiplication des

cœurs de calcul simple et de faible fréquence dans les supercalculateurs semble être la voie choisie par les constructeurs pour atteindre l'exascale, et l'architecture du nouveau numéro 1 chinois du Top500 [6] confirme cette tendance. De même, les transferts mémoire devenant trop coûteux en énergie, les constructeurs proposent à la communauté d'utiliser de nouvelles mémoires non-volatiles (NVRAM) qui apportent leur part de complexité supplémentaire à l'équation : le coût d'une écriture dans ces mémoires est tel qu'il est parfois plus rentable en termes de performance et d'énergie consommée de recalculer un résultat plutôt que de l'écrire dans ces mémoires. De nouvelles méthodes numériques cherchant à minimiser le nombre d'écritures en mémoire voient actuellement le jour dans le but de préparer ce changement de paradigme [28]. Avec cette complexité supplémentaire de programmation et d'optimisation apportée par ces nouveaux matériels, il devient évident que le besoin d'abstraction des machines pour le programmeur d'applications scientifiques ne va qu'augmenter.

Dans cette thèse, nous avons montré que la soumission séquentielle des tâches, vue comme une limite au passage à l'échelle du modèle STF au départ, apparaît en fait comme un avantage : comme la soumission des tâches est dissociée de leur exécution réelle, il est possible d'imposer un ordre séquentiel à la soumission des tâches afin de contrôler l'utilisation des ressources de calcul sans gêner de manière excessive l'exécution des tâches, comme nous avons montré en Section 2.2. Nous avons montré dans cette thèse comment utiliser ce mécanisme pour contrôler l'utilisation des ressources de calcul au sein d'un processus, mais il est également possible de l'utiliser pour équilibrer dynamiquement la consommation de ces ressources entre les processus. En effet, comme la soumission des tâches est généralement plus rapide que leur exécution, il est possible de reconsidérer l'équilibrage de charge entre les processus à la soumission des tâches. L'idée est la suivante : lorsque StarPU arrête la soumission de tâches, il peut déclencher l'appel à un équilibreur de charge lorsqu'une certaine fraction des tâches acceptées en soumission sur chaque processus se sont exécutées (80% par exemple). Cet équilibreur de charge va alors récolter les informations de charge depuis les autres processus et appliquer une redistribution de la charge pour les tâches *qui restent à soumettre*. L'avantage majeur de cette méthode d'équilibrage de charge dynamique distribué est qu'il est possible de recouvrir le temps de calcul de l'équilibreur de charge ainsi que les synchronisations nécessaires entre processus pour l'échange des données de charge par du calcul de l'application (la fraction de tâches qu'il reste à exécuter). L'utilisation d'un tel mécanisme nécessite donc de trouver un compromis entre la fraction de tâches dont il faut attendre l'exécution pour que l'équilibreur de charge prenne de bonnes décisions, et la fraction de tâches nécessaire à conserver pour recouvrir correctement le calcul de l'équilibreur de charge par le calcul du programme.

Bien que la soumission séquentielle des tâches offre un certain potentiel pour optimiser automatiquement l'utilisation des ressources de calcul, cette

séquentialité reste une limite forte de ce modèle pour son passage à l'échelle. Il reste des possibilités pour optimiser l'élagage du graphe de tâches, notamment par l'utilisation de techniques de compilation pour directement élaguer les nids de boucle du programme, mais cette soumission limitera nécessairement les performances du programme à un certain moment. À l'inverse, le modèle à flot de données permet un excellent passage à l'échelle en nombre d'unités de calcul, mais ne peut pas contrôler l'utilisation des ressources durant l'exécution par conception. Par conséquent, nous pensons que l'avenir de la programmation par tâches, pour l'exascale et au-delà, se trouve dans la combinaison du modèle à flot de contrôle séquentiel, pour contrôler l'utilisation des ressources des machines à la soumission des tâches, et du modèle de programmation à flot de données parallèle, pour que l'exécution des tâches soit efficace et passe à l'échelle. Des travaux sont en cours pour permettre à l'utilisateur de StarPU d'implémenter des macro-tâches, appelées *bulles*, qui déroulent un sous-graphe de tâches qui peut alors être exécuté soit par StarPU, soit par un autre support d'exécution à base de tâches. Cette solution ouvre la voie vers une collaboration de ces deux modèles dans StarPU : la soumission séquentielle des macro-tâches permet de contrôler à gros grain l'utilisation des ressources de calcul, et l'exécution à grain fin des sous-tâches est dirigée par un support d'exécution à flot de données efficace et qui passe à l'échelle.

