Amélioration des performances via un parallélisme multi-niveaux en maillage sur un code de CFD en maillages non structurés

Benjamin Lorendeau

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

par **Benjamin Lorendeau**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Amélioration des performances via un parallélisme multi-niveaux sur un code CFD en maillages non structurés

---

Date de soutenance : 16 Décembre 2019

Devant la commission d'examen composée de :

Pascal Bruel	Chargé de recherche, CNRS	rapporteur
Christine Eisenbeis .	Directrice de Recherche, Inria	rapporteure
Yvan Fournier	Ingénieur Chercheur Expert, EDF .	co-encadrant
Emmanuel Jeannot .	Directeur de recherche, Inria	directeur de thèse
Raymond Namyst . .	Professeur, Université de Bordeaux	examinateur
Gabriel Staffelbach	Chercheur senior, CERFACS	examinateur

– 2019 –

Résumé L'évolution constante ainsi que la complexification qui s'ensuit des architectures matérielles obligent les personnes développant des codes de simulations scientifiques à une mise à jour perpétuelle de leur logiciel et de leurs connaissances afin de maintenir une bonne exploitation des performances de ces plateformes de calcul. L'hétérogénéité récente de ces plateformes et la multiplication du nombres de coeurs par machine posent notamment une nouvelle problématique aux logiciels largement parallélisés via l'utilisation de la bibliothèque d'échange de message MPI et qui se voient perdre en performance au fur et à mesure de l'augmentation du parallélisme local à exploiter. Pour *Code_Saturne*, la transition vers l'exploitation d'un parallélisme hybride via l'association de MPI avec OpenMP (en parallélisme de boucles) peine à s'imposer et ne résout pas les soucis de passage à l'échelle causé par un manque d'équilibrage de charge. Pour répondre à ce problème (ainsi que ceux de la portabilité d'un code et sa capacité à exploiter les plateformes hétérogènes), l'utilisation de la programmation par tâches a récemment gagné en popularité à travers trois méthodes: parallélisme par boucle, tâches soumises de manière séquentielle (STF, flux de tâche séquentiel) ou paramétrique (PTG: graphe de tâches paramétré). Cette thèse se concentre sur l'effort de transition de *Code_Saturne*, un code de simulation en mécanique des fluides sur maillages non structurés (algèbre linéaire creuse) vers l'utilisation d'un support exécutif à base de tâches. Pour se faire, une évaluation des deux derniers paradigmes est proposée via l'utilisation des supports StarPU et PaRSEC. Nous établissons leur intérêt pour l'évolution d'un code tel que *Code_Saturne* comparé à l'utilisation de MPI+OpenMP et nous les comparons sur des questions de complexité et de performance avec l'existant dans *Code_Saturne*. Cette thèse est une collaboration entre le département MFEE de EDF R&D et l'équipe Tadaam qui met en commun Inria Bordeax Sud-Ouest, le LaBRI, l'Université de Bordeaux 1 et Bordeaux INP.

Mots-clés CFD, maillages non structurés, graphe de tâches, PTG, STF

Laboratoire d'accueil / Hosting Laboratory Inria Bordeaux Sud-Ouest, 200 Avenue de la Vieille Tour, 33405 Talence, France

Title Improving performance of a CFD code with unstructured meshes through multi level parallelism

Abstract The ever-evolving and ultimately complexification of computer architectures enforce scientific developpers to maintain both their software and technical knowledge in order to pertain their ability to fully exploit recent high performance computing platforms. Moreover the birth of heterogeneous plateforms and the large increase in core number per processor lead developpers of software parallelized through distributed paralleslim to see a small but consistent decrease of performance on recent plateforms. For *Code_Saturne*, the transission from a full MPI code to a MPI+OpenMP code is not fully satisfying and cannot adress the scalability issue that a lack of load balancing causes. This load balancing issue is a complex problem to solve for a code such as *Code_Saturne* where partitionning is a NP problem and must data being indexed due to the unstructured nature of the meshes used. To solve this kind of issue and many others (such as portability and heterogeneity), the use of a task based runtime recently won popularity through three different ways: loop parallelism, sequential submission of task (STF) or parameterized task graph (PTG). This thesis is focused on the transition effort of *Code_Saturne*, a computational fluid dynamic (CFD) code with unstructured meshes (sparse linear algebra) from a mostly distributed parallelism to the integration of a task based runtime. In order to do this, we investigated the STF and PTG paradigms through the use of StarPU and PaRSEC in *Code_Saturne*. We demonstrate their advantages over a solution such that MPI + OpenMP as well as compare their use in terms of complexity and performance with the existing solutions in *Code_Saturne*. This thesis was made through the collaboration between the MFEE department of EDF R&D and the Tadaam research team, grouping Inria Bordeaux Sud-Ouest, the LaBRI, University of Bordeaux 1 and Bordeaux INP.

Keywords CFD, unstructured meshes, task graph, PTG, STF

Laboratoire d'accueil / Hosting Laboratory Inria Bordeaux Sud-Ouest, 200 Avenue de la Vieille Tour, 33405 Talence, France

Amélioration des performances via un parallélisme multi-niveaux sur un code CFD en maillages non structurés

Benjamin Lorendeau

9 avril 2020

À mes grands-pères, Georges et Victor

Remerciements

J'ai toujours su que la phase d'écriture des remerciements serait avant tout un exercice de style. Comme beaucoup de choses d'ailleurs. Bien sûr, il n'en est ainsi que lorsque l'on croit la difficulté comme étant majoritairement dans l'aptitude à provoquer au lecteur un quelconque plaisir. Celui-ci se trouverait surement de par la lecture de ces mots que l'on vient de poser doigt après doigt en frappant un clavier éreinté par 3 années de thèse (ou 5, oui bon.). Or je crois bien que j'avais tort, car il m'est aujourd'hui bien difficile de formuler ces derniers. Certes il fait beau dehors mais cette excuse-là ne marche bien que pour aujourd'hui. Nous voilà maintenant confinés pour la troisième semaine, et ces quelques lignes toujours à l'état d'embryon. Mais soit, allons-y !

Je pense bien évidemment à mon jury dans son entièreté et plus particulièrement à mes deux rapporteurs Mr Bruel et Mme Eisenbeis pour leur bienveillance et la qualité de leurs retours. Yvan et Emmanuel, mes deux superviseurs, ont su me supporter 2 années supplémentaires et m'encourager à finir ma rédaction malgré un contexte compliqué, une chose pour laquelle je leur resterai éternellement reconnaissant. Merci ! Ces cinq années n'ont pas été simples et votre patience et compréhension m'a été très appréciable. J'espère aussi que les potentiels cheveux blancs dont la faute serait à ma charge vous siéent à merveille, parce que sur moi c'est pas glorieux. J'ai appris à vos côtés beaucoup de choses, et ne cesserai d'apprécier l'expérience et la sagesse que vous avez partagées avec moi.

Ces deux dernières années n'ont pas été difficiles que pour cette thèse et ses acteurs, et je me dois d'exprimer toute la gratitude que j'ai envers Sarah, ma compagne, et Naima sa mère, qui ont su supporter mes pérégrinations vers toute autre tâche qui pouvait me libérer un tant soit peu de l'angoisse croissante que me procurait alors la rédaction de ce manuscrit. Merci à vous deux. Les mots ne suffiront pas pour exprimer les bienfaits de vos actes et de votre patience.

Je remercie aussi l'ensemble des membres de l'équipe de StarPU et de ParSEC et plus largement de feu Runtime Inria ainsi que MFEE EDF R&D. Un merci particulier à Mathieu Faverge pour son temps passé avec moi et son expertise de ParSEC.

Ces remerciements ne sauraient être complets sans une liste forcement incomplète de tous ceux dont j'ai pu croiser le chemin. Les singes de #starpu****er en font bien évidemment partie. Chrishh, Rools, Sylvain, Paulo, Nico, Marc, cela a été un plaisir de partager avec vous une partie de ces dernières années. Antoine, mon vieux binôme d'avant et de maintenant. Le parc EDF de Chatou a lui aussi eux quelques belles cuvées, Xavier toujours là pour picoler, Pierre et son amour du beau mot mais aussi Antoine, Ricardo, Gaétan et Clément. Et pour finir, mes partenaires de crimes, Sarah et William, the godfather. Merci à vous tous d'avoir donné de la couleur à ces dernières années.

Introduction

Arriver à comprendre les phénomènes physiques qui nous entourent est un sujet vaste et fascinant. Le monde scientifique s'attache pour cela à modéliser ces phénomènes par le biais d'équations. L'arrivée de l'informatique et du calcul parallèle ont permis d'accéder à une compréhension plus large de ces phénomènes et de faire évoluer les modèles pour en affiner leur précision. Pour EDF, la simulation en mécanique des fluides (appelée CFD, computational fluid dynamics) est un domaine d'étude indispensable à la pérennité de ses installations et de ses futurs projets (la production d'électricité par les biais nucléaire, d'un barrage à eau ou d'une éolienne, provient toujours d'un fluide mettant en mouvement une pièce mécanique, générant ainsi une énergie facilement transformable en électricité). La CFD permet donc d'étudier et d'améliorer les rendements des diverses solutions d'EDF et d'inventer celles de demain. Cependant l'établissement de ces simulations ne peut se faire que par la mise en commun de deux expertises, celle des modèles physiques et celle du développement scientifique (ou calcul haute performance, on regroupe dans ce terme les spécificités de développement qui a trait a la résolution d'équations). Dans ce cadre, le développeur scientifique se doit de faire des choix architecturaux et d'implémentations lui permettant d'obtenir une solution à ses équations en un temps raisonnable sans y sacrifier ses critères de qualités. Certains de ses choix feront partie des fondations du logiciel de simulation qui en découlera et seront difficiles à changer. On peut lister parmi ceux-ci la façon de représenter l'objet de notre simulation. Cette représentation influera sur la qualité des résultats obtenus mais aussi sur le coût en calcul de la solution. La finalité de ces choix peut être appréhendée à travers le prisme de la pérennité d'un code scientifique : qu'adviendra-t-il de celui-ci dans 20 ans ?

Cette question peut sembler banale si elle ne prenait pas en compte toute la complexité des évolutions matérielles auxquelles l'informatique fait face continuellement. Bien avant que l'augmentation du nombre de transistors par processeur ne commence à ralentir et ne pousse à l'invention des processeurs multi-cœurs, l'utilisation d'une multitude de machines pour répartir un calcul autrement trop long ou trop conséquent (en donnée) avait déjà largement impacté le monde du développement scientifique : le calcul distribué était né. Depuis, les processeurs se sont fait multi-cœurs, et leur mémoire rendue accessible de manière globale mais non uniforme (NUMA). Enfin, les branches les plus gourmandes de l'informatique se sont orientées vers les architectures massivement parallèles telles que les GPGPUs ou les architectures Many Cores toujours dans un but d'obtenir de meilleures performances.

Si l'on se positionne du point de vue d'un code scientifique fort d'une vingtaine d'années, on se retrouve face à un éventail de technologies à la puissance théorique très grande mais aussi bien plus complexe à exploiter qu'auparavant. Pour *Code_Saturne* (dont la première version date de 1997 même si le code est basé sur un code plus ancien), ce constat se traduit par une observation simple : la multiplication du nombre d'unité de calcul tend à l'augmentation de l'écart entre les performances théoriques et celles obtenues sur la plateforme [1]. De plus, le panel de solutions matérielles est large (GPGPUs aux architectures et bibliothèque de développement en constante évolution, architectures

1. En 2010, le cluster Ivanoe était fait de processeurs à 6 cœurs de 2.93GHz. En 2014, Porthos plus que doublait le nombre de cœurs par processeurs (14 au lieu de 6) mais avec une fréquence de 2.6GHz. Ce type d'évolution qui n'est pas spécifique à EDF, pénalise tout logiciel n'ayant pas une parallélisation optimale car la puissance par cœur baisse légèrement.

Many Cores telles que le Xeoon Phi à mi chemin entre un CPU et un GPGPU), limitant les efforts possibles et souhaitables à ceux dont la portabilité (et pérennité) est garantie. Cette problématique est une préoccupation commune à l'ensemble de la communauté du calcul scientifique et fait l'objet de nombreux travaux sous la formulation de "passage à l'échelle". Pour *Code_Saturne*, la problématique de ce passage à l'échelle née de par l'utilisation de la norme d'échange de donnée MPI (message passing interface, dont les bibliothèques l'implémentant les plus connues sont OpenMPI ou IntelMPI notamment). Utilisée de manière à ce que chaque cœur d'une machine soit doté d'un processus MPI, l'augmentation du nombre de cœurs – que ce soit par l'intermédiaire de l'augmentation du nombre de machines ou par l'utilisation de cartes accélératrices – implique une augmentation du nombre de processus échangeant des données et ainsi d'une augmentation des points de contentions dues aux synchronisations inter-processus mises en place pour forcer la cohérence des calculs. Cette problématique fait appel à une autre notion : l'équilibrage de charge. Un manque d'équilibrage induira des attentes aux points de synchronisation (les unités de calculs ayant fini devant alors attendre les autres), réduisant l'efficacité de notre solution en diminuant la charge moyenne de nos processeurs.

Parmi les solutions possibles à ces problématiques (équilibrage de charge et passage à l'échelle), l'utilisation de la programmation basée sur le paradigme de tâche offre sur le papier de nombreux avantages : en représentant les calculs comme étant les sommets d'un graphe et les échanges de données ou synchronisation comme étant ses arrêtes, on fournit une représentation qui permet à l'utilisateur d'exprimer l' asynchronisme de ses algorithmes, ce qui permettrait –quand les conditions sont réunies– de maximiser le potentiel en parallélisme exploitable d'un algorithme. En mettant en place cette stratégie, on facilite l'implémentation de mécanismes permettant de régler ou de contourner la problématique de l'équilibrage de charge : du vol de tâche, du recouvrement des tâches nécessitant une synchronisation et du recouvrement des communications par des calculs par exemple. Enfin en jouant sur la granularité des tâches on modifie la fréquence à laquelle la partie décisionnelle reprend le contrôle et on augmente les possibilités d'activer ces mécanismes.

Cette réflexion a été la cible de nombreuses propositions et a vu la naissance de solution telle que Cilk, Intel TBB ou plus récemment StarPU, PaRSEC, StarSS. En mettant en pratique cette représentation d'un algorithme avec une bibliothèque responsable d'ordonnancer les tâches et de gérer les échanges de données, on obtient un outil de développement à la fois efficace sur la vitesse de développement et exploitant au maximum les capacités de la machine tout en extrayant en partie cette préoccupation au développeur.

On s'attache dans cette thèse à apporter des éléments de réponse quant aux axes de développement possibles d'un code de simulation en mécanique des fluides dans un milieu industriel et ayant atteint la limite des performances atteignables via les solutions logicielles qu'il utilise sans entacher la portabilité du code en question. Plus particulièrement, les travaux présentés ci-après confrontent l'amélioration des performances de *Code_Saturne* via des travaux sur la partie algorithmique de *Code_Saturne* à l'utilisation de la programmation via le paradigme de tâche et cela par le biais de deux supports exécutifs, PaRSEC et StarPU. Les deux supports se distinguent par une approche différente quant à la manière d'exprimer un algorithme sous forme de graphe de tâches, et l'on s'attache à évaluer la compatibilité de ces deux approches avec les spécificités d'un code de simulation en mécanique des fluides avec maillages non structurés tel que *Code_Saturne*.

La première partie présente les travaux algorithmiques autour de *Code_Saturne* et notamment une nouvelle approche pour la reconstruction des gradients qui favorise le calcul parallèle tout en augmentant la quantité calcul (avec l'hypothèse que cette augmentation bénéficiera aux nouvelles architectures massivement parallèles). Ces travaux facilitent en deuxième partie du manuscrit, l'étude de la reconstruction des gradients (un calcul en matrice creuse et à très faible intensité) via PaRSEC et StarPU. On apporte dans cette partie une revue des bénéfices de chacune des deux solutions lorsqu'il s'agit de son utilisation pour un code industriel de simulation en mécanique des fluides tel que *Code_Saturne*.

Ces travaux ont fait l'objet d'une présentation en conférence à ParCFD en 2017 à Glasgow

(Écosse) ainsi que d'un article dans le journal Computers & Fluids 173 [2]).

2. (DOI : 10.1016/j.compfluid.2018.03.076

Première partie

Évolutions algorithmiques

1 Parallélisme et mécanique des fluides : Introduction

La simulation numérique des écoulements de fluides fait appel à de nombreuses notions mathématiques et informatiques. On s'attache ici à énumérer une partie de ces dernières afin de proposer une vue d'ensemble des paramètres qui peuvent nuire aux performances de *Code_Saturne* notamment et qui permettront aussi la compréhension des sous sections suivantes. On présente ensuite l'algorithme de reconstruction de gradients sur lequel se basera une partie de nos expériences.

I Présentation générale de *Code_Saturne*

Code_Saturne [10, 53] est un logiciel libre de simulation informatique en mécanique des fluides. Développé depuis 1997 par la division Recherche et Développement d'EDF, il a été placé sous la licence GNU GPL en mars 2007.

Basé sur une approche volumes finis co-localisés qui accepte des maillages de type non structuré conforme ou non conforme et contenant tout type d'élément (tétraèdre, hexaèdre, pyramide, polyèdre quelconque...), *Code_Saturne* permet de modéliser les écoulements incompressibles ou dilatables, avec ou sans turbulence ou transfert de chaleur. Des modules dédiés sont disponibles pour des physiques particulières comme le transfert de chaleur par rayonnement, la combustion (gaz, charbon pulvérisé, fioul lourd…), la magnéto-hydrodynamique, les écoulements compressibles, les écoulements polyphasiques (approche Euler/Lagrange avec couplage inverse), ou bien des extensions à des applications spécifiques (écoulements atmosphériques par exemple).

II Parallélisme et Mécanique des fluides numérique

Lorsque l'on effectue une simulation numérique, on construit une solution approchée de notre problème dans un espace continu en s'appuyant sur une représentation discrète. Dans ce cadre, on parle de discrétisation pour la décomposition en formes géométriques simples de l'espace continue initial, tandis que l'on parle de modélisation pour la définition des différents aspects numériques propres à cette simulation (spécificité des matériaux et fluides simulés, conditions initiales et aux limites, et cetera).

II.1 Parallélisme, limites intrinsèques

Parmi les concepts importants du parallélisme, un est particulièrement important à rappeler. La loi d'Amdhal introduite en 1967 définit ci-dessous l'idée d'une borne inférieure des gains en performances que l'on puisse espérer obtenir pour un programme donné. Il démontre notamment que ce gain est limité par l'ensemble des parties non parallèles de notre programme, quelle que soit la quantité de processeurs mis a contribution. Avec S – le facteur par lequel l'on a accéléré notre programme – fonction de p (le pourcentage en temps d'exécution de la partie parallèle) et d's (le gain en performance de ce pourcentage, aussi appelé speedup).

$$S(s) = \frac{1}{1 - p + \frac{p}{s}} \tag{1.1}$$

Cette loi est un premier pas pour définir la complexité d'obtenir un gain en performance à la hauteur de nos espérances lorsqu'on implémente la parallélisation des calculs. Mais cette loi ne prend par exemple pas en compte l'impact des différentes bandes passantes[9] (mémoire locale, réseau, bus d'échange), la complexité des accès mémoires des architectures récentes (NUMA notamment) et les schémas d'accès mémoires utilisés de manière précise. Les travaux présentés dans [9] visent à mettre à jour ces aspects là pour faire évoluer cette loi.

Une grande partie des techniques informatiques présentées dans la suite de ce manuscrit ont été mises en place pour réduire autant que faire se peut les limites dictées par cette loi.

II.2 Méthodes de résolution numérique

Différentes méthodes ont été mises en œuvre pour la simulation en mécanique des fluides, qui nécessitent la résolution numérique d'équations aux dérivées partielles. On distingue les méthodes faisant appel à des structures sous formes de graphes appelés maillages (volumes finis, éléments finis, différences finies, méthodes de Lattice-Boltzmann) et les méthodes dites sans maillage, telle que la méthode SPH (smoothed-particle hydrodynamics [60]), ou les méthodes de type Monte-Carlo (une majorité des méthodes énumérées ici sont présentées dans [100]). Lors de l'utilisation d'un maillage, chaque entité (sommet, arête, face, ou cellule) possède un certain nombre d'entités en contact immédiat appelées "voisins", et cette connectivité peut être alors utilisée pour calculer certains opérateurs différentiels tels que les gradients, et divergences en certains points du maillage, ainsi que les intégrales d'une fonction sur des surfaces ou volumes. On met ensuite à profit ces opérateurs pour définir un ensemble d'équations simulant notre écoulement. C'est l'approche qui est utilisée pour la résolution numérique des équations de Navier-Stokes dans *Code_Saturne*. Les caractéristiques géométriques et topologiques du maillage peuvent avoir un impact important sur la qualité et la précision des résultats numériques. La précision d'une solution est ainsi grandement liée à celle de son maillage et il faut ainsi savoir jongler entre précision voulue et coût de calcul associé (on impose ainsi en cas réel rarement le même niveau de précision à l'ensemble du maillage).

Les méthodes sans maillage ont été introduites pour palier à diverses limitations liées aux maillages, mais ont leurs propres inconvénients, et nécessitent souvent plus de ressources pour obtenir un même niveau de précision que les méthodes avec maillages, ce qui fait qu'elles n'ont gagné en popularité que très récemment [105, 43]. Le spectre des besoins industriels d'EDF R&D et les caractéristiques des méthodes de type volumes finis justifient encore aujourd'hui le choix des méthodes à base de maillage dans *Code_Saturne*.

II.3 Résolution d'un pas de temps dans *Code_Saturne*

Il existe plusieurs variantes du schéma de résolution en temps dans *Code_Saturne*, le schéma de référence étant de type "SIMPLEC" [83]. Les autres schémas disponibles sont similaires du point de vue des opérateurs utilisés et des aspects "performance".

Ce choix de schéma implique de construire et résoudre plusieurs systèmes linéaires différents à chaque pas de temps. Les systèmes sont faiblement couplés entre eux grâce à l'utilisation de termes correcteurs issus des précédents systèmes du même pas de temps.

Le schéma est aussi co-localisé, au sens où les diverses variables s'appuient sur le même support de discrétisation. Les divers systèmes linéaires résolus s'appuient donc sur des matrices aux coefficients différents mais à structure identique.

Les avantages de ce type de schéma dit "segregated"sont :

— une consommation mémoire beaucoup plus faible qu'un schéma couplé (qui nécessite de faire intervenir toutes les équations dans un même système)

— des systèmes linéaires avec de bonnes propriétés adaptées à l'utilisation d'algorithmes de résolution itératifs

L'inconvénient de cette approche réside principalement dans la limitation de la taille des pas de temps utilisés, et de difficultés pour des cas qui ne représentent pas la majorité des écoulements visés par l'application tel que le cas des fluides à forte viscosité. Les avantages sont ici largement supérieurs quant aux besoins d'EDF R&D et de *Code_Saturne* (la consommation mémoire étant un aspect primordial).

La reconstruction des gradients s'insère dans ce schéma du fait de la nécessité pour la correction de termes liés aux non-orthogonalités du maillage de résoudre chaque système de manière incrémentale. Pour chaque système, on doit assembler un système en s'appuyant sur des équations de bilan, puis le résoudre. Les calculs de bilans font intervenir un opérateur de type "gradient", qui est "reconstruit" dans notre choix de discrétisation spatiale : la résolution se fait sur la conservation de

la matrice initiale auquel on ajoute le dernier second membre calculé, dont la construction fait intervenir des calculs de bilans et une construction de gradients. Ce processus est effectué de manière itérative, seul le second membre étant mis à jour.

On notera aussi que l'approche incrémentale utilisée est assez contraignante d'un point de vue parallélisme, des réductions globales impliquant une synchronisation de certaines valeurs étant utilisées entre chaque étape. La plupart de ces réductions concernent des tests de convergence des sous-étapes.

II.4 Discrétisation par maillages structurés

On représente souvent les maillages structurés comme étant cartésiens (c.f. figures de gauche en figure 1.1) mais de manière générale, un maillage structuré se définit par la possibilité d'exprimer de manière algébrique l'ensemble des voisins d'une cellule. Pour un maillage cartésien 3D et une cellule $C_{i,j,k}$, on peut alors lister ses voisines en $(i \pm 1, j, k)$, $(i, j \pm 1, k)$ et $(i, j, k \pm 1)$. Cette connaissance intrinsèque de la cartographie du maillage permet au développeur de se soustraire à certaines gymnastiques et on considère les maillages structurés comment étant l'approche la plus simple. L'approche a cependant quelques défauts, dont une difficulté à représenter des objets aux formes complexes, sans impacter fortement les performances (on peut raffiner un maillage structuré de manière à "coller" à des formes plus complexes, mais cela signifie une forte augmentation du nombre d'entité et donc du temps de calcul).

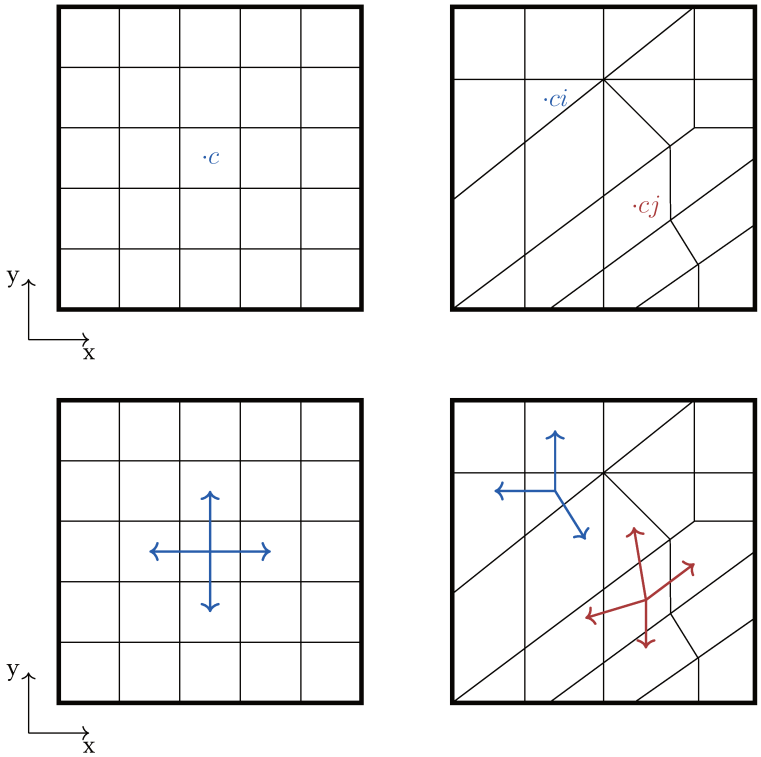


Figure 1.1: Exemples de maillages structurés (à gauche) et non structurés (à droite). On distingue la variabilité du nombre de voisin pour une cellule donnée en milieu non structuré.

II.5 Maillages non structurés

Pour représenter des formes plus complexes, on peut alors faire appel à des maillages dits non structurés (cf. figures de droite en figure 1.1). Ceux-ci se définissent essentiellement par l'incapacité

à déterminer de manière algébrique le voisinage d'une cellule. Ils peuvent être composés de types d'éléments, de tailles et de formes variables, nécessitant une indexation des éléments (le voisinage de chaque cellule est alors stocké, de même que les relations face→ cellules par exemple). Cette représentation peut impacter les performances, l'indirection nécessaire à l'accès du voisinage et la numérotation utilisée pouvant induire des sauts dans les accès mémoire voir des défauts de cache (c.f. figure 1.2).

II.6 Maillages hybrides et multiblocs

Au delà des types de maillages classiques, on peut dénombrer des solutions permettant de tirer profit des deux mondes. Les maillages structurés multiblocs[11], permettent notamment de considérer un maillage comme un ensemble de maillages structurés (avec des raffinements différents). Les zones de contacts inter maillages sont alors les seules parties nécessitant des indirections pour définir les connectivités. Les maillages AMR (pour Adaptative Meshing Refinement [16, 92]) permettent eux de maintenir une logique cartésienne du maillage tout en proposant différents niveaux de raffinement (par subdivision des cellules). La nature structurée du maillage est alors perdue mais la régularité du maillage permet de conserver une partie des hypothèses de calculs faites en structuré.

II.7 Choisir un type de maillage

Différents aspects justifient l'utilisation d'un type de maillage en particulier. On peut noter principalement la justesse de la discrétisation obtenue ainsi que des résultats de simulations, l'impact sur les performances, la facilité à mailler, l'impact sur le développement (complexité du code).

L'approche retenue dans *Code_Saturne* est l'utilisation de maillages non structurés, choix commun à de nombreux logiciels de simulation en mécanique des fluides. Ce choix peut être discuté sur différents points, mais se justifie largement par les besoins industriels d'EDF R&D (complexité et grande variété des maillages utilisés notamment).

Il est reconnu que de manière générale la complexité du maillage impacte directement les performances[50] (solveurs plus lents, structure du système d'équations moins optimale, complexité du code : accès aux connectivités par tableaux d'indirections plutôt que par des règles prédéfinies, entrainant un surcout qui sera fonction de la bande passante mémoire et de la taille du cache). L'utilisation d'un maillage non structuré permet cependant de représenter une géométrie complexe pour une quantité de mailles inférieure, ainsi que d'adapter la taille de la maille (et donc la précision de la simulation) en fonction de la localité des phénomènes physiques étudiés. On note ici que gain en memoire n'est donc pas synonyme de gain en performance. Le choix de tel ou tel type de maillage pourra donc se jouer sur le ratio entre perte de performances des méthodes impliquées et gain sur la taille du maillage (on peut s'attendre à une perte de performance d'un facteur entre 3 et 10 contre un gain en taille de maillage entre 10 et 100. Ce ratio justifie ce choix pour *Code_Saturne*). Malheureusement ce type de structure peut sembler aller à l'encontre des évolutions matérielles (GPGPU massivement parallèles, cartes accélératrice Many-core) car il est plus difficile d'appliquer les stratégies d'optimisations recommandées[88], limitant de fait notre capacité à réduire l'écart avec la performance crête des machines récentes. Des travaux pour appliquer ces techniques sur maillages non structurés tels que celles présentées dans [86] tendent à décourager les tentatives tant l'idée d'obtenir un code à la fois performant et portable semble lointaine.

D'un autre coté, les méthodes hybrides peuvent sembler intéressantes. L'aspect "meilleur des (deux) mondes" est cependant à remettre en perspective, les méthodes hybrides induisant naturellement une complexité des structures plus importante, complexifiant par ce biais la partie numérique (l'aspect hybride impose de facto au développeur de prendre en compte à la fois la nature structurée et non structurée du maillage si il souhaite tirer profit de ces méthodes). L'utilisation de méthodes hybrides est qui plus est réduite à l'utilisation de maillages spécialement construits pour maintenir

les informations des relations entre blocs, et les outils le permettant sont limités et difficiles à utiliser (le format CGNS, initialement co-développé par la NASA et Boeing dans un but de simplifier les échanges de données entre scientifiques le permet notamment, mais ce n'est pas le cas de la majorité des mailleurs) rendant ces méthodes peu adaptées au contexte industriel. Les auteurs de [50] reconnaissent d'ailleurs leur succès comme assez faible.

Les méthodes AMR sur base de maillages structurés souffrent quant à elles de l'aspect régulier en dimension des cellules de base, obligeant un sur raffinement du maillage lors qu'en situation de proche parois et d'alignement avec les axes Cartésiens [1].

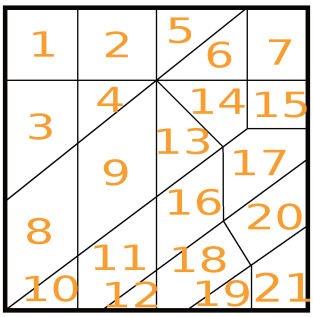


FIGURE 1.2: Numérotation et indirection possible associée à un maillage non structuré.

II.8 Localité de la mémoire

On distingue deux modèles principaux (à haut niveau) de mémoire définis selon le rapport entre unité de calcul et mémoire. D'un côté la mémoire distribuée implique la notion que l'ensemble de notre mémoire utilisable est répartie sur un ensemble d'unité séparées (un assemblage de machines, chaque nœud ayant sa propre mémoire). C'est un schéma que l'on retrouve très fréquemment en calcul scientifique, et qui a façonné les bibliothèques pour ce dernier, avec notamment des bibliothèques pour faciliter l'échange de messages dont la plus connue est MPI (Message Passing Interface[58], une bibliothèque qui cible le paradigme de programmation par échange de message). En effet, afin d'exploiter une multitude d'unités de calculs distantes les unes des autres et ayant chacune leur propre mémoire (mémoire locale, ou distribuée), il est alors nécessaire de répartir la charge de travail (partitionnement) et de faire communiquer (si une dépendance sur le travail du voisin existe) ces unités, l'une ne pouvant accéder aux données de l'autre de manière transparente. L'arrivée des processeurs à plusieurs cœurs a mis en avant la localité de la mémoire, plusieurs unités de calculs (cœurs) pouvant alors accéder simultanément à la même mémoire. On parle alors de mémoire partagée. À l'inverse de la mémoire distribuée, l'accès à l'ensemble des données est en théorie transparent pour le développeur. Il doit néanmoins prendre soin de ne pas avoir deux unités interférant sur la même section de mémoire (problème de concurrence), et dans les faits, la rapidité d'accès à différentes parties de la mémoire peut dépendre de son emplacement physique. On parle alors d'effet NUMA[64, 71], en référence à l'architecture (pour Non-Uniform Memory Access).

II.9 Partitionnement

Le découpage des données est une des premières étapes au calcul parallèle. Elle se fait l'écho de l'arrivée massive du parallélisme comme moyen d'accélération où elle est primordiale.

1. Les méthodes AMR étant basées sur une représentation à base d'octree, le raffinement d'une maille est un processus identique quelque soit la situation et ce dans toutes les dimensions, là où en non structuré et pour la même situation on se contenterait d'étirer les mailles parallèlement à la paroi (limitant largement le nombre de mailles nécessaires).

Dans ce cadre, elle a deux objectifs principaux : d'une part faire en sorte que chaque unité de calcul se voit attribuer une charge de travail similaire, et d'autre part de minimiser les zones de frontière en nombre et en taille (la multiplicité des échanges pouvant être aussi (voir plus) coûteuse que leur taille en fonction des propriétés du support de communication [2]. Ces objectifs ont pour but final de minimiser le temps perdu par chaque unité de calcul en attente des autres (équilibre de charge) ainsi que le temps perdu à échanger des informations entre unités de calcul.

Rapporté au monde de la simulation numérique où il est courant de représenter le domaine de calcul par un maillage (discrétisation de l'espace), cette étape est appelée le partitionnement de maillage. On établit ainsi une relation entre le maillage de départ, trop conséquent pour être pris en charge par une seule unité de calcul dans des temps convenables, et un partitionnement de ce maillage de telle sorte que chaque unité de calcul ait sa propre partition. Afin de maintenir une bonne cohérence de nos données, chaque unité doit cependant échanger continuellement avec celles partageant une frontière de leur propre partition. On les caractérise alors communément comme étant voisines. Ce terme est généralisé à toute unité (géométrique, de calcul) qui possède une adjacence avec une autre.

Le partitionnement de maillage peut être une étape quasi implicite lorsque par exemple le maillage utilisé permet une répartition géométrique de ce dernier. C'est par exemple le cas d'un maillage cartésien régulier (ou dès lors que le maillage est structuré et composé d'éléments de tailles similaires), auquel cas l'on est capable de manière intuitive de proposer une répartition équitable de la charge sous forme de blocs. Partitionnement et stratégie de bloc sont alors étroitement liés (on peut aisément voir leur association dans les exemples de parallélisation de la factorisation Cholesky par paradigme de tâches qui sont proposés par les équipes de StarPU et PaRSEC [21, 1], où la granularité est le seul élément qui dissocie une partition d'un bloc).

En considérant des géométries plus complexes telles que des maillages structurés hiérarchiques (octree/AMR) ou non structurés, leurs partitionnements cessent d'être facilement déterminables. On distingue deux grandes familles d'algorithmes visant à répondre à cette problématique :

— algorithmes dits géométriques
— algorithmes à base de graphes

Les algorithmes géométriques opèrent un partitionnement spatial agrégeant ensembles les cellules géométriquement proches. On peut dénombrer plusieurs approches majeures telles que la Recursive Coordinate Bisection (RCB) et sa variante Multi-jagged [45], la méthode inverse de Cuthill-Mckee (RCM [73]) ainsi que les courbes fractales de remplissage d'espace Morton et Hilbert. Ces approches ont le mérite d'être assez simples et performantes mais ne proposent pas vraiment d'optimisation des communications (mis à part le RCM dont c'est l'usage premier).

Ce dernier aspect peut être plus facilement pris en compte lorsqu'on utilise un partitionnement à base de graphe. SCOTCH, METIS et leur variantes parallèles PT-SCOTCH [36] et ParMETIS [66] sont deux des nombreuses solutions proposant un partitionnement par graphe. Une des approches du partitionnement par graphe consiste à établir un graphe dual de notre maillage (chaque cellule devient un sommet du graphe et chaque sommet est relié par une arrête si leurs cellules d'origine sont voisines). Dans ce cas, minimiser les communications revient à minimiser le nombre d'arêtes sortantes que chaque sous graphe contiendra. Le partitionnement de graphe est reconnu comme faisant parti de la classe des problème NP-difficiles et de ce fait les bibliothèques telles que SCOTCH essayent d'approcher, sans garantie, une solution optimale qui peut varier d'une exécution à une autre. Elles utilisent pour se faire un ensemble d'heuristiques telles que celles de Kernighan-Lin[67] ou Fiduccia-Matheyses[52] qui sont des méthodes dites locales (qui sont appliquées petit à petit à

2. On oppose ici la latence à la bande passante. La latence est un coût qui affecte chaque échange, tandis que la bande passante affecte les performances des échanges de manière générale puisque c'est la vitesse de l'envoi des données qui est impactée. Si l'on souhaite effectuer de nombreux petits échanges sur un réseau avec une bonne bande passante mais une latence moyenne, on payera alors une multitude de fois le coût de latence. Ce coût aurait pu alors être évité en agglomérant ensemble ces envois en un seul.

une partie du graphe à la fois seulement) et qui fonctionnent sur des graphes pondérés. Par défaut dans *Code_Saturne* le graphe obtenu à partir de notre maillage n'est pas pondéré, et les librairies telles que SCOTCH ou METIS ne peuvent donc pas tirer pleinement partie de leur optimisations locales. On pourrait pondérer les arêtes du graphe en fonction du type de cellules qu'elles relient (tétraèdre, hexaèdre) mais cette pondération ne s'avèrent correcte que pour une partie des opérations effectuées durant la simulation. En effet, si certains opérateurs ont un coût constant en fonction du type d'élément, d'autres ne sont appliqués que pour certaines parties du maillage. La pondération du graphe dans un tel environnement revient elle-même à l'application empirique d'heuristiques. [95] rapporte les performances de la plupart des techniques de partitionnement présentées ci dessus. La figure 1.3 présente l'étape de partitionnement d'un maillage 2D et montre la relation à l'échelle des sous domaines, entre partitionnement et numérotation (lors d'un partitionnement par SFC tel que Morton ou Hilbert). En pratique, mis à part le choix de la méthode de partitionnement, il nous est difficile d'établir une stratégie viable pour améliorer l'équilibrage de charge, la définition de l'équilibre de charge présentée en [95] ne représentant qu'une vision assez simple du problème (problème NP-complet).