Bibliographie

[1] Chameleon : A dense linear algebra software for heterogeneous architectures. https://project.inria.fr/chameleon/. Consulté le 14/10/2016.

[2] PlaFRIM : Plateforme Fédérative pour la Recherche en Informatique et Mathématiques. https://www.plafrim.fr. Consulté le 14/10/2016.

[3] StarPU : A Unified Runtime System for Heterogeneous Multicore Architectures. http://starpu.gforge.inria.fr/. Consulté le 14/10/2016.

[4] TERA-100 - bull bullx super-node s6010/s6030.

[5] Workshop EM ISAE 2016. http://websites.isae.fr/workshop-em-isae-2016. Consulté le 14/10/2016.

[6] Liste du TOP500 pour Juin 2016. http://www.top500.org/lists/2016/06/, 2016. Consulté le 16/09/2016.

[7] E. Agullo, G. Bosilca, B. Bramas, C. Castagnede, O. Coulaud, E. Darve, J. Dongarra, M. Faverge, N. Furmento, L. Giraud, X. Lacoste, J. Langou, H. Ltaief, M. Messner, R. Namyst, P. Ramet, T. Takahashi, S. Thibault, S. Tomov et I. Yamazaki : Matrices Over Runtime Systems at Exascale. *In High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion*, pages 1330–1331, Nov 2012.

[8] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Hatem Ltaief, Raymond Namyst, Samuel Thibault et Stanimire Tomov : A hybridization methodology for high-performance linear algebra software for gpus. *GPU Computing Gems, Jade Edition*, 2:473–484, 2011.

[9] Emmanuel Agullo, Bérenger Bramas, Olivier Coulaud, Eric Darve, Matthias Messner et Toru Takahashi : Task-based FMM for multicore architectures. *SIAM Journal on Scientific Computing*, 36(1):C66–C93, 2014.

[10] Emmanuel Agullo, Alfredo Buttari, Abdou Guermouche et Florent Lopez : Implementing multifrontal sparse solvers for multicore architectures with Sequential Task Flow runtime systems. Rapport de recherche IRI/RT–2014-03–FR, IRIT, Université Paul Sabatier, Toulouse, novembre 2014.

[11] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek et Stanimire Tomov : Numerical linear algebra on emerging architectures : The PLASMA and MAGMA projects. *In Journal of Physics : Conference Series*, volume 180, page 012037. IOP Publishing, 2009.

[12] Edward Anderson, Zhaojun Bai, Jack Dongarra, Anne Greenbaum, Alan McKenney, Jeremy Du Croz, Sven Hammerling, James Demmel, C Bischof et Danny Sorensen : LAPACK : A portable linear algebra library for high-performance computers. *In Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, pages 2–11. IEEE Computer Society Press, 1990.

[13] Cédric Augonnet, Olivier Aumage, Nathalie Furmento, Raymond Namyst et Samuel Thibault : StarPU-MPI : Task Programming over Clusters of Machines Enhanced with Accelerators. *In Recent Advances in the Message Passing Interface*, pages 298–299. Springer, 2012.

[14] Cédric Augonnet, Jérôme Clet-Ortega, Samuel Thibault et Raymond Namyst : Data-aware task scheduling on multi-accelerator based platforms. *In Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on*, pages 291–298. IEEE, 2010.

[15] Cédric Augonnet, Samuel Thibault et Raymond Namyst : Automatic calibration of performance models on heterogeneous multicore architectures. *In European Conference on Parallel Processing*, pages 56–65. Springer, 2009.

[16] Cédric Augonnet, Samuel Thibault, Raymond Namyst et Pierre-André Wacrenier : StarPU : A Unified Platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation : Practice and Experience*, 23(2):187–198, 2011.

[17] Luciano Porto Barreto, Rémi Douence, Gilles Muller et Mario Südholt : Programming OS schedulers with domain-specific languages and aspects : new approaches for OS kernel engineering. *In Proceedings of the 1st AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, pages 1–6. Citeseer, 2002.

[18] M. Bebendorf et S. Kunis : Recompression Techniques for Adaptive Cross Approximation. *Journal of Integral Equations and Applications*, 21(3):331–357, 2009.