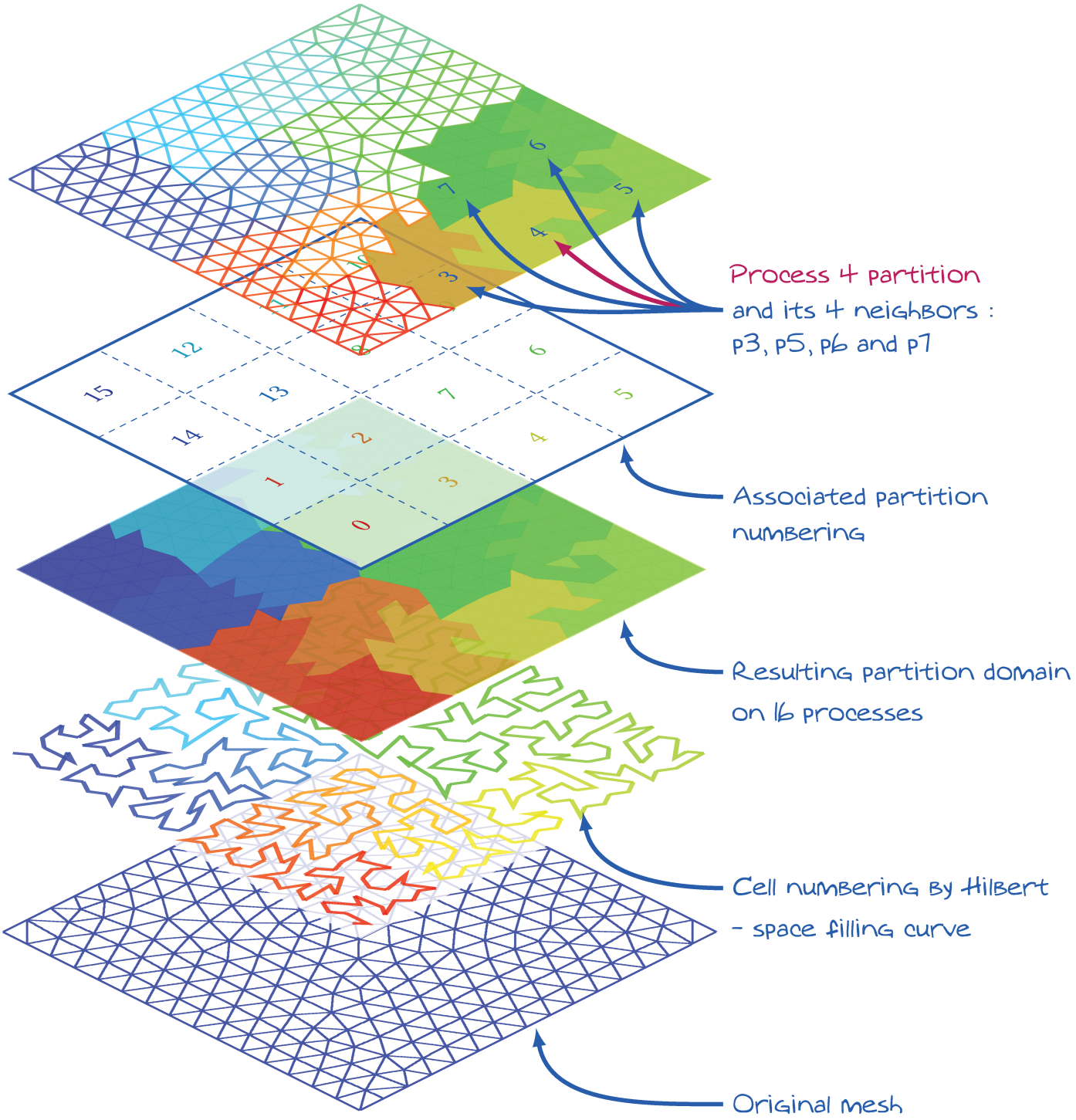


FIGURE 1.3: Présentation et relation entre le partitionnement et la numérotation

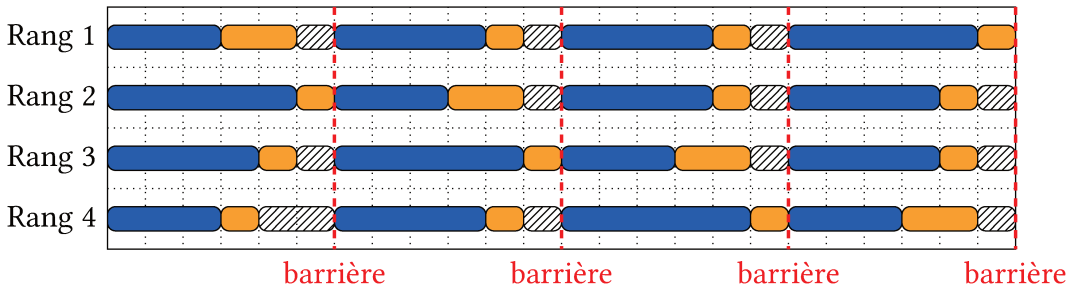


FIGURE 1.4: Une exécution classique via le modèle BSP (Bulk Synchronous Parallelism). En bleu les calculs, et orange les réceptions de données et en rayés les temps d'attente. L'ensemble des rangs sont synchronisés de manière périodique. Ces synchronisations peuvent notamment être effectuées pour garantir la validité des données après des échanges entre processus. Si il existe un déséquilibre de charge, tous les processus devront attendre le plus long avant de pouvoir poursuivre leur opérations.

Au delà de cette problématique majeure, un bon partitionnement doit aussi limiter le volume d'échange (bande passante) et le nombre de voisins à chaque partition (latences) car il peut être nécessaire d'effectuer une mise à jour fréquente des données aux frontières de chaque sous domaine, que l'on appellera ici échange des halos (aussi appelés cellules fantômes). La reconstruction des gradients par exemple est une opération qui met à jour le scalaire d'une cellule en fonction de ses voisines. Or une fois les partitions distribuées, les cellules aux frontières de chacune d'entre elles auront alors besoin d'accéder aux cellules correspondantes sur les partitions voisines. Cette problématique est d'ailleurs largement mise en avant par le modèle BSP qui tend à pénaliser toute répartition qui ne saurait être suffisamment équilibrée.

II.10 Importance de l'équilibrage de charge et de la localité

Le modèle BSP (Bulk Synchronous Parallelism) est largement utilisé par l'ensemble du parc logiciel faisant appel à la bibliothèque de communication MPI (Message Passing Interface). Il consiste en la prise en compte de manière explicite des communications inter processus dans l'expression d'algorithmes parallèles. La figure 1.4 montre le cas où un manque d'équilibrage de charge pourrait nuire aux performances. Pour un logiciel de simulation utilisant des maillages non structurés, on peut s'attendre à un déséquilibre plus important que sur maillages structurés. En effet, la différence du nombre de voisins pour un tétraèdre ou un hexaèdre (respectivement 12 et 24 voisins) implique des itérations à taille et à temps variable en fonction du type d'élément sur lequel on procède. Deux partitions ayant le même nombre d'éléments mais pas le même type ne prendront donc pas le même temps à être simulées.

La figure 1.5 quant à elle, montre l'effet de la localité d'une numérotation spatiale. Si ce type de numérotation opère de manière locale, elle créée forcément des sauts dans la numérotation pour deux cellules pourtant voisines (cf. les deux cellules vertes voisines a gauche du maillage). On peut s'attendre à des comportements similaires à des endroits différents en fonction de la numérotation choisie (aucune ne peut être parfaite, la figure 1.6 liste le niveau de proximité d'une cellule avec ses cellules voisines en fonction du type de numérotation choisie. On constate alors que plus de 50% des cellules voisines se situent dans un intervalle de 16 à 128 de la cellule qu'elles partagent.). Puisque l'on fait intervenir un certain nombre de grandeurs positionnées sur les cellules (on possède ainsi une multitude de tableaux ordonnés en fonction de cette numérotation), la qualité de la numérotation impactera les performances de l'ensemble des accès a ces tableaux.

II.11 Reconstruction des gradients

On liste deux méthodes majeures de reconstruction de gradients dans *Code_Saturne* :

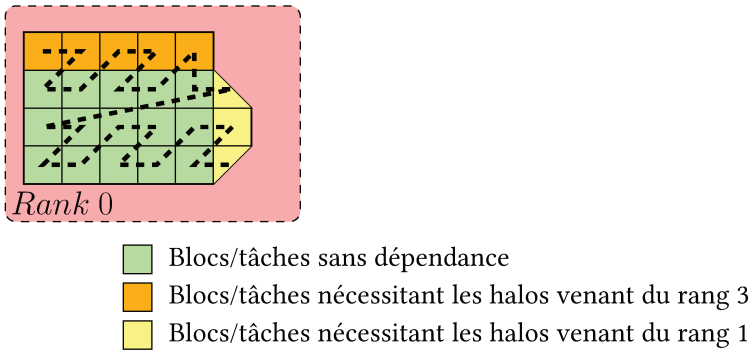


FIGURE 1.5: Représentation d'une numérotation SFC (les Space Filling Curves, courbes de remplissage, sont des méthodes permettant de s'assurer du parcours de la totalité du graphe et sont fréquemment utilisées pour la numérotation du fait de leurs simplicités et de leurs résultats) sur une partie de maillage. On met en avant ici les sauts en numérotation périodique qui peuvent arriver (ici sur les deux cellules voisines vertes a gauche, la première étant numérotée en début de la courbe fractale et sa voisine étant numérotée en milieu de parcours).

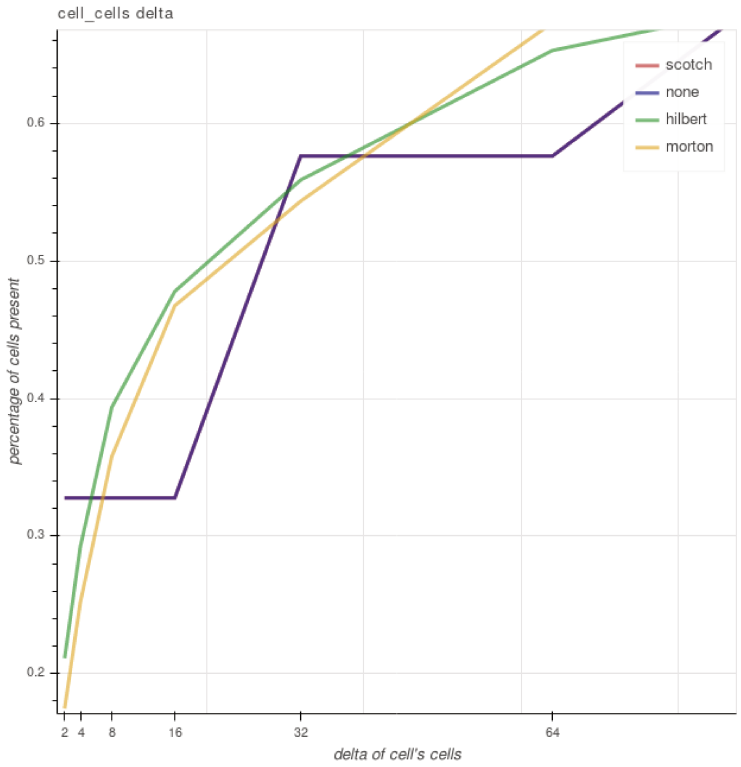


FIGURE 1.6: Proximité des cellules voisines en fonction de la numérotation choisie.

— méthode itérative
— méthode de moindres carrés (méthode LSQ)

Ils font partie intégrante et largement non négligeable de la résolution d'un pas de temps (allant de 20 a 60% des temps de calculs). En y ajoutant la résolution des systèmes linéaires, on peut monter cette estimation a 80% du temps de calcul (90% par le passé).

Si la résolution des systèmes linéaires peut être déléguée à des bibliothèques externes spécialisées (sur lesquelles des efforts importants sont réalisés depuis de nombreuses années par des équipes diverses, avec une force de développement supérieure à celle de *Code_Saturne*) telle que PETSc [59], la reconstruction des gradients ne peut bénéficier de cette approche. La reconstruction présente aussi des schémas similaires à d'autres opérateurs de *Code_Saturne*, rendant l'étude de cette dernière logique et pragmatique.

Résolution d'$Y = AX$ en algèbre linéaire creuse La résolution des équations aux dérivées partielles telles que celles de Navier-Stokes sur la base d'un maillage sous-tend l'utilisation de systèmes linéaires représentés par des matrices creuses (car ne faisant intervenir que les voisins immédiats), qui peuvent nécessiter la simple application locale de produits de ces matrices par des vecteurs (pour des schémas explicites en temps) ou la résolution de systèmes linéaires (ou non) représentés par des matrices creuses (pour des schémas temporels au moins en partie implicites en général plus stables et permettant l'utilisation de plus grands pas de discrétisation en temps).

Le traitement informatique de ces dernières a un certain impact sur la programmation et les performances. Les matrices sont compressées et indexées afin de minimiser la quantité de mémoire nécessaire à leur stockage. Le parcours des termes de systèmes de grande taille entrainant des potentiels défauts de caches, il est laissé au développeur le soin de trouver les algorithmes privilégiant au maximum les accès locaux. L'arrivée d'unités de calcul massivement parallèles permit la prolifération des travaux d'optimisations de cette résolution de système en matrice pleine bien que dans notre cas il reste plus efficace de prendre en compte l'aspect creux des matrices.

Du fait de la grande taille des systèmes à résoudre [3], *Code_Saturne* utilise des méthodes itératives (et non directes, sur lesquelles les décompositions auraient pu être appliquées). La résolution du système est alors accélérée au moyen d'un pré conditionneur (en pratique Jacobi, multi-grille ou pré conditionneur polynômial).

3. Ce choix est actuellement motivé aussi pour la capacité de ces méthodes à converger de manière efficace lorsque le choix du pré conditionneur est probant. L'aspect mémoire est aussi un facteur de choix puisque un solveur direct nécessite de tenir en mémoire l'ensemble du système, contrairement à un solveur itératif.

2 | Contributions algorithmiques

I Contribution sur la reconstruction de gradients

Telle que détaillée précédemment, la reconstruction de gradients est une étape non seulement coûteuse en temps dans *Code_Saturne* mais aussi à la fois représentative d'autres opérateurs utilisés dans le code et très spécifique à *Code_Saturne* (les solveurs sont eux la cible d'un effort de nombreuses équipes de développement spécialisées en HPC).

I.1 Minimisation par moindres carrés

La reconstruction de gradients via la minimisation par moindres carrés utilise l'estimation d'une composante du gradient aux faces grâce aux valeurs de la fonction au centre des cellules voisines. Cette méthode est plus rapide mais moins robuste que la méthode itérative.

La figure 2.1 représente la reconstruction du gradient d'une cellule comme l'agrégation de donnée située sur les faces de nos cellules voisines. Elle met aussi en avant la problématique d'attente sur la réception des halos, nécessaire à la reconstruction de gradients aux frontières de chaque sous-domaine. En pratique, cette attente est généralisée à la totalité de la reconstruction de telle sorte qu'aucun recouvrement de communication par les calculs n'est effectué. Diverses méthodes pourraient néanmoins permettre celui-ci, tels qu'une numérotation prenant en compte la localité des cellules en regard des sous domaines voisins pour positionner ces dernières en aval des autres ou encore la programmation par tâches (dont on espère la représentation moins impactante tant en terme de complexité du code que des performances attendues).

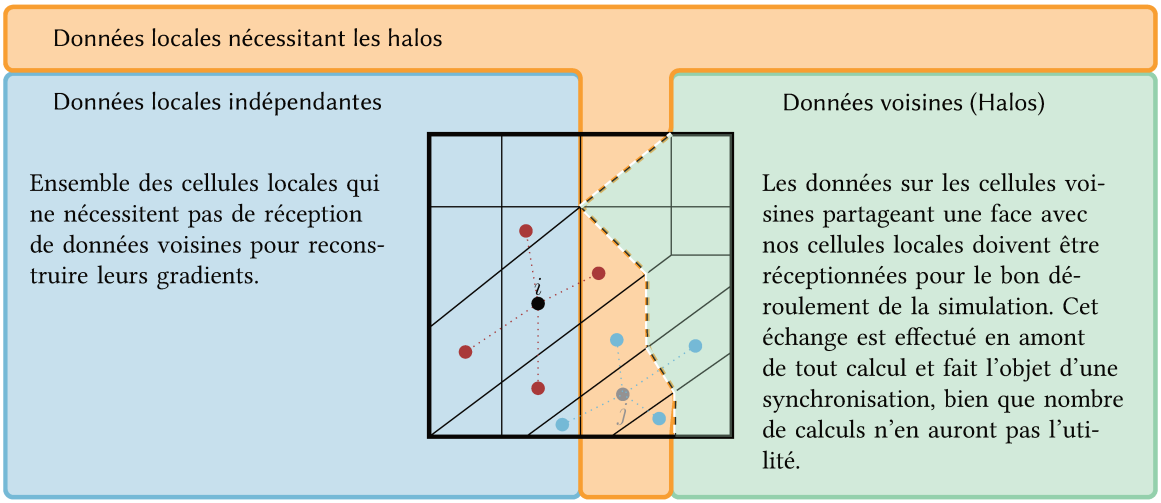


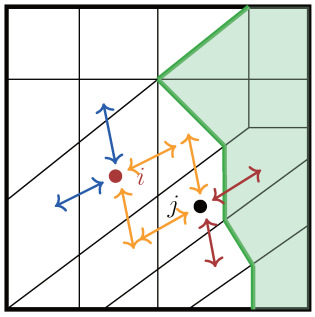
FIGURE 2.1: La reconstruction des gradients par méthode des moindres carrés se fait via l'estimation des composantes du gradient aux faces à partir des valeurs de la fonction au centre des cellules voisines.

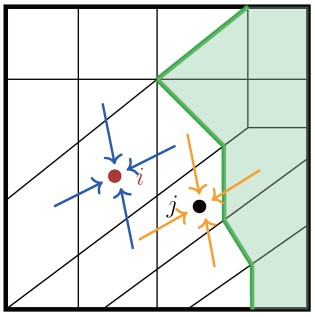
I.2 Reconstruction face par face

Pour chaque face donnée, l'estimation du gradient que l'on en retient est, au signe près la même pour les deux cellules qui la partagent. On bénéficie alors de cette symétrie par un parcours du maillage sur les faces, de telle sorte que le gradient d'une cellule ne soit entièrement reconstruit qu'une fois que l'ensemble des faces qui la composent ne soit parcourues. On minimise ainsi le nombre de calcul effectués (voir la sous section I.6 sur la modélisation des performances). La figure 2.2 reprend cette procédure et met en avant la problématique qu'elle impose sur un parallélisme en mémoire partagée. En découpant les opérations de calculs par face, on impose plusieurs écritures

pour chacune des valeurs stockées par cellule. Il est alors nécessaire de s'assurer que ces écritures ne sont pas exécutées de manière concurrentes sous peine d'invalider le résultat final.

Reconstruction par face : on propage la valeur en partant de chaque faces à ses deux cellules. Il y a risque d'écriture concurrente si des précautions ne sont pas prises. Ici les faces des cellules i et j ne peuvent pas être utilisées en même temps sans risque d'écrasement de donnée





Reconstruction par cellule : on agrège pour chaque cellule la valeur aux faces de ses voisines. La contribution se fait alors pour une seule cellule. On perd donc l'économie obtenue lorsque l'on exploite la symétrie de la contribution d'une face à ses deux cellules. On supprime cependant notre problème de concurrence.

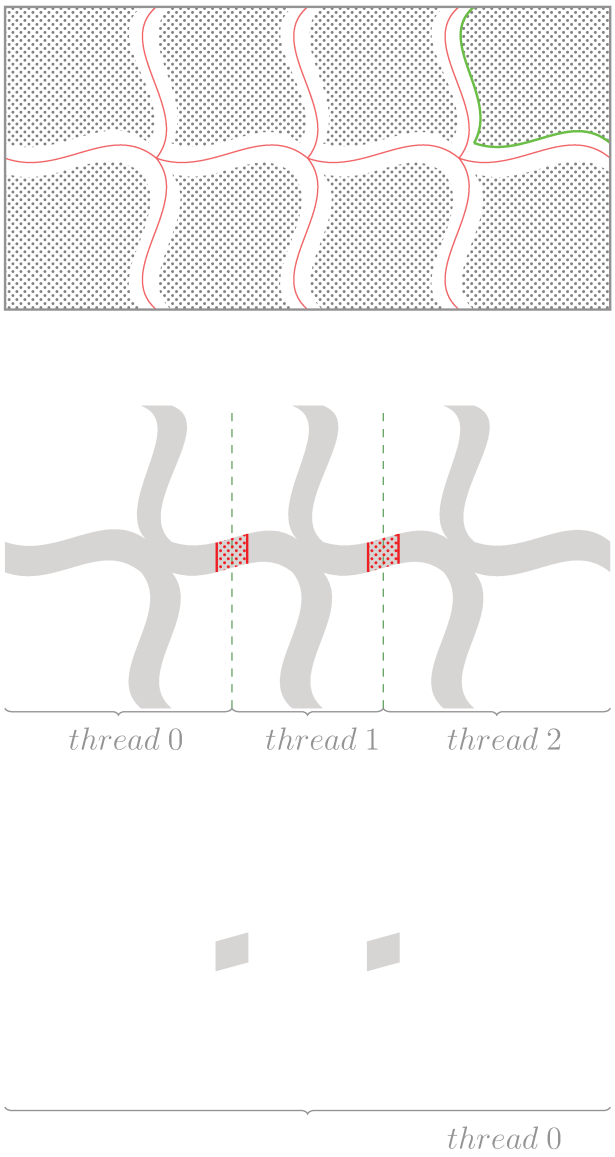
FIGURE 2.2: Contributions possibles dans le cas d'un reconstruction des gradients par face (à gauche) et par cellule (à droite).

I.3 Contribution : Reconstruction par itération sur les cellules

Étant donné le profil peu intensif en calcul de l'algorithme de reconstruction des gradients par face (voir la sous-section I.6 en page 26) et les problématiques de parallélisme en mémoire partagée amenée par cette reconstruction, une possibilité non utilisée dans *Code_Saturne* actuellement serait d'effectuer la reconstruction avec pour référentiel d'itération la cellule. On mettrait alors à jour pour chaque cellule ses données en fonction de ses voisines (en itérant sur chacune d'elles). Le schéma 2.2 présente la différence entre les deux modèles de reconstruction. En opérant de cette manière, on supprime une problématique majeure en parallélisme en mémoire partagée. On augmente par la même largement la quantité de calculs nécessaire, ce qui peut être un problème ou un avantage en fonction du support matériel utilisé. On présente et oppose ces deux méthodes dans la suite du manuscrit, tant sur leur impact sur la facilité à les paralléliser que sur leur performances potentielles.

I.4 Parallélisme multi-niveau

La parallélisation de la reconstruction, lorsqu'elle est effectuée face par face, demande la prise de certaines précautions. En effet, en mettant à profit la symétrie de la contribution d'une face à ses deux cellules, on impose de facto une contrainte en écriture sur ces deux dernières. Lors d'une parallélisation via des processus légers, on est alors contraint de sécuriser l'accès en écriture de chaque gradient. La figure 2.2 reprend l'expression schématique de ce problème. La solution retenue à l'époque a été la constitution de groupes de faces pouvant être utilisés en parallèle via l'algorithme multi-passe dont le fonctionnement est détaillé en figure 2.3. Dans un premier temps les faces sont groupées par blocs n'étant pas en contact les uns des autres. Pour se faire, une marge de séparation est prise, qui sera traitée dans une étape suivante. Une fois cette étape effectuée, on est en possession d'un groupe de blocs, chaque bloc étant exécutable en parallèle des autres. L'exécution au sein d'un même groupe est quant à elle obligatoirement séquentielle. On procède enfin à la même étape sur les faces restantes qui n'auront pu être sélectionnées auparavant jusqu'à ce qu'à chaque face corresponde un bloc dans un groupe. La figure expose naturellement le risque de manque d'équilibrage de charge que cette solution présente, où les dernières itérations sont susceptibles de posséder un nombre de blocs exécutables en parallèle inférieur au nombre de processus légers mis à notre disposition.

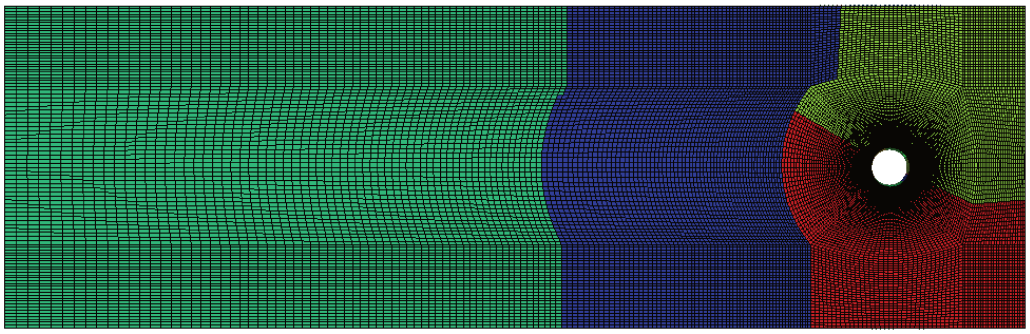


Etape 1 : le maillage est découpé en groupes de taille similaire et n'ayant pas de faces en commun. Chaque groupes peut ainsi être attribué à un processus léger différent sans risque d'écrasement de mémoire.

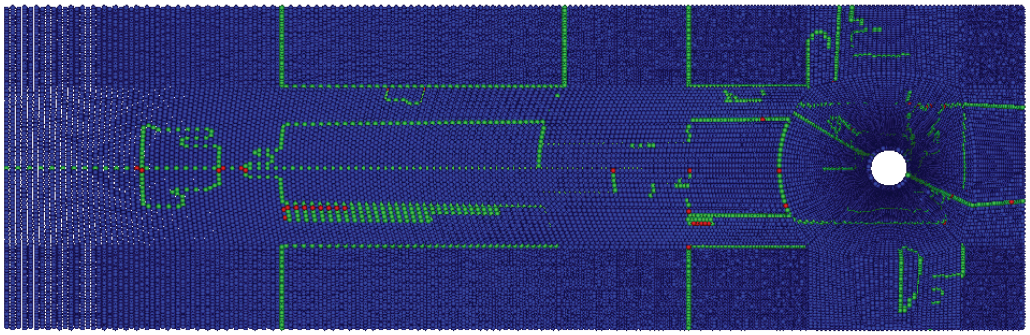
Etape 2 : Il ne reste plus que quelques cellules à calculer que l'on essaie de répartir de manière équitable. Des zones de contacts restent encore, et l'on utilise la même méthode qu'en étape 1 jusqu'à ce que le nombre de cellules atteigne un certain seuil en dessous duquel toute répartition de la charge de calcul serait dérisoire.

Etape 3 : Il ne reste que trop peu de cellules pour qu'il soit intéressant de répartir sur plusieurs processus. L'ensemble des cellules/faces restantes est donc attribué à un seul processus.

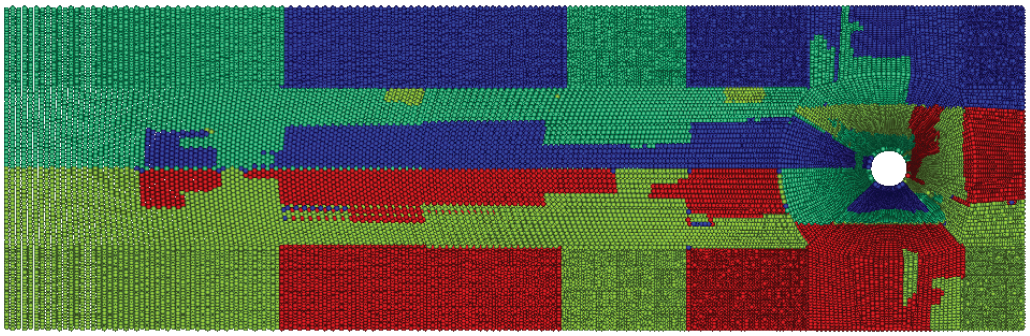
FIGURE 2.3: Détail de la procédure de séparation des faces en groupe de façon à ce que toutes les faces d'un groupe aient leur scalaire calculable en parallèle.



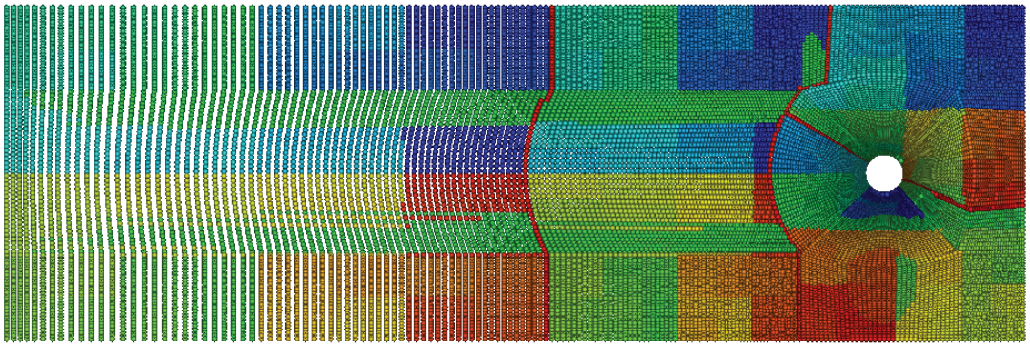
Domaine parallèle : Chaque couleur représente la partie du maillage allouée à un processus différent.



Répartition des faces avec l'algorithme multi passe : Chaque couleur représente un ensemble de faces dont les valeurs associées pourront être modifiées en parallèle sans risque d'interférence.



Association des faces avec leur processus léger et en fonction du domaine parallèle et de leur groupe déterminé ci dessus.



Numérotation des cellules associée, par courbe fractale Hilbert.

FIGURE 2.4: De haut en bas, les différents niveaux de répartition et numérotation des faces et cellules. On retrouve visuellement la correspondance de la répartition des faces en algorithme multi passes avec la figure 2.3. On constate donc un grand déséquilibre entre les différentes passes (deuxième figure en partant du haut) qui peut jouer sur la qualité de l'équilibrage de charge des algorithmes utilisant cette solution (notamment la reconstruction des gradients par face).

I.5 Impact de la numérotation

La numérotation est une étape obligatoire pour tout calcul scientifique opérant sur des données non structurées. Elle permet de spécifier dans quel ordre on aura stocké nos données, et sera la référence sur laquelle on construira les correspondances de connectivité entre éléments. Puisque les maillages sont définis par plusieurs ensembles liés, on est amené à construire différents tableaux de correspondance : une liste de cellules, une liste de faces par cellules, ou l'inverse et cetera : le nombre et la nature des numérotations utilisées variant en fonction de ce qui peut simplifier l'expression de nos algorithmes. Puisque la numérotation va spécifier l'ordre d'accès à la mémoire, il est important que cette numérotation soit faite de telle sorte à privilégier une certaine localité car cela aura un impact fort sur les performances. À l'instar du partitionnement, des stratégies similaires sont utilisées (le Reverse Cuthill-McKee est notamment utilisé pour la numérotation des systèmes en algèbre linéaire creuse afin de limiter les besoins en bande passante). SCOTCH fournit notamment plusieurs stratégies de numérotation auxquelles nous ajoutons les algorithmes de numérotation sur base de SFC (Morton et Hilbert). La figure 1.3 présente la numérotation et son rapport avec le partitionnement.

On observe en pratique (voir figure 2.4) une augmentation de la disparité de la numérotation au fur et à mesure que l'on raffine le niveau de possession d'une entité : au départ le maillage est réparti de manière assez simple (bien qu'on puisse aisément tomber sur des cas où dès l'étape de partitionnement, la répartition des données ne semble pas privilégier des interfaces réduites entre voisins). Lorsque l'on applique par la suite l'algorithme multi-passe puis une numérotation de cellules qui en découle (dernière image de la figure 2.4), on obtient alors un résultat parfois assez chaotique. La figure 2.5 essaye de clarifier le phénomène en découpant chaque partie de la numérotation des cellules sur un rang, en fonction de son processus associé. On peut y observer la localité mais aussi la complexité de satisfaire cette heuristique sur un partitionnement réel aux formes complexes.

Dans le cas de la reconstruction des gradients par face, la numérotation a un impact particulier sur la gestion des données du fait de l'aspect itératif du calcul qui nécessite de maintenir l'ensemble des données temporaires de calculs en mémoire avant de pouvoir opérer la mise à jour des gradients. La figure 2.6 met entre autre en avant cette problématique.

A contrario, la reconstruction cellule par cellule permet l'essai de différentes techniques visant à l'amélioration des performances telles que le découpage des calculs par blocs (pour améliorer la localité des données) ou faciliter l'expression de l'aspect vectoriel d'un calcul [1]. Le calcul des termes temporaires est effectué pour chaque cellule en une seule itération, et la mise à jour de chaque gradient est disponible directement. Le stockage des membres nécessaires à la reconstruction n'est dès lors plus nécessaire. Cela peut représenter un gain significatif à la fois en utilisation mémoire mais aussi potentiellement un impact sur les défauts de cache. La figure 2.6 présente l'impact de la numérotation sur la reconstruction basé sur les cellules.

I.6 Modèle de performance

La création d'un modèle de performance permet d'apprécier et de prévoir le comportement de nos algorithmes. Ce modèle sert alors à estimer la distance séparant les performances actuelles de notre implémentation avec celles théoriques et optimales que l'on pourrait obtenir. On peut en retenir aussi les bénéfices de l'utilisation de telle ou telle implémentation en fonction de divers paramètres. Le modèle peut ainsi permettre d'optimiser une taille de bloc[79], ou servir de réglage automatique, en privilégiant un algorithme sur un autre en relation avec la machine (privilégier un

1. Si l'utilisation des capacités de calculs vectoriels des processeurs permet d'obtenir des résultats intéressants, la portabilité du code peut être entachée si celle-ci est effectuée via les intrinsèques[39, 98] d'Intel par exemple. Une pratique a mi-chemin via l'exposition au compilateur de la nature vectorielle d'une boucle de calcul (auto vectorisation [81]) permettrait une solution portable mais à l'efficacité non garantie.

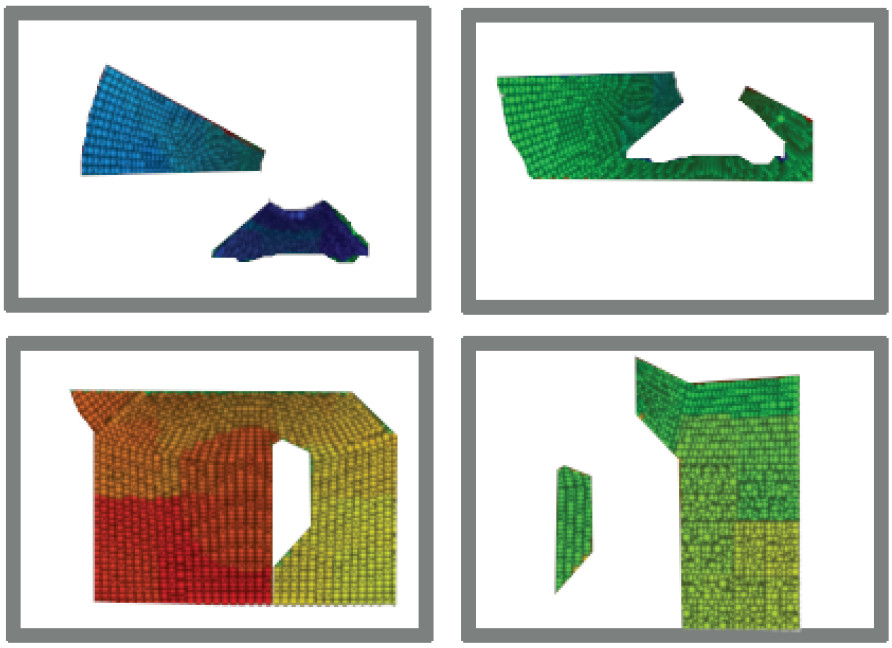


FIGURE 2.5: Découpage grossier des cellules d'un rang en fonction de son processus léger associé. Chaque bloc contient les données attribuées à un processus léger. On peut observer la relation de localité que créée la numérotation par courbe fractale. La couleur varie en fonction de l'évolution de la numérotation (en partant du bleu foncé jusqu'au rouge).

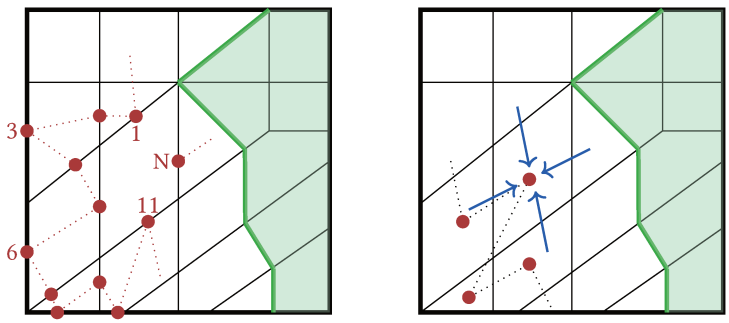


FIGURE 2.6: Comparaison d'une reconstruction sur les faces (à gauche) avec parcours sur les cellules (à droite) en fonction de la numérotation (ici on montre un schéma de numérotation de Morton). On en déduit que le calcul des termes sur chaque cellule ne peuvent être obtenu qu'après la fin des itérations y contribuant lorsque l'on reconstruit sur les faces. Au contraire, via une itération sur les cellules, chaque terme est rendu disponible dès la fin de l'itération sur cette cellule, de telle sorte que l'on rend disponible le terme à la reconstruction de gradient "au plus tôt".