[19] Mario Bebendorf : Approximation of boundary element matrices. *Numerische Mathematik*, 86(4):565–589, 2000.

[20] Mario Bebendorf : Approximation of Boundary Element Matrices. *Numerische Mathematik*, 86:565–589, 2000.

[21] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall et Yuli Zhou : Cilk : An efficient multithreaded runtime system. *Journal of parallel and distributed computing*, 37(1):55–69, 1996.

[22] Robert D Blumofe et Charles E Leiserson : Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.

[23] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault et 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 et Jack Dongarra : From serial loops to parallel execution on distributed systems. *In European Conference on Parallel Processing*, pages 246–257. Springer, 2012.

[25] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Jakub Kurzak, Piotr Luszczek, Stanimire Tomov et Jack J Dongarra : Scalable dense linear algebra on heterogeneous hardware. *Advances in Parallel Computing*, 2013.

[26] Zoran Budimlić, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach *et al.* : Concurrent collections. *Scientific Programming*, 18(3-4):203–217, 2010.

[27] Alfredo Buttari, Julien Langou, Jakub Kurzak et Jack Dongarra : A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures. *Parallel Computing*, 35(1):38–53, 2009.

[28] Erin Carson, James Demmel, Laura Grigori, Nicholas Knight, Penporn Koanantakool, Oded Schwartz et Harsha Vardhan Simhadri : Write-avoiding algorithms. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-163*, 2015.

[29] Jaeyoung Choi, James Demmel, Inderjiit Dhillon, Jack Dongarra, Susan Ostrouchov, Antoine Petitet, Ken Stanley, David Walker et R Clinton Whaley : ScaLAPACK : A portable linear algebra library for distributed memory computers—Design issues and performance. *In Applied Parallel Computing Computations in Physics, Chemistry and Engineering Science*, pages 95–106. Springer, 1996.

[30] Y. Choquet-Bruhat : *Géométrie différentielle et systèmes extérieurs.* Dunod, 1968.

[31] Lyndon Clarke, Ian Glendinning et Rolf Hempel : The MPI message passing interface standard. *In Programming environments for massively parallel distributed systems*, pages 213–218. Springer, 1994.

[32] E. G. Coffman, M. Elphick et A. Shoshani : System deadlocks. *ACM Comput. Surv.*, 3(2):67–78, juin 1971.

[33] Michel Cosnard et Michel Loi : Automatic task graph generation techniques. *In System Sciences, 1995. Proceedings of the Twenty-Eighth Hawaii International Conference on*, volume 2, pages 113–122. IEEE, 1995.

[34] X Cray : Mp computer systems, cray x-mp series mainframe reference manual. hr-0032, cray research. *Inc., Mendota Heights, Minn*, 1982.

[35] Anthony Danalis, George Bosilca, Aurelien Bouteiller, Thomas Herault et Jack Dongarra : PTG : an abstraction for unhindered parallelism. *In Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC), 2014 Fourth International Workshop on*, pages 21–30. IEEE, 2014.

[36] Edsger W Dijkstra : Cooperating sequential processes. *In The origin of concurrent programming*, pages 65–138. Springer, 1968.

[37] A Dominek, H Shamanski, R Wood et R Barger : A useful test body. *In Proc. Antenna Measurement Techniques Assoc*, volume 24, 1986.

[38] Jack J Dongarra, James R Bunch, Cleve B Moler et Gilbert W Stewart : *LINPACK users' guide.* Siam, 1979.

[39] Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell et Judit Planas : Ompss : a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.

[40] Paolo Gai, Luca Abeni, Massimiliano Giorgi et Giorgio Buttazzo : A new kernel approach for modular real-time systems development. *In*

Real-Time Systems, 13th Euromicro Conference on, 2001., pages 199–206. IEEE, 2001.

[41] Michael R Garey et David S Johnson : *Computers and Intractability : A Guide to the Theory of NP-Completeness*, volume 29. wh freeman New York, 2002.

[42] M. Goli, M. T. Garba et H. Gonz´lezVélez : Streaming Dynamic Coarse-Grained CPU/GPU Workloads with Heterogeneous Pipelines in FastFlow. *In High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pages 445–452, June 2012.

[43] Lars Grasedyck : Adaptive Recompression of $\mathcal{H}$-matrices for BEM. *Computing*, 74:205–223, 2005.

[44] Leslie Greengard et Vladimir Rokhlin : A fast algorithm for particle simulations. *Journal of computational physics*, 73(2):325–348, 1987.

[45] John A Gunnels, Fred G Gustavson, Greg M Henry et Robert A Van De Geijn : FLAME : Formal linear algebra methods environment. *ACM Transactions on Mathematical Software (TOMS)*, 27(4):422–455, 2001.

[46] Fred G Gustavson : High-performance linear algebra algorithms using new generalized data structures for matrices. *IBM Journal of Research and Development*, 47(1):31–55, 2003.

[47] Torsten Hoefler, James Dinan, Darius Buntinas, Pavan Balaji, Brian Barrett, Ron Brightwell, William Gropp, Vivek Kale et Rajeev Thakur : Mpi+ mpi : a new hybrid approach to parallel programming with mpi plus shared memory. *Computing*, 95(12):1121–1136, 2013.

[48] ISO/IEC : ISO International Standard ISO/IEC 14882 :2012(E) – Programming language C++ [Working Draft], 2012.

[49] Laxmikant V Kale et Sanjeev Krishnan : *CHARM++ : a portable concurrent object oriented system based on C++*, volume 28. ACM, 1993.

[50] David M Kunzman et Laxmikant V Kalé : Programming heterogeneous clusters with accelerators using object-based programming. *Scientific Programming*, 19(1):47–62, 2011.

[51] Xavier Lacoste, Mathieu Faverge, George Bosilca, Pierre Ramet et Samuel Thibault : Taking advantage of hybrid systems for sparse

direct solvers via task-based runtimes. *In Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 29–38. IEEE, 2014.

[52] Julia L Lawall, Gilles Muller et Luciano Porto Barreto : Capturing OS expertise in an Event Type System : the Bossa experience. *In Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 54–61. ACM, 2002.

[53] Chuck L Lawson, Richard J. Hanson, David R Kincaid et Fred T. Krogh : Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3):308–323, 1979.

[54] Charles E Leiserson : The Cilk++ concurrency platform. *The Journal of Supercomputing*, 51(3):244–257, 2010.

[55] Sidi Ahmed Mahmoudi, Pierre Manneback, Cédric Augonnet, Samuel Thibault *et al.* : Traitement d'images sur architectures parallèles et hétérogènes. *Technique et Science Informatiques*, 31(8-10):1183–1203, 2012.

[56] Víctor Martínez, David Michéa, Fabrice Dupros, Olivier Aumage, Samuel Thibault, Hideo Aochi et Philippe OA Navaux : Towards seismic wave modeling on heterogeneous many-core architectures using task-based runtime system. *In Computer Architecture and High Performance Computing (SBAC-PAD), 2015 27th International Symposium on*, pages 1–8. IEEE, 2015.

[57] John Nickolls, Ian Buck, Michael Garland et Kevin Skadron : Scalable parallel programming with cuda. *Queue*, 6(2):40–53, mars 2008.

[58] CUDA Nvidia : CUBLAS library. *NVIDIA Corporation, Santa Clara, California*, 15:27, 2008.

[59] OpenMP ARB : OpenMP application program interface version 3.0, 2008.

[60] OpenMP ARB : OpenMP application program interface version 4.5, 2015.

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

[62] Martin C. Rinard, Daniel J. Scales et Monica S. Lam : Jade : A high-level, machine-independent language for parallel programming. *Computer*, 26(6):28–38, 1993.

[63] John Shaeffer : Direct Solve of Electrically Large Integral Equations for Problem Size to 1M Unknowns. *IEEE Transactions on Antennas and Propagation*, 56(8), Août 2008.

[64] John Shalf, Sudip Dosanjh et John Morrison : Exascale Computing Technology Challenges. *In* JoséM.LaginhaM. Palma, Michel Daydé, Osni Marques et JoãoCorreia Lopes, éditeurs : *High Performance Computing for Computational Science – VECPAR 2010*, volume 6449 de *Lecture Notes in Computer Science*, pages 1–25. Springer Berlin Heidelberg, 2011.

[65] A. Sodani : Knights landing (knl) : 2nd generation intel xeon phi processor. *In 2015 IEEE Hot Chips 27 Symposium (HCS)*, pages 1–24, Aug 2015.

[66] Clemens Szyperski : Component technology : what, where, and how ? *In Proceedings of the 25th international conference on Software engineering*, pages 684–693. IEEE Computer Society, 2003.