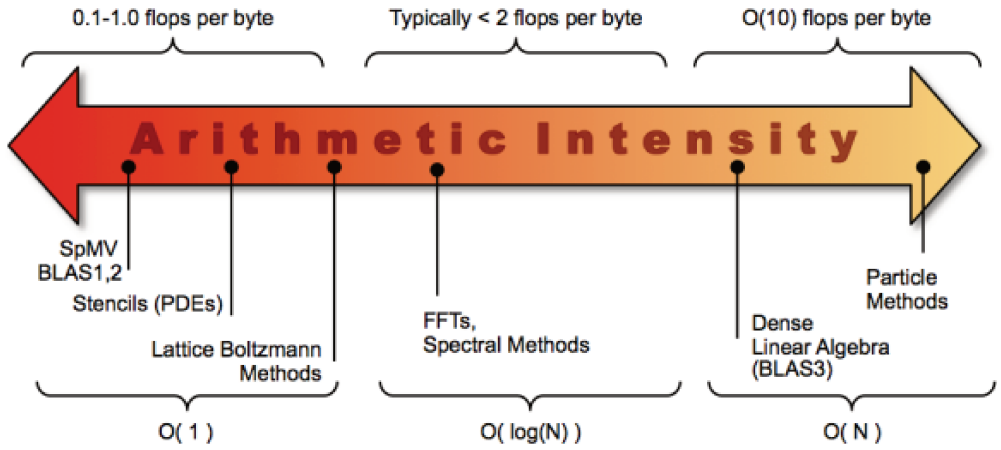


FIGURE 2.7: Classement des catégories d'algorithmes de calculs les plus connus en fonction de leur intensité arithmétique (Reproduction avec l'autorisation gracieuse de Samuel Williams).

algorithme à plus forte intensité arithmétique si l'on est face à une architecture de type Many-core par exemple).

I.6.1 Intensité arithmétique

Les évolutions récentes des architectures qui ont amené les applications scientifiques à atteindre le mur de la mémoire ont imposé un virage dans la manière de développer, forçant le développeur de telles applications à minimiser les entrées/sorties. En mettant en rapport le nombre d'opérations flottantes que notre algorithme effectue avec le nombre de lectures et d'écritures, on établit une vision nette de l'efficacité de notre algorithme. Plus ce rapport est grand, et plus notre algorithme est capable de rentabiliser son coût en mémoire. On dit alors que cet algorithme a une intensité arithmétique forte : c'est le cas de l'algèbre linéaire dense, telle qu'on la retrouve dans les BLAS3 par exemple. La figure 2.7 positionne les familles d'algorithmes les plus connus en fonction de leur intensité.

Intensité arithmétique :

$$\sigma = \frac{nombre\ de\ FLOPs}{8 \times nombre\ d'\ IOs} \tag{2.1}$$

On peut alors profiter de la connaissance de notre machine pour effectuer quelques prédictions de performances une fois l'intensité arithmétique de notre algorithme déterminée. En particulier, avec la bande passante théorique et les performances pics de la machine il est possible déterminer les performances atteignables comme étant le minimum entre ce pic et l'intensité arithmétique multipliée par la bande passante.

I.6.2 Roofline model

Le modèle de performance roofline met en relation les opérations flottantes mesurés d'un code avec son intensité arithmétique. Couplées aux performances calculatoires et mémoires pics de la machine, on dresse alors un tableau représentatif des performances assujetties aux contraintes de la machine. Il permet entre autre de juger quels sont les critères bloquants de notre code. La figure 2.8 reprend une description théorique spécifiant pour chaque niveau d'intensité arithmétique qu'elle en serait la cause ainsi qu'une des solutions possible. On remarque rapidement l'importance de proposer des algorithmes prenant en compte les effets du cache sur les performances. La figure 2.9 quant à elle montre l'importance dans un second temps de maximiser l'utilisation des accélérations

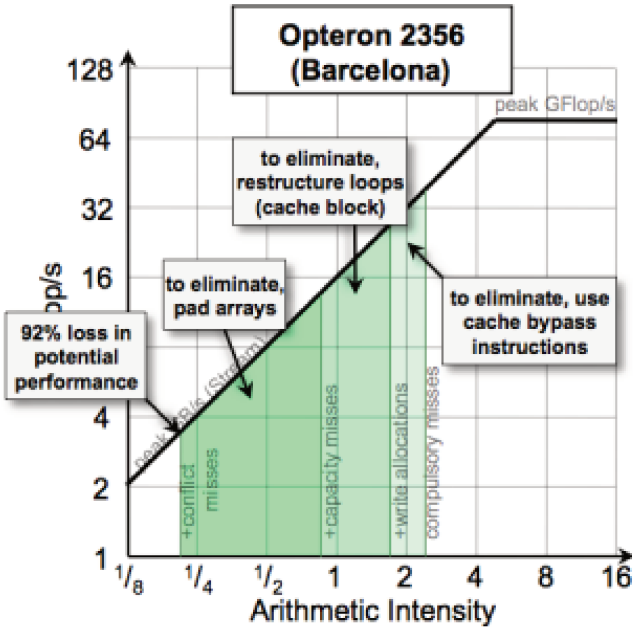


FIGURE 2.8: Exemple de roofline, à chaque intervalle d'intensité arithmétique correspond un certain type d'amélioration qui peuvent permettre un gain en intensité et donc en performance (Reproduction avec l'autorisation gracieuse de Samuel Williams).

matérielles offertes par le processeur. Ce type de graphe permet aussi aux constructeurs d'évaluer les performances des machines qu'ils proposent.

Les auteurs de [8] confirment l'efficacité du roofline model pour prédire les performances de tels algorithmes ainsi que leur prédisposition à être limités par la bande passante de la mémoire dynamique.

I.6.3 Construction d'un modèle de performance primitif

Afin de pouvoir définir un modèle de performance, il est nécessaire de déterminer un certains nombre de caractéristiques propres à notre algorithme. On s'attache ici à préparer le terrain en comptant le nombre d'opérations flottantes, mais aussi les I/Os de manière assez précise (Table 2.1). On pondère ainsi le comptage en prenant en compte les temps théoriques respectifs de chacune des opérations (écriture/lecture, addition/multiplication) [2].

- pour le calcul des gradients par face :
  $(75f_b + 22f_i + 20i) \bullet \beta + (78f_b + 19f_i + 15i)$
- pour le calcul des gradients par cellule :
  $(7i + 9Neigh\,(i) + 16f_b) \bullet \beta + (36i + 49f_b + 35Neigh\,(i))$

En se basant sur la table des opérations effectuées (table 2.1 en page 30), la formule 2.1 et les caractéristiques de notre machine, on peut en déterminer l'intensité arithmétique théorique de nos deux versions de reconstruction des gradients (c'est-à-dire le rapport des opérations flottantes calculées sur le nombre de bytes utilisés). Une fois calculée, cette intensité arithmétique permettra d'établir un graphe des performances atteignables sur une machine en fonction de ses performances mémoire et processeur spécifiques.

2. De manière générale, les écritures prenant singulièrement plus de temps que les lectures et les multiplications/-divisions étant bien plus coûteuses que les autres opérations flottantes, on met en avant en premier lieu les produits et les écritures.

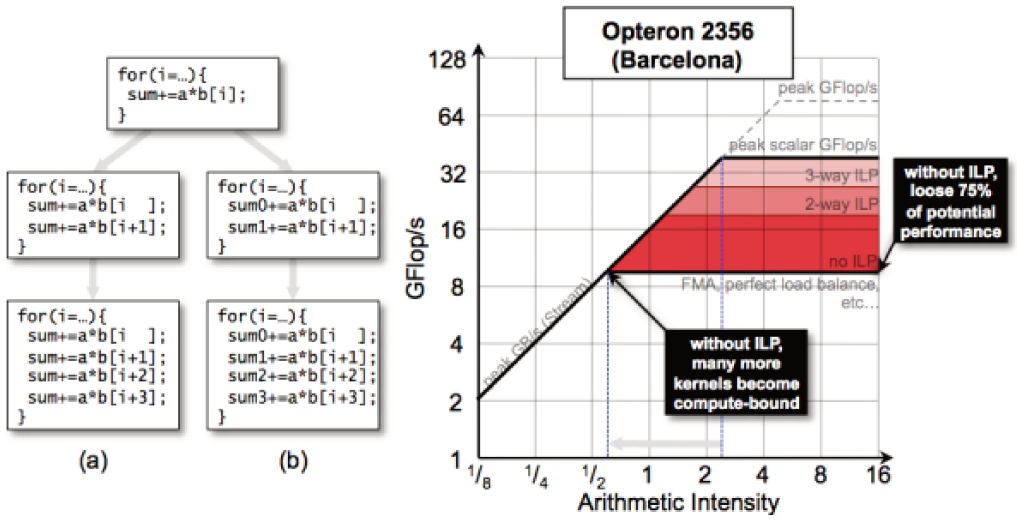


FIGURE 2.9: Autre exemple de Roofline mettant ici en avant l'importance de l'exploitation des capacités de calculs du processeur (vectorisation, et cetera) (Reproduction avec l'autorisation gracieuse de Samuel Williams).

boucles sur	métrique	calcul par itération sur : les faces	les cellules
faces de bords (f_b)	I/Os	75	
	FLOPs	78	
faces internes (f_i)	I/Os	22	
	FLOPs	19	
cellules (i)	I/Os	20	$16 \times \mathcal{F}_i^{ext} + 9 \times Neigh\,(i) + 7$
	FLOPs	15	$49 \times \mathcal{F}_i^{ext} + 35 \times Neigh\,(i) + 36$

TABLE 2.1: Comparaison des opérations sur les cellules de bord

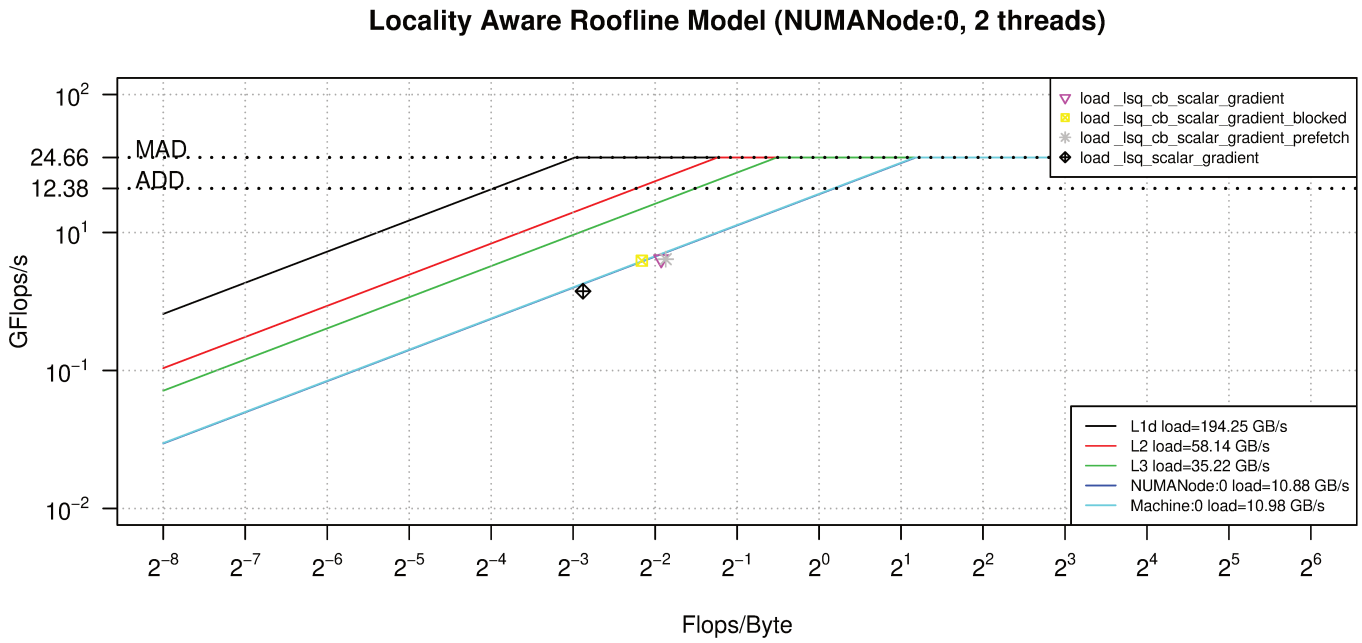


FIGURE 2.10: Performance de nos différentes versions de reconstruction de gradient (en chargement mémoire) remise en contexte vis à vis de l'intensité arithmétique de ces calculs. On constate que les performances ne varient pas en fonction de l'augmentation de l'intensité arithmétique et se cantonne aux performances de la mémoire RAM. Ces mesures indiquent largement qu'il faut se concentrer sur l'amélioration des accès mémoires pour espérer gagner en performance sur la reconstruction des gradients.

I.6.4 Roofline model sur Intel Core i7-5600U : mesures

Pour une machine composée d'un processeur Intel i7-5600U[3], on obtient les caractéristiques techniques suivantes :

— Pic théorique en bande passante (c.f. référence technique) : 25.6GB/s

— Pic théorique en GFlops : 92.8 GFlops/s (sur les 4 coeurs)

Des outils existent néanmoins pour mesurer la performance pic réelle d'une machine, permettant ainsi une évaluation plus juste des performances de notre application, notamment le Roofline Toolkit[101]. Les figures 2.10 et 2.11 présentent les mesures relevés sur ce processeur (via l'outil de mesure LARM (Locality Aware Roofline Model, maintenant CARM [44])). En pratique on observe un pic réel de 25 GFlops environ pour 4 cœurs. En exécution mono processus on mesure une puissance maximale de 12.7 GFlops/s pour une intensité arithmétique d'un peu plus de $\frac{1}{8}$ flop/byte.

Comparer les performances maximales de la machine avec les performances de nos algorithmes permet d'évaluer les intérêts des nos expériences en fonction des caractéristiques de la machine. Ces résultats peuvent ensuite être extrapolés pour prédire la meilleure version et les performances théorique d'une autre machine (Xeon Phi par exemple).

I.6.5 Discussion

On peut mettre à profit les courbes obtenues via l'utilisation du roofline model pour comprendre les limitations qu'imposent nos choix algorithmiques. En particulier, on observe que la performance

3. Il est bien évidemment préférable d'établir un tel modèle de performance sur une machine de calcul dédiée. Compte tenu des résultats présentés ici, l'établissement des résultats ne peut ici être vu comme préjudiciable : les performances de la reconstruction des gradients dans *Code_Saturne* est trop limitée par la mémoire du processeur pour avoir besoin de prendre en compte la vitesse du processeur.

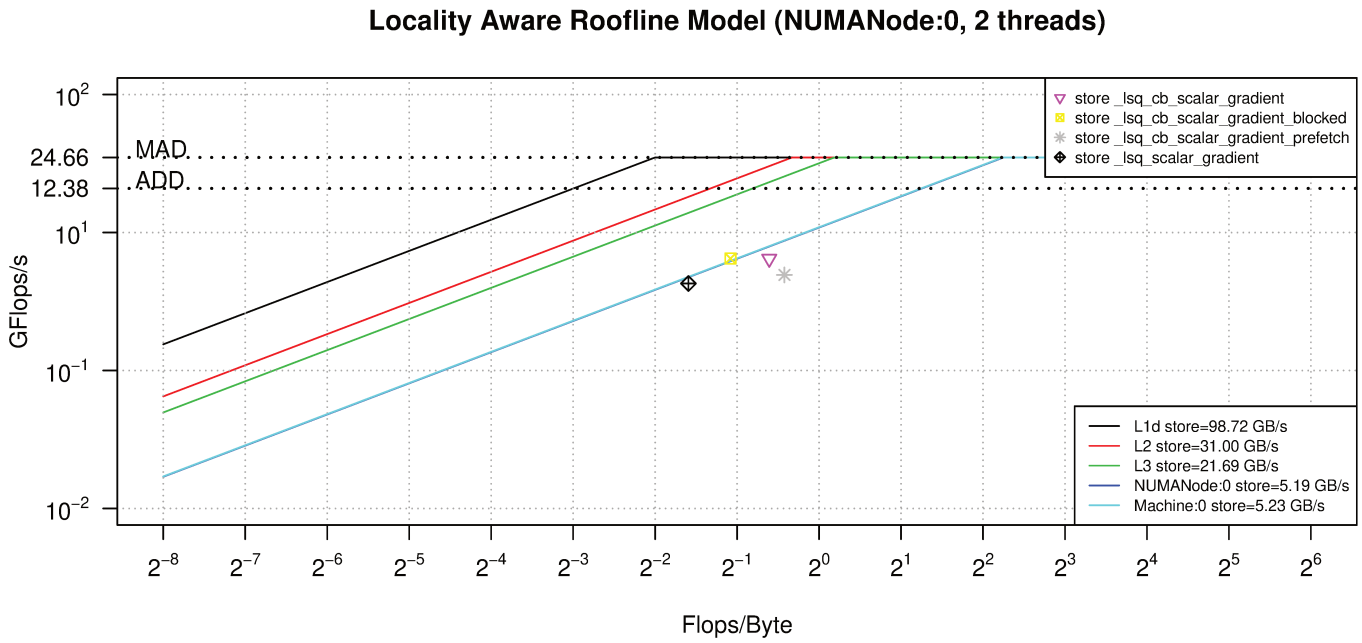


FIGURE 2.11: Performances de nos différentes versions de reconstruction de gradient (en écriture mémoire). Les conclusions sont les mêmes que pour la figure 2.10.

de la reconstruction des gradients est largement limitée par les performances mémoires de la machine et bien qu'étant assez faible l'intensité arithmétique de ce produit est donc déjà suffisante pour éprouver la machine. On ne peut donc ici pas s'attendre à une augmentation des performances via une augmentation de l'intensité arithmétique (transition de face par face à cellule par cellule).

	cell based	face based
GFLOPs/s	7.64	3.56
GFLOPs	35.507760	12.605210
I / O (doubles)	9 114 916	14 461 632
temps d'exécution (ms)	4.648	3.7427

TABLE 2.2: Comparaison des performances de nos deux méthodes de calcul.

Pour une exécution sur un seul processus, on obtient les résultats exposés en Table 2.2. L'algorithme en itération sur les cellules nous permet bien d'améliorer les performances (on les double par rapport à une itération sur les faces), car l'on multiplie quasiment par 3 le nombre d'opération flottantes tout en restant dans un ordre de grandeur similaire quant au temps d'exécution (pour 3 fois plus de calcul, le temps d'exécution sur un seul processus et pour un nombre de cellules égal à 65536 n'est que de 20% plus cher). Les données sont cependant moins misent à contribution puisqu'on approvisionne le processeur d'à peu près 1.5 fois moins de "doubles" pour obtenir le même résultat.

Finalement le calcul de gradients "par cellule" bien que peu intéressant sur les mesures présentées ici quant à ses performances en temps d'exécution permet néanmoins d'utiliser une plus grande partie de la puissance du processeur en plus de proposer une expression du calcul qui facilitera (on le verra en seconde partie) l'expression de la reconstruction des gradients sous forme de tâches. Ces résultats permettent aussi de justifier l'intérêt de l'approche "par cellule" en dehors du cadre prévu par notre modèle de départ, qui ne prend pas en compte la complexité ajoutée par les défauts de

caches par exemple.

II Contribution sur la localité des données de halos

Afin de pouvoir mettre à jour nos données aux frontières de chaque domaine de calcul, il est nécessaire de récupérer les données aux frontières des domaines voisins. Cette pratique commune est souvent discutée sous l'appellation d'échange des halos (ou encore ghost cells en anglais). Dans *Code_Saturne*, ces données supplémentaires sont stockées à la fin de chaque tableau de données sur lequel ces échanges sont nécessaires et sont indexées et ordonnées sur le rang MPI auxquels ils font référence. La figure 2.16 en page 38 reprend cette topologie et la détaille en prenant en compte la numérotation par courbe de remplissage.

II.1 Le produit matrice-vecteur en algèbre linéaire creuse

Le produit matrice-vecteur (ici en matrice creuse) est une étape importante dans une majorité d'applications scientifiques et prend part aux étapes du solveur itératif (le lecteur souhaitant s'introduire aux méthodes itératives en algèbre linéaire creuse est convié à consulter [91]) de *Code_Saturne* lors de l'utilisation du solveur de matrice creuse multigrille (méthode de résolution réputée pour sa scalabilité et qui peut aussi être appelée en étape de pré-conditionnement de la matrice). Il peut être nécessaire d'effectuer une mise à jour des halos du vecteur à chaque itération ou calcul du SpMV (Sparse Matrix Vector product) en fonction de ce que l'on résout. Actuellement les halos sont stockés à la fin du vecteur, entrainant de larges sauts dans la mémoire lorsque l'on y accède. Ce produit en matrice creuse étant connu pour sa très faible intensité arithmétique, ses performances sont avant tout régies par notre aptitude à l'approvisionner en données le plus rapidement possible. Cela est dû notamment à la représentation des matrices creuses en stockage, qui utilise fréquemment (et dans notre cas) une représentation via CSR[49] (compressed sparse row storage) ou autre (MSR, BCRS). Le but de ces représentations est de réduire en mémoire la taille des matrices ainsi compressées en tirant profit du fait qu'elles sont composées d'une multitude de zéros. Ce type de représentation est globalement composée de la matrice originale sous une forme 1D et sans les zéros. Un ou plusieurs indexes viennent ensuite compléter cette table pour spécifier les positions originelles de chaque élément de la table 1D, permettant ainsi de reconstruire la matrice 2D initiale. Plus la matrice est creuse et plus le format est rentable. Le format possède les problématiques inhérentes à toute transformation de la structure de donnée qui rajoute une indirection (index), d'où son impact sur les performances.

II.2 Contribution : relocalisation des halos

Afin de fluidifier les accès mémoires on souhaite donc améliorer la localité des halos afin de diminuer les défauts de caches lors du calcul du produit matrice-vecteur. Hors si l'on se replace dans le contexte d'un partitionnement par fractale de remplissage (SFC de type Morton ou Hilbert, voir figure 1.3 page 17, et que l'on associe à ce partitionnement une numérotation des éléments via la même SFC, on constate alors que l'on peut déterminer dans le code quelle zone de notre maillage aura besoin des cellules voisines [4]. Les figures 2.12,4.5 et 2.16 détaillent les éléments de compréhension correspondants.

II.3 Résultats

On compare en figure 2.13 la vitesse des deux implémentations en fonction du type de produit, du nombre de processus et de la localité des halos. On observe un meilleur résultat du produit avec

4. Lorsqu'on se met dans le contexte d'un partitionnement via SCOTCH ou METIS, le caractère non reproductible et aléatoire fait qu'il est impossible de traduire de manière algébrique l'ordre du partitionnement (permettant de relocaliser au plus proche un halo de là où seront stockées les cellules adjacentes) constaté visuellement post simulation.

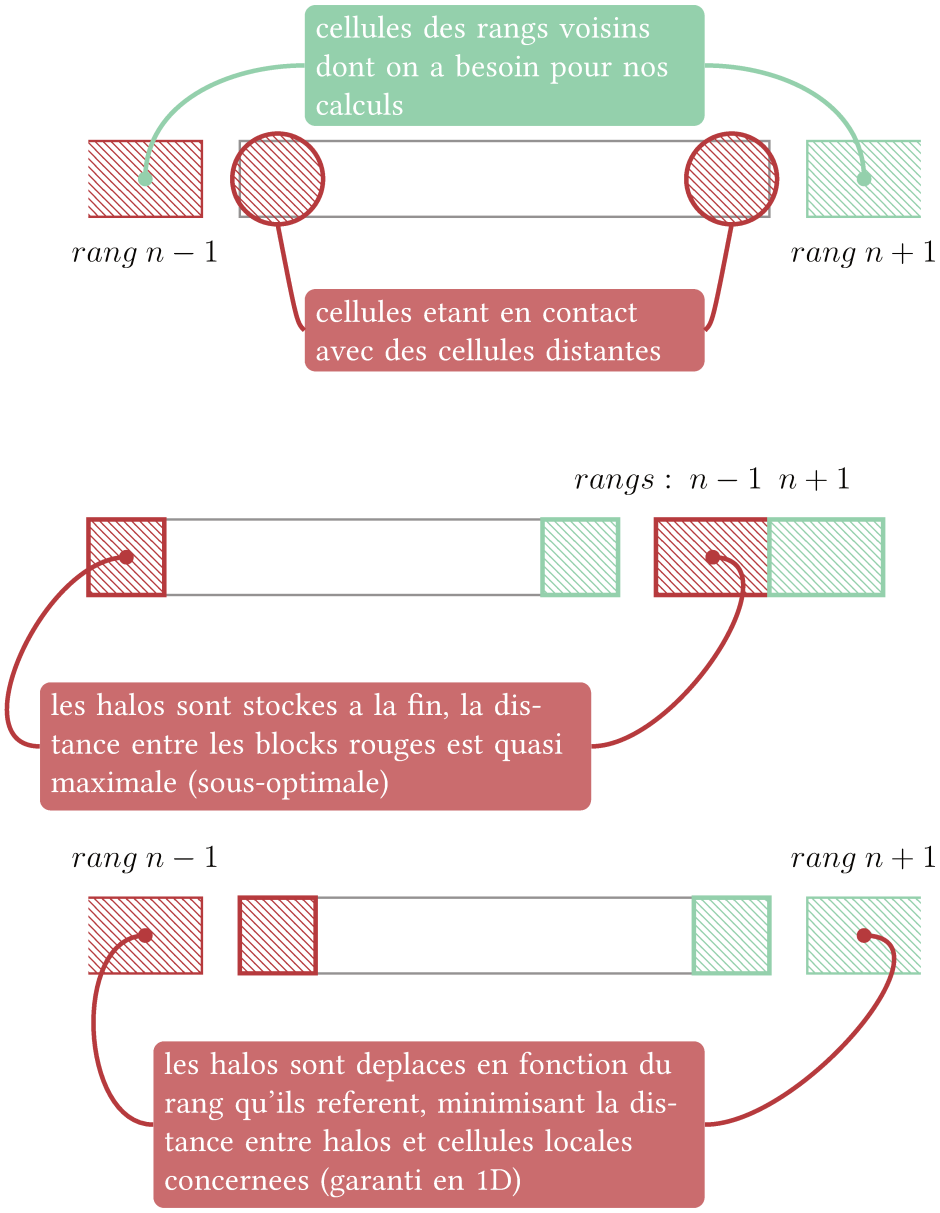


FIGURE 2.12: En 1D, le problème de localité entre numérotation des cellules et localité avec les halos se voit directement (en haut la description du cas d'un maillage 1D réparti sur plusieurs processus, puis comment sont stockés les données sur les cellules et enfin une ébauche de ce qui pourrait être fait). En 2D/3D, il faut raisonner en partant du partitionnement en SFC jusqu'à la numérotation liée (voir figures suivantes).

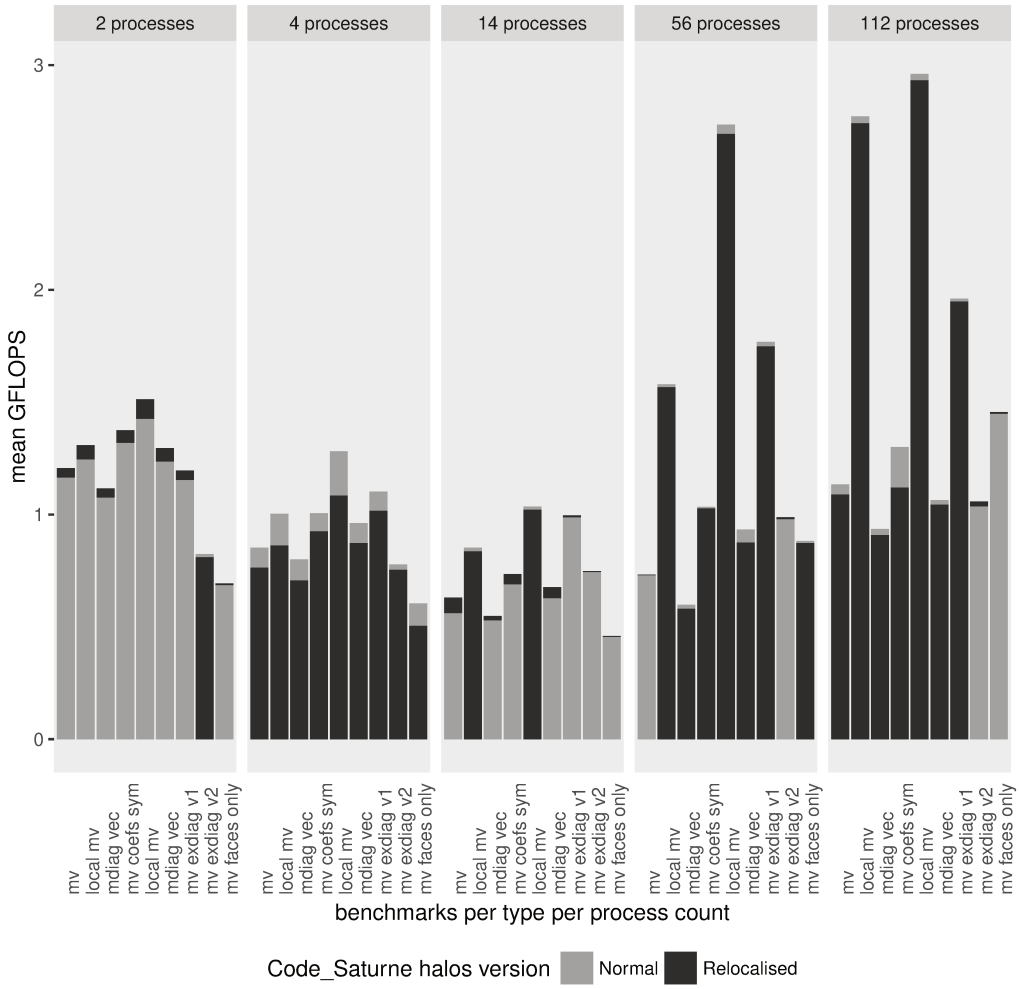


FIGURE 2.13: Résumé des performances obtenues avec et sans déplacement des halos, en fonction du ratio de cellules par processus et du type de produit matrice-vecteur effectué.

les halos re-localisés lorsque le nombre de processus MPI est très faible (2 processus, la taille du maillage utilisé ne varie pas en fonction du nombre de processus et reste à 1 million de cellules). Dans les autres cas, on observe peu de différence mis à part avec 4 processus, où les résultats 2.13 sont à l'inverse de ceux précédemment observés. Cela s'explique largement en raisonnant sur les schémas présentés en figure 2.12, 4.5 et 2.16 : si la procédure est intéressante lorsque le cas se limite à deux voisins par rangs, on observe déjà la limite du raisonnement du fait de la numérotation potentiellement utilisée (une SFC ne garanti en aucun cas une bonne localité des cellules nécessitant les halos : elles peuvent être numérotées de manière éparse et irrégulière sans spécialement s'agglutiner en début ou fin de tableau). Ce comportement est ensuite amplifié avec l'augmentation du nombre de voisins (et donc à un passage en 2D et 3D).

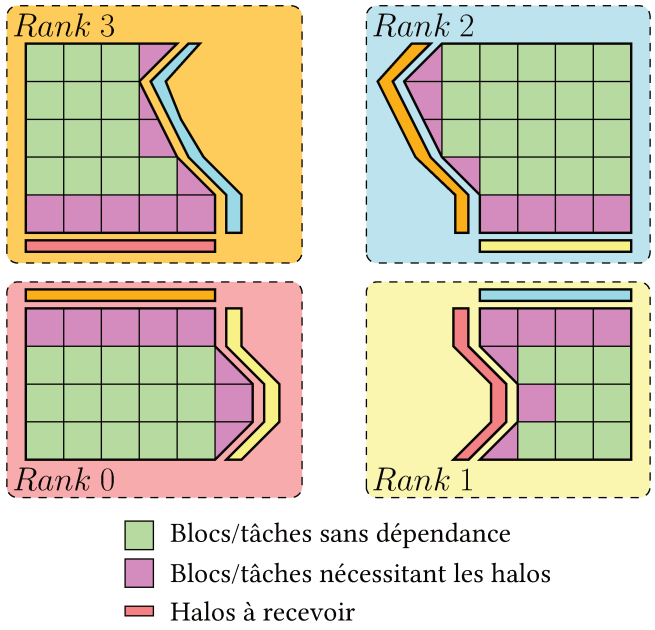


FIGURE 2.14: Vue d'un maillage partitionné pour 4 rangs MPI. On fait une correspondance entre tâche et bloc pour la reconstruction de façon à ce qu'une tâche de reconstruction corresponde à un bloc de cellule. On peut alors déterminer les dépendances entre tâches en fonction du voisinage de son bloc de cellule.

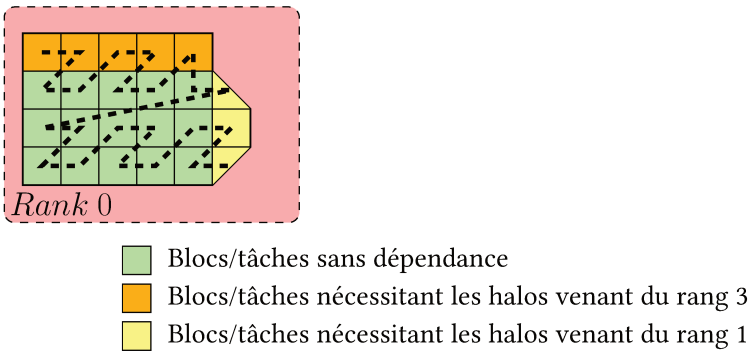


FIGURE 2.15: Spécification pour chaque bloc de cellules des halos nécessaire, de manière numérique et en accord avec les indexes des cellules hors locales (c.f. la figure 4.6). En traçant par dessus une SFC grossière, on visualise aisément l'aspect irrégulier des accès aux halos (et aux cellules en général). Une version raffinée de cette courbe (qui passerait de cellule en cellule) respecterait la même géométrie générale. On constate les allers retours constants entre partie purement locale et partie nécessitant les halos. Une transposition 1D de la numération qui en découle nous montrerait la disparité de la répartition des cellules en bord de maillage.

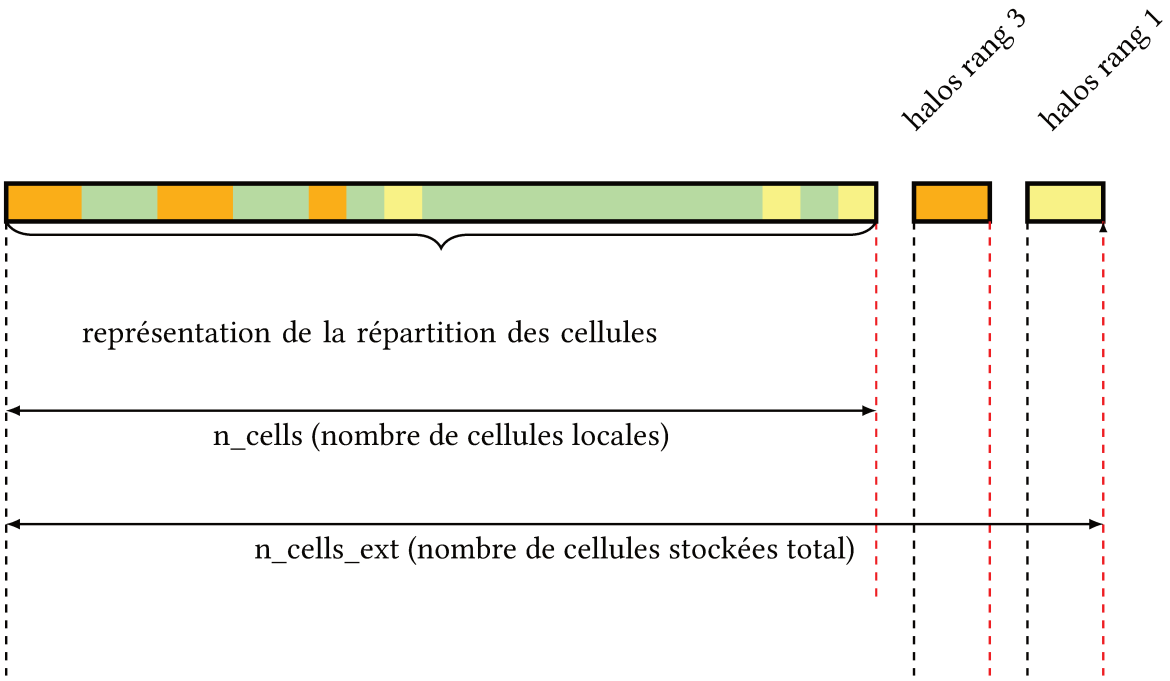


FIGURE 2.16: Vue de la localisation des cellules nécessitant les halos, en fonction des voisins. On observe une certaine disparité dans leur disposition du fait de la numérotation (c.f. Courbe de remplissage fractale en figure 4.5). La distance entre cellules locales et distantes lorsqu'elles sont en relation directe est maximale pour les cellules locales indexées en tout début. Si la quantité de données n_cell_ext ne rentre pas en cache, on va alors perdre beaucoup de temps en défauts de cache.

III Perspectives

III.1 Mise en cache régulière des futurs halos à accéder

Une solution envisageable qui pourrait apporter de meilleurs résultats que la relocalisation des halos serait de maintenir manuellement un cache anticipant pour chaque rang les indices à cacher. On peut ainsi partir du même constat établi, et pré-charger en cache la partie jugée comme susceptible d'être utilisée en fonction du numéro de cellule en cours de calcul. Lorsque le cache ne contient pas cette donnée, on remplit ce dernier avec une quantité "suffisante" de halos en évinçant si nécessaire les données du cache les plus anciennes en utilisation.

III.2 Mini application

Le développement des mini applications bien que peu répandues ont un certain intérêt dans le cadre d'expériences multiples sur une application de base. Une telle mini application permettrait de cloisonner l'expérience à la partie unique où elle est développée, permettant ainsi de manière simple d'en évaluer l'intérêt [5]. Le projet Mantero[62] justifie leurs intérêts pour de multiples raisons dont les plus intéressantes pour nous sont les suivantes :

- les suites de benchmarks HPC actuelles se limitent à des algorithmes très répandus (cf BLAS), et hors de ce spectre là, les outils permettant d'évaluer les performances de plusieurs variantes d'un algorithme et sur plusieurs machines sont proches de l'inexistant
- comparer différents langages rapidement
- étude de passage à l'échelle (qui demandera probablement une génération de donnée paramétrée, ou une l'extraction des données depuis des maillages existant (pour ensuite les charger dans la mini application))

Dans le cadre d'une application de simulation utilisant des maillages non structurés, c'est à dire dont le comportement est hautement variable, la génération de données paramétrée semble peu approprié car peu représentative.

Puisqu'il s'agirait de tester des parties très spécifiques du code, une solution possible consisterait en la sortie de cette partie du code et la transition vers des structures de données dont la définition serait fixée par un format de sérialisation tel que protobuf par exemple (le choix doit être fait en fonction du nombre de langages supportés ou de la facilité à ajouter le support pour tout autre langage. La vitesse de sérialisation et de-sérialisation n'est pas un facteur important puisque hors de l'évaluation des performances). On pourrait alors exporter les données d'une simulation pour une configuration donnée (un nombre de processus et une taille de maillage), puis les charger en amont dans notre mini application avant d'exécuter notre algorithme extrait. Cela permettrait a la fois de pouvoir exploiter de vraies données, tout en n'ayant pas à se préoccuper de l'aspect IO. Le choix du format de sérialisation est cependant important puisqu'il définira les langages sur lesquels ont pourra expérimenter. Dans notre cas, les dits langages étant souvent des additions au C/C++, on peut ne pas être trop impacté. Une solution plus intéressante qu'un format de sérialisation générique serait dans notre cas le format VTK[94] qui est déjà utilisé par *Code_Saturne* (fonctions d'export des données de simulations déjà disponibles). Ce format est la pierre fondatrice du logiciel de visualisation Paraview[7], ce qui permettrait par la même d'espérer pouvoir exploiter les résultats de la mini application et donc d'explorer différents comportements avec le confort d'une application de visualisation de maillages. Les informations de partitionnement doivent cependant être gérées ou maintenues (il est possible de stocker n'importe quelle donnée pour chacune des cellules/sommets

5. Ce point est discutable, car si il devient plus simple d'avoir un rendu positif de notre expérience, on n'a cependant pas un échantillon représentatif du vrai impact puisqu'on ne prend pas en compte les effets de bord sur le reste de notre application.

du graphe en sortie (et donc de renvoyer le numéro du rang possédant l'entité), on peut alors partir de cette information pour reconstruire un maillage distribué pour *Code_Saturne*.

III.3 Modélisation des performances

Une fois un modèle de mini application faite, modéliser les performances devient alors plus simple à la fois à vérifier et à mettre en place. La mise en place d'une modélisation des performances précise est une étape importante pour déterminer la granularité optimale des calculs en fonction du type des unités de calcul à notre disposition. [79] détaille une approche pour trouver la taille de bloc optimale à leur algorithme sous forme de tâche. La complexité de *Code_Saturne* tend à motiver l'investigation de modélisation semi-automatiques via l'utilisation de réseaux de neurones (c.f [77] pour une revue des méthodes actuelles).

Deuxième partie

Transition vers une reconstruction des gradients par tâche

3 | Paradigme de tâches et supports exécutifs : Introduction

I Introduction

I.0.1 Graphe de tâches

La théorie des graphes est un des fondements de l'informatique. Elle est l'étude des graphes, une représentation abstraite par laquelle on établit des relations entre un ensemble d'objets. On parle alors de sommets pour ces objets, et d'arêtes entre les sommets lorsque deux sommets sont liés par une quelconque relation. On utilise alors la notation $G = (V, E)$, l'ensemble des nœuds (sommets) V et E l'ensemble des arêtes du graphe G. Un des algorithmes les plus connus qui fait appel à cette discipline est l'algorithme de Dijkstra, ou algorithme de plus court chemin, et qui vise à minimiser la traversée d'un graphe d'un nœud A à un nœud B. Cet algorithme est notamment utile pour le calcul d'itinéraires routiers. Pour un graphe de tâches G, chaque nœud V représente un ensemble de fonction. Les arêtes quant à elles signifient une nécessité de synchronisation ou d'ordre d'exécution entre des appels de fonctions provenant des deux nœuds ainsi reliés. Si les fonctions des deux nœuds n'utilisent pas la même mémoire, l'arête peut aussi symboliser un échange de donnée nécessaire du premier nœud au second.

I.0.2 Paradigme de tâche

Son utilisation en tant que paradigme de programmation consiste donc en la mise en relation de blocs de code afin d'obtenir une représentation sous forme de graphe pour une partie de notre application. Ce graphe est orienté, de telle manière à pouvoir exprimer la nature (entrante ou sortante) de chacune de nos dépendances. La figure 3.1 représente le graphe de tâches correspondant à

l'appel du pseudo-code séquentiel du calcul de la suite de Fibonacci (pseudo code 1 ci-dessous). Ici, chaque tâche représente un appel à la fonction "Fibonacci".

```
1: procedure Fibonacci(n)
2:     if n<2 then return n
3:     else
4:         return Fibonacci(n-1) + Fibonacci(n-2)
```

Pseudo-code 1: Calcul séquentiel (non optimal) du nombre de Fibonacci

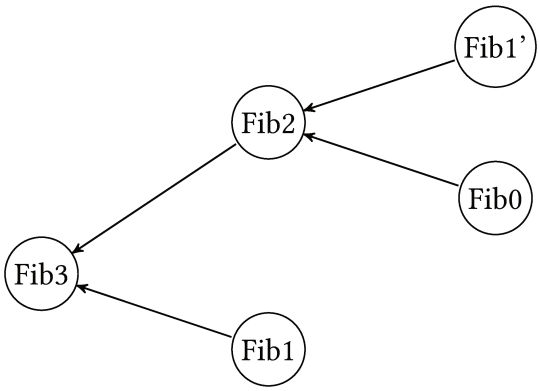
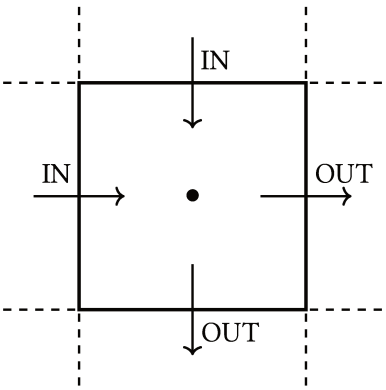


FIGURE 3.1: Graphe de l'exécution séquentielle de Fibonacci(3). On observe qu'au mieux, deux appels de Fibonacci peuvent être exécutés en même temps.

La figure 3.2 montre un autre exemple de graphe de tâches pour un balayage cartesien. En mettant à jour continuellement ce graphe (en supprimant les nœuds et arêtes déjà exécutés), on peut visualiser le degré de parallélisme de l'algorithme correspondant à un instant "t" donné en dénombrant chaque nœud de degré entrant nul (nœuds n'ayant plus d'arêtes entrantes non résolues). On sait alors que ces derniers sont l'ensemble des tâches qui peuvent être ordonnancées dès à présent.

L'intérêt de cette pratique est multiple. Non seulement elle recentre le développeur sur l'expression du parallélisme via le choix de l'algorithme et la représentation des données liées à son problème, mais elle permet aussi la libération à moindre coût d'une partie du parallélisme. En effet, en se focalisant sur l'expression des dépendances et en laissant la main à un support exécutif sur l'ordonnancement (a minima) de nos tâches, on obtient ainsi un parallélisme avec un degré de liberté largement supérieur à un modèle de type BSP (Bulk Synchronous Parallelism) auquel correspond une grande partie des applications scientifiques sur base de MPI+X.



$T5$	$T7$	$T8$
$T2$	$T4$	$T6$
$T0$	$T1$	$T3$

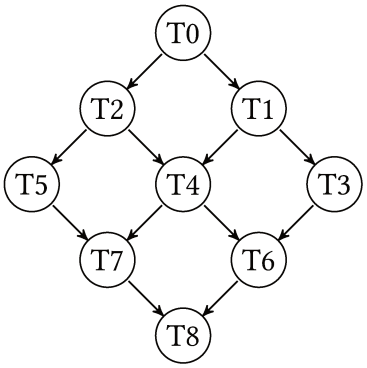


FIGURE 3.2: Représentation d'un algorithme de balayage 2D cartésien. Ici, la répartition des tâches peut être directement assimilée à la géométrie du maillage. Les dépendances en lecture et écriture de chaque blocs étant les mêmes, il est alors trivial d'en déduire un graphe de tâche (non optimal).

I.0.3 Dépendances de tâches

Puisque la seule raison valable de ne pas exécuter deux blocs de code de manière concurrente est une dépendance en donnée entre les deux, on exprime les différentes dépendances entre tâches pour A et B de la manière suivante (conditions de Bernstein [17]) :

— dépendance de donnée (A lit une donnée que B modifie) $B \rightarrow A$
— anti-dépendance de donnée (A modifie une donnée lue par B) $A \rightarrow B$
— dépendance de sortie (A et B modifient tous les deux une même donnée) : état de concurrence

On considère alors que les tâches A et B sont indépendantes si et seulement si aucune de ces dépendances n'existe, et ce pour toutes les entrées et sorties de A et B.

I.0.4 Degré de parallélisme

Le paradigme de parallélisme par tâches permet d'exprimer notre degré de parallélisme comme étant l'ensemble des tâches T_i à un instant T qui sont libres de toute dépendance, et qui sont donc exécutables de manière simultanée. Ainsi, sur la figure 3.1, même si le graphe correspond à l'algorithme séquentiel, il permet de visualiser tout de même la quantité de parallélisme extractible de cet algorithme. Dans cet exemple, on voit que le graphe généré correspond à un arbre dont la racine correspond à la fin de notre exécution. La racine ne peut qu'être en possession de deux fils maximum, eux-mêmes en possession d'au plus deux fils et ainsi de suite. Chaque fils ayant besoin d'attendre son confrère, on se retrouve alors avec un degré de parallélisme maximal égal aux nombres de fils au dernier niveau de l'arbre, puis une nette réduction d'en moyenne la moitié du degré à chaque niveau supérieur de l'arbre. On peut dès alors établir le manque d'équilibrage de charge d'un tel algorithme s'il advenait à être parallélisé sous cette forme.

I.0.5 Localité du graphe : graphe global et local

Lorsque l'on ordonnance un graphe d'exécution sur un ensemble de processus, il est important de visualiser à quel niveau l'on se positionne. Les architectures récentes étant composées de plusieurs couches de parallélisme, on peut adapter à chacune de ces couches une vision différente. D'un point de vue logique, on s'attend à ce qu'en amorçant une transition vers le paradigme de tâche, tout devienne tâche mis à part les échanges de données qui, eux, ne deviennent plus qu'une dépendance d'une tâche vers une autre et ne doivent pas être effectués au niveau utilisateur. Cette logique-là est celle d'un graphe global. Pour fonctionner, il est nécessaire à chaque unité de calculs du plus haut niveau de connaitre l'ensemble du graphe (un nœud, un processus MPI), de telle sorte qu'il puisse tout simplement savoir avec qui il doit effectuer les communications nécessaires à la réalisation de ses tâches personnelles. Vient alors une notion de possession de données qui fait suite à la construction du graphe par dépendances. La figure 3.3 montre ce que pourrait être un graphe global et sa répartition sur l'ensemble des unités de calcul.

D'un autre côté, ce modèle impose certaines contraintes. Chaque processus doit connaître tout ou partie du graphe global, rajoutant ainsi un surcoût en mémoire qui est fonction de la taille du graphe, et donc, d'une part de la quantité de travail à effectuer (taille du maillage entre autre) mais aussi du nombre d'unités de calcul (et donc, nombre de voisins à chaque unité) et de la granularité des tâches. Or une simulation moyenne dans *Code_Saturne* fait appel à ces trois paramètres dans des conditions impactant grandement la taille du graphe. La granularité optimale d'une simulation de *Code_Saturne* avoisinant les 10 000 cellules par cœur, et les maillages atteignant une taille classique entre 50 et 200 millions de mailles (cellules), la taille du graphe global, s'il devait être connu et donc stocké par chaque processus, pourrait être un frein à l'utilisation d'un tel paradigme.

Pour pallier ce problème, une approche hybride peut être considérée. En limitant le champ de vision de chaque processus à son propre graphe, et en considérant les échanges de données inter-nœuds comme étant des tâches à part entière, on réduit d'autant les besoins en stockage du graphe

pour chaque processus. On peut alors parler de graphe local. Les échanges de donnés sont explicites, et inclus ou non dans des tâches (c.f. Figure 3.3). Différents mécanismes peuvent être imaginés pour gérer les dépendances en données vers l'extérieur en fonction des supports exécutifs utilisés (dont on en détaillera certaines en section I.3).

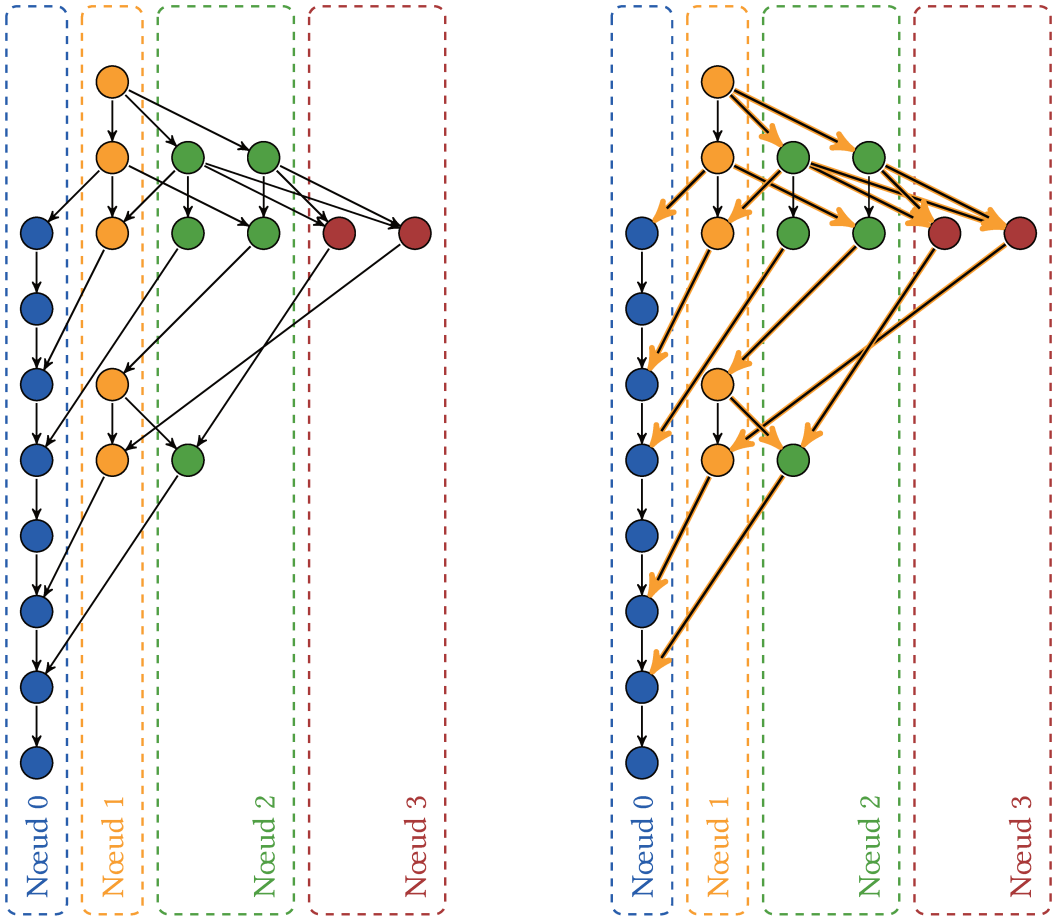


FIGURE 3.3: Exemple grossier d'un graphe global (gauche) et d'un graphe local (droite) d'un algorithme (Cholesky). Une dépendance entre deux tâches sur deux nœuds implique un mouvement de donnée. Une dépendance entre deux tâches d'un même nœud définit avant tout un ordre d'exécution. Lorsque le graphe est global chaque nœud doit avoir la vision sur les tâches de ses voisins (ou plutôt des besoins en données). Lorsque le graphe est local l'exécution peut ne pas différer mais les nœuds n'ont pas connaissance de dépendances distantes. La gestion des échanges doit être explicite et potentiellement hors du support exécutif via MPI. Dans ce cas, la dépendance sur la réception doit être émulée par le développeur.

II Paradigme de tâches et support d'exécution

II.1 Modèles de paradigme de tâche

Il existe différentes façons d'introduire la notion de tâche en programmation. À l'aide de la théorie des graphes, il est possible d'exprimer les tâches sous forme de graphe. On distingue alors deux manières de construire ce graphe, l'une appelée séquentielle parce que le graphe est défini suivant l'ordre d'insertion des tâches créé par l'utilisateur et la manière paramétrée (Parameterized Task Graph) où le graphe est déduit simplement par l'ensemble des dépendances exprimées. La logique de tâche est aussi aisément transférable sur une logique de boucle où chaque bloc d'itération serait alors une tâche.

II.1.1 Modèle de tache par boucle

Sûrement le modèle le plus simple au premier abord mais aussi le plus compliqué à mettre en valeur en terme de performance, le modèle de tâches par boucle propose souvent un mécanisme de parallélisation par l'utilisation de simples additions au langage de programmation initial. Sous la forme de pragmas, ces instructions permettent d'opérer diverses opérations (réduction, min, max) et de jouer sur la granularité et la localité des données. C'est un des modèles offerts par les supports OpenMP et Cilk notamment mais aussi des langages de type PGAS (Partitionned Global Address Space) tels que Chapel qui généralisent cette pratique à un ensemble de machines. Ce type de parallélisme de tâches peut être extrêmement puissant lorsque le type d'algorithme utilisé s'y prête. Dans notre cas, un tel modèle semble largement inadapté du fait de son manque de flexibilité.

II.1.2 Modèle séquentiel

L'expression d'un code sous forme de tâches par voie séquentielle sous-tend une construction du graphe continue, au fur et à mesure que l'on insère des tâches. Cette démarche implique une insertion de la tâche décrite en dur dans le code via un appel de fonction non bloquant (c.f. Pseudo-code 2). Comme la tâche en question peut être de granularité variable, on peut imaginer un graphe de tâches correspondant à un code séquentiel où chaque tâche T serait identique à chacun des appels de fonctions présents dans la version séquentielle. On obtient alors une suite d'insertion de tâches similaire au code séquentiel.

```
1: procedure Fibonacci(n)
2:     if n<2 then return n
3:     else
4:         i ← InsertTask(Fibonacci, n − 1)
5:         j ← InsertTask(Fibonacci, n − 2)
6:
7:         Wait for completion of both tasks
8:         return i+j
```

Pseudo-code 2: Calcul (non optimal) du nombre de Fibonacci parallélisé par tâches, modèle séquentiel

De ce fait, il est nécessaire (pour le développeur et le support exécutif) de connaître et de faire des choix quant à l'ordonnancement des tâches de son graphe de manière à soumettre ses tâches dans l'ordre qui suit au mieux le déroulement optimal de son algorithme. Faillir à cet exercice peut limiter l'expression du parallélisme et sous-exploiter la machine (via une soumission trop tardive de

certaines tâches pourtant disponibles à l'ordonnancement ou étant clés pour libérer les dépendances d'autres tâches) [1].

Ce défaut est aussi une qualité car ce modèle, en étant proche de l'exécution d'un algorithme séquentiel, permet une transition spontanée de ce dernier vers ce modèle de tâche. Or la parallélisation d'un code via l'utilisation du modèle BSP via MPI consistant essentiellement à l'exécution en parallèle de code séquentiel et à des échanges de messages réguliers, l'effort à fournir pour cette transition peut s'avérer faible.

II.1.3 Modèle paramétré

On peut néanmoins s'abstraire des problématiques d'insertion de tâches au bon moment en se focalisant sur les besoins en lecture et en écriture de chaque tâche. On sait ainsi rapidement qui a besoin de qui pour réaliser son travail. Cette logique est utilisée dans le modèle paramétré [38]. On peut voir avec le pseudo-code 3 le calcul du nombre de Fibonacci avec cette méthode. Si la méthode semble inadaptée pour l'expression d'un algorithme possédant un seul type de tâche, il permet néanmoins à grande échelle de largement simplifier la façon de représenter un calcul. En effet, en centralisant la vision au niveau d'une tâche et de ses besoins, on renforce la localité de la réflexion en permettant au développeur d'oublier ce qui est inutile à la rédaction d'une solution parallèle (i.e. le reste de son code). L'interaction entre tâches se fait au moyen de la description des flux d'entrées et de sorties (A, B et C pour les flux, sections READ et WRITE de l'algorithme). Ce flux est exprimé généralement pour un indice i correspondant à la tâche T_i. Ici on observe que la tâche $Fibonacci_{(i)}$ contribue à la tâche $Fibonacci_{(i+1)}$ et $Fibonacci_{(i+2)}$, et récupère les données A et B provenant respectivement de $Fibonacci_{(i-1)}$ et $Fibonacci_{(i-2)}$. Cette description de flux à elle seule suffit à déterminer le graphe de tâche correspondant à notre algorithme.

Puisque le graphe de tâches est déductible par les flux de données de chaque tâche, il est constructible à tout instant, et peut donc être réévalué en cours d'exécution, permettant ainsi une série d'optimisations et d'heuristiques en vue d'améliorer les performances de cette exécution spécifique. Cette méthode offre donc une plus grande flexibilité tout en requérant une rigueur liée au modèle. En effet, les indices de tâches étant difficilement variable, les domaines d'exécution des tâches doivent être déterminés dès la rédaction, de telle sorte que le graphe de tâches soit instanciable dans sa totalité dès le départ tandis que le modèle séquentiel n'impose aucune contrainte.

Compte tenu des spécificités liées au calcul scientifique comme on peut les trouver dans *Code_Saturne*, c'est-à-dire faisant appel à des maillages de type non structuré pour la simulation de phénomènes physiques, l'on s'attend à ce que le modèle séquentiel soit plus adapté à nos besoins que le modèle paramétré. L'indexation des entités telles que les cellules et les faces imposées par la nature variable des éléments limite les raisonnements présentés en figure 3.2 car le nombre d'entrées et de sorties ne peut être déterminé qu'au cours de l'exécution et non pas de manière préalable. Si l'on se situe au niveau supérieur et que l'on considère des blocs d'éléments, voire un sous domaine, le problème reste sensiblement le même du fait de l'étendue possible des maillages traitables et du choix de partitionnement (à courbe de remplissage, ou de type PT-SCOTCH ou PARMETIS). De même, le modèle de tâches par boucle semble très loin de la façon dont nos algorithmes sont exprimés.

Malgré ce constat, nous estimons qu'il est important d'évaluer d'un point de vue pratique au moins deux des trois modèles dans un contexte de maintenance de *Code_Saturne*, un code de calcul industriel déjà largement parallélisé afin d'éprouver leur qualités respectives. Si l'exercice a déjà été réalisé quant à l'implémentation d'une solution ou de l'autre dans un code de calcul[80, 19], il n'y a eu à notre connaissance que peu de comparaison des deux modèles sur un cas précis industriel.

1. En pratique, l'impact de l'ordre d'ordonnancement est aussi et surtout fonction du nombre de tâches soumises et du graphe initial.

```
1   Fibonnaci(N)
2
3   n = 1..N
4   READ A (n >= 2) ? ← C Fibonnaci(n−1) : 1
5   READ B (n > 2) ? ← C Fibonnaci(n−2) : 0
6   WRITE C (n < N) ? → C Fibonnaci(n+1)
7   WRITE C (n < N−1) ? → C Fibonnaci(n+2)
8
9   BODY
10  If (n < 2)
11    C ← n
12  Else
13    C ← A + B
14  END
```

Pseudo-code 3: Calcul (non optimal) du nombre de Fibonnaci parallélisé par tâches, modèle paramétré. On définit en ligne 3 pour n le domaine d'exécution de l'algorithme, c'est-à-dire les valeurs possibles que la variable peut prendre. Cela définit directement le nombre de tâche dont notre graphe sera composé. Si nous avions d'autres variables définissant le domaine d'exécution, le nombre total de tâche créé serait alors le produit de chacune de leur valeur maximale. Les lignes 4 à 7 définissent les arêtes de notre graphe en spécifiant qu'elles sont les entrées d'une tâche en fonction de n et qu'elles sont les données qu'elle produira, et à l'attention de qui. Une condition tertiaire permet de spécifier les valeurs initiales du calcul de la suite pour les tâches $n == 1$ et $n == 2$. On fait référence aux valeurs d'entrées et de sorties grâce à un identifiant (ici A, B et C). Pour la tâche $n == 2$, le code se lit de la manière suivante : mettre dans A la valeur C produite par la tâche $n == 2 - 1 == 1$, mettre dans B la valeur 0 (car la condition $n > 2$ n'est pas vérifiée), envoyer le résultat de $A + B$ aux tâches $n == 2 + 1$ et $n == 2 + 2$.

II.2 Supports exécutifs

Afin d'implémenter une partie de *Code_Saturne* en tâche, il est nécessaire de faire appel à un nouveau support d'exécution. Appelé runtime system en anglais, ce terme fait référence à toute bibliothèque permettant au développeur de déléguer une partie de la gestion de l'exécution du programme. Les implémentations de MPI telles qu'OpenMPI[54] ou IntelMPI ou autre, peuvent donc être considérées comme étant elles-mêmes un support exécutif puisque mettant à disposition les outils pour développer en parallèle de manière quasi transparente. Le rôle d'un support exécutif diffère selon l'ambition qu'il nourrit, parmi lesquelles on peut dénombrer notamment les solutions telles que OpenMP[14] et Cilk++[72] qui permettent un parallélisme à base de processus léger ou tâches via une modification du code minimale, CUDA[68] et OpenCl[97] qui assurent avant tout la gestion des architectures de type GPGPU via une API, tandis que PaRSEC[26], StarPU[12] offrent un parallélisme de tâche respectivement via un langage beaucoup plus formel que le C ou au contraire sans profonde rupture avec le style de développement usuel. Enfin, les langages de type PGAS (Partitionned Global Address Space, UPC[37], X10[35] par exemple) s'attaquent à la représentation en mémoire distribuée partagée. Il existe donc une multitude de solutions, quand bien même on souhaiterait se limiter à se diriger vers un support exécutif de tâche.

Divers critères nous intéressent a priori particulièrement quant au choix de support, parmi lesquels :

— le modèle utilisé (paramétré, séquentiel, boucle)
— les architectures supportées (accélérateurs)
— l'aspect de la solution (API, source à source, annotations)
— le niveau d'abstraction de la gestion mémoire (mémoire partagée, distribuée)

D'autres critères nous intéressent, mais ne peuvent être vraiment appréciés qu'a posteriori, notamment :

— la facilité d'implémentation
— la capacité de développement incrémental
— la flexibilité
— la documentation
— la facilité du déboggage
— les performances
— l'activité du projet
— la réactivité de l'équipe
— la pérennité du support exécutif

Nous détaillons ci-dessous une partie de ces derniers.

Capacité de développement incrémental Cet aspect est important et regroupe deux problématiques. La première est que partant d'une application déjà conséquente en taille et déjà parallélisée à l'aide de MPI, la transition ne peut se faire que partie après partie. D'une part car il n'est peut être pas souhaitable de tout changer mais éventuellement de n'ajuster que les parties concentrant la majorité du coût en temps du logiciel et d'autre part parce qu'il est avantageux de pouvoir maintenir l'application en état de fonctionnement durant l'entièreté de la transition.

La deuxième problématique est sur le long terme. Il est par défaut plus intéressant de choisir un support permettant l'exploitation d'un maximum d'architectures différentes, ne serait ce que dans l'expectative d'un futur déploiement. Cet aspect rejoint en partie l'importance liée à l'activité du projet ainsi que sa pérennité et par conséquent la capacité de l'équipe à suivre les évolutions matérielles informatiques.

Flexibilité Certains supports ont une approche ambigüe sur la façon dont le parallélisme par tâches est utilisable et permettent différentes approches au sein de la même bibliothèque. Compte

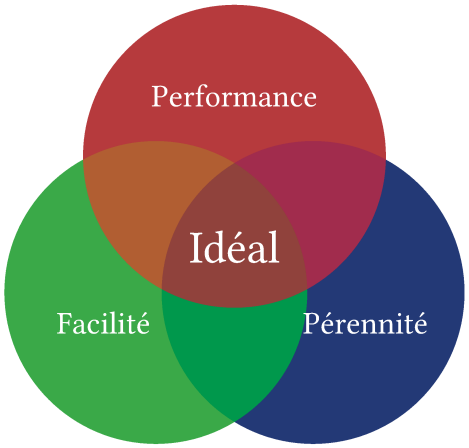


FIGURE 3.4: Le support exécutif se situe quelque part à l'intersection des performances qu'il offre, de sa facilité d'utilisation et de sa pérennité

tenu de la multiplicité des algorithmes et de leur nature, il est encore une fois souhaitable de maximiser les façons de résoudre un même problème. Trop de possibilités peut cependant entacher la lisibilité et de la documentation et du futur code produit. On peut aussi argumenter en faveur d'une approche qui n'offrirait qu'un seul modèle dans la mesure où l'on saura ce dernier parfaitement maîtrisé.

Performances C'est la condition la plus évidente mais qu'il faut cependant pondérer par l'ensemble des autres paramètres imposés par le contexte industriel. La figure 3.4 synthétise l'alchimie idéale entre les différents critères de sélection d'un support exécutif de tâche. L'approche se veut pragmatique, le support doit nous permettre d'être meilleur ou d'approcher les performances actuelles en simplifiant la maintenance du logiciel. Pour deux supports exécutifs similaires en performance à 10% près, on privilégiera celui qui permet l'implémentation de nouvelles solutions en un temps minimal [2].

II.3 Taxonomie des supports exécutifs à tâches

La quantité de supports exécutifs étant conséquente, il peut devenir assez complexe d'en choisir un, spécifiquement dans un contexte industriel. On propose ici une classification d'une partie d'entre eux afin que le lecteur ait une vision représentative de l'état des supports exécutifs supportant le paradigme de tâches actuellement. Dans un premier temps, il est nécessaire de distinguer la sémantique offerte par le support pour exprimer du parallélisme. Nombre d'entre eux supportent plusieurs expressions mais tendent à exceller ou favoriser l'une plutôt que l'autre. Le choix de la sémantique est important puisqu'il permettra d'écrire des algorithmes dont le niveau de parallélisme, l'asynchronicité et la facilité à les exprimer pourront varier grandement. Il est ainsi assez commun qu'un algorithme soit simple à exprimer en utilisant un support éxecutif et devienne une corvée en en utilisant un autre. Ainsi, l'expression de tâches par boucle qui propose une solution simple (à un probleme simple) est surtout intéressante quant à sa rapidité potentielle de prise en main mais force l'utilisateur à se cantonner à de la parallélisation très similaire à ce qui se fait hors parallélisme de tâches (cf pthreads) et ne se différencie de ces supports que par la possibilité d'offrir du vol de travail. Essayer de recouvrir le coût des communications par des calculs dans ce cadre demande plus

2. On souhaite faire particulièrement la distinction avec les supports promettant de bonnes performances et un modèle exprimant des concepts de manière très simple. Ce type de support qui peut sembler trop beau pour être vrai limite souvent sa vitrine a présenter l'élégance et l'efficacité syntaxique de son modèle aux problèmes les plus simples. Un modèle potentiellement aussi restreint promet d'être beaucoup plus complexe lorsqu'il s'agira d'exprimer des algorithmes tels que ceux de *Code_Saturne*.

de gymnastique que ce que l'utilisation d'un graphe de tâches aurait permis. On note toutefois que le vol de travail offert par un tel support serait une bonne solution si l'on a une vision réduite des optimisations d'ordonnancement que l'on doit effectuer car on peut alors avoir une prise en charge automatique de cette irregularité dans les temps de calculs sans avoir à se préocupper d'une quelconque optimisation à opérer de notre part. On voit bien alors que le modèle de tâches par boucle peut ainsi être un sérieux compétiteur aux modèles paramétrés et séquentiels. Pour rappel, ce dernier offre au programmeur une parfaite maîtrise de l'ordre d'exécution de ses tâches, ce qui lors d'un développement parallèle d'un algorithme bien connu (cholesky, pour prendre un exemple, est un algorithme très souvent utilisé pour présenter les capacités d'un support exécutif). Si c'est un avantage conséquent lorsqu'aucune hésitation n'existe sur la façon de paralléliser un algorithme, le modèle paramétré permet d'offrir – à l'instar du modèle par boucle – aux développeur moins versés dans le parallélisme, un développement loin de ces préoccupations tout en arrivant à des performances similaires.

L'idée d'exprimer un algorithme sous forme de tâches est assez ancienne et marque par sa simplicité à transposer nos raisonnements de manière naturelle. Certains des supports visant cet effet ont largement contribué au calcul haute performance, comme par exemple Cilk[18, 72](Intel, 1994) qui est l'un des premiers supports à proposer une logique de vol de tâches avec efficacité, une technique encore aujourd'hui intéressante[82] lorsque la granularité le permet. OpenMP[40](1998) quant à lui, a été largement adopté par la communauté scientifique pour sa simplicité au premier abord. Permettant d'abord une approche des tâches assez transparente [3], il a depuis largement évolué et supporte actuellement des mécaniques de tâches beaucoup plus poussées ainsi que les architectures hétérogènes à base de GPGPU. Le modèle de tâches par boucle étant somme toute assez limité, les supports exécutifs se sont par la suite orientés sur d'autres pratiques et ce afin de répondre à divers critères dont on retiendra les suivants : abstraire le matériel à l'utilisateur, abstraire la notion de programmation distribuée, abstraire l'hétérogénéité des machines, se rapprocher d'une programmation quasi algorithmique (langage simplifié). Cette volonté générale de faciliter le développement des applications scientifiques (et peut être aussi d'améliorer leur portabilité) fait suite aux évolutions tant dans le matériel que dans les pratiques, notamment avec l'arrivée d'accélérateurs dédiés au calcul scientifique (Xeon Phi) mais aussi à l'adaptation des calculs pour profiter du parallélisme massif offert par les architectures GPGPU.

L'introduction des architectures NUMA a, par exemple, largement modifié la façon dont sont pensés les développements et, d'une manière générale, là où l'on se préoccupait avant du coût en calcul d'une opération se préoccupe-t-on maintenant du coût d'accès aux données nécessaire pour ce calcul. La disparité dans les temps d'accès à la donnée qu'a introduite les processeurs NUMA a ainsi forcé les développeurs d'applications scientifiques à se focaliser sur la localité des données [76, 82, 29, 27]. Dans ce cadre, le modèle de tâches paramétrique (PTG) [24, 42] semble être un format idéal. La description des tâches par leurs flux de données entrant et sortant permettrait en théorie de maximiser l'ordonnancement de ces dernières en fonction de leur localité, la connaissance de cette dernière étant au cœur même du support. Elle doit aussi pouvoir obliger le développeur à raisonner sur cette localité, ce qui est un atout majeur.

Plus récemment, les langages de type PGAS (pour Partitionned Global Adress Space) visent eux a répondre à la problématique de la programmation multi-nœuds (et de manière générale multi-entités). En considérant la mémoire comme étant un espace accessible/partitionné par/sur l'ensemble des unités de calcul, ils abstraient à l'utilisateur les besoins d'exprimer les communications entre nœuds. On dénombre de nombreuses solutions dont on retiendra les langages PGAS suivants : Chapel[34], X10[35], (il nous faudrait citer pour être plus exhaustif UPC, les Coarray et Global Arrays, SHMEM, Fortran 2008).

3. Le parallélisme de tâches par boucle peut être appréhendé sans réelle connaissance première en théorie des graphes par exemple, ni ne demande une grande réflexion sur les dépendances entre tâches de manière générale mis a part pour préserver la cohérence du calcul.

La gestion de l'équilibrage de charge a reçu de nombreuses contributions, l'augmentation drastique du nombre d'unités de calculs continuant d'augmenter l'importance d'un équilibrage de charge de plus en plus précis et efficace. Des supports et langages tels que Charm++[65], IntelTBB[87] utilise par Taggre[90], KAAPI[55] et XKAAPI[56] montrent l'importance de cet aspect du parallélisme actuel.

Enfin, les débuts de l'utilisation des accélérateurs de type GPGPU puis Xeon Phi ont eux aussi un certain nombre de contribution s'appropriant cette problématique en addition à d'autres aspects, et on peut citer entre autre OpenACC[103], StarPU[12], PaRSEC[26], OmpSs[13, 48], mais aussi OpenMP, Legion[15], X10 et Cilk++ (chez qui la prise en charge des accélérateurs existe mais que l'on dissocie car n'étant pas le but premier du support/langage en question).

Les tables 3.1, 3.2 et 3.3 [4] présentent une comparaison de divers supports exécutifs en fonction des caractéristiques qui nous semblent important de mettre en avant quant à nos besoins mais aussi à ceux de tout autre développeur d'application scientifique. Certains de ces supports sont présentés afin d'être exhaustifs mais ne sont pas recommandés comme à considérer (Cilk par exemple est déprécié dans les mises à jour récentes des outils Intel 2018). Les paragraphes ci-après détaillent notre attachement a telle ou telle caractéristique et notre choix en conséquence.

support	boucle	séquentiel	paramétrée
Chapel	●●	●	
Charm++			●●
Cilk		●●	
Cilk++	●●	●●	
Intel CnC	●		●●
Intel TBB	●	●●	
Legion		●●	
OMPSs	●	●●	
OpenACC	●●	●	
OpenMP $\geqslant 4$	●●		
OpenMP $\leqslant 3.1$	●●	●	
PaRSEC			●●
StarPU		●●	
Taggre	●		●●
X10	●●		
X-KAAPI	●	●●	

Table 3.1: Classification de supports exécutifs selon le modèle d'expression des tâches possible.

Mémoire distribuée et distribution *Code_Saturne* utilisant actuellement MPI, le code est donc structuré avec une gestion de du parallélisme distribué explicite. Les approches telles que PaRSEC et StarPU semblent donc plus appropriées si l'on souhaite minimiser l'impact sur la structure du code. En effet, une bonne partie de l'effort est fait dans *Code_Saturne* pré-simulation afin de répartir et distribuer correctement l'ensemble du maillage. Une distribution implicite semblerait plus adaptée à un code qui souhaiterait amorcer une transition sur du parallélisme inter nodal après s'être cantonné à du parallélisme intranoeud.

4. Les méthodes de livraisons de PaRSEC semblent ici jouer en leur défaveur. L'activite réelle du dépôt est régulière et récente en 2018.

support	mém. distribuée	hétérogénéité	distribution	synchronisation