[67] Enric Tejedor, Montse Farreras, David Grove, Rosa M Badia, Gheorghe Almasi et Jesus Labarta : A high-productivity task-based programming model for clusters. *Concurrency and Computation : Practice and Experience*, 24(18):2421–2448, 2012.

[68] William Thies, Michal Karczmarek et Saman Amarasinghe : StreamIt : A language for streaming applications. *In International Conference on Compiler Construction*, pages 179–196. Springer, 2002.

[69] Martin Tillenius : SuperGlue : A shared memory framework using data versioning for dependency-aware task-based parallelization. *SIAM Journal on Scientific Computing*, 37(6):C617–C642, 2015.

[70] Haluk Topcuoglu, Salim Hariri et Min-you Wu : Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE transactions on parallel and distributed systems*, 13(3):260–274, 2002.

[71] Jeffrey D. Ullman : NP-complete scheduling problems. *Journal of Computer and System sciences*, 10(3):384–393, 1975.

[72] Asim YarKhan : *Dynamic task execution on shared and distributed memory architectures.* Thèse de doctorat, University of Tenessee, 2012.

[73] Keshong Zhao, Marinos N. Vouvakis et Jin-Fa Lee : The Adaptive Cross Approximation Algorithm for Accelerated Method of Moments Computations of EMC Problems. *IEEE Transactions on Electromagnetic Compatibility*, 47(4), Novembre 2005.

Publications

Conférences nationales avec actes

[74] Marc Sergent et Simon Archipoff : Modulariser les ordonnanceurs de tâches : une approche structurelle. *In* Pascal Felber, Laurent Philippe, Etienne Riviere et Arnaud Tisserand, éditeurs : *ComPAS 2014 : conférence en parallélisme, architecture et systèmes*, Neuchâtel, Suisse, avril 2014.

Conférences internationales sans actes

[75] Emmanuel Agullo, Olivier Aumage, Mathieu Faverge, Nathalie Furmento, Florent Pruvost, Marc Sergent et Samuel Thibault : Overview of Distributed Linear Algebra on Hybrid Nodes over the StarPU Runtime. *In SIAM Conference on Parallel Processing for Scientific Computing (SIAM PP 2014)*, Portland, Oregon, États-Unis, février 2014.

[76] Marc Sergent, David Goudin, Samuel Thibault et Olivier Aumage : Controlling the Memory Subscription of Distributed Applications with a Task-Based Runtime System. *In SIAM Conference on Parallel Processing for Scientific Computing (SIAM PP 2016)*, pages 318 – 327, Paris, France, avril 2016.

Colloques internationaux avec actes

[77] Marc Sergent, David Goudin, Samuel Thibault et Olivier Aumage : Controlling the Memory Subscription of Distributed Applications with a Task-Based Runtime System. *In 21st International Workshop on High-Level Parallel Programming Models and Supportive Environments*, Chicago, États-Unis, mai 2016.

Colloques internationaux sans actes

[78] Emmanuel Agullo, Olivier Aumage, Mathieu Faverge, Nathalie Furmento, Florent Pruvost, Marc Sergent et Samuel Thibault : Harnessing clusters of hybrid nodes with a sequential task-based programming model. *In International Workshop on Parallel Matrix Algorithms and Applications (PMAA 2014)*, Lugano, Suisse, juillet 2014.

Posters internationaux

[79] E. Agullo, O. Aumage, G. Bosilca, B. Bramas, A. Buttari, O. Coulaud, E. Darve, J. Dongarra, M. Faverge, N. Furmento, L. Giraud, A. Guermouche, J. Langou, F. Lopez, H. Ltaief, S. Pitoiset, F. Pruvost, M. Sergent, S. Thibault et S. Tomov : Poster : Matrices over runtime systems at exascale. *In High Performance Computing, Networking, Storage and Analysis (SCC), 2015 SC Companion*, Austin, États-Unis, Nov 2015.

Rapports de recherche

[80] Emmanuel Agullo, Olivier Aumage, Mathieu Faverge, Nathalie Furmento, Florent Pruvost, Marc Sergent et Samuel Thibault : Achieving High Performance on Supercomputers with a Sequential Task-based Programming Model. Research Report RR-8927, Inria Bordeaux Sud-Ouest ; Bordeaux INP ; CNRS ; Université de Bordeaux ; CEA, juin 2016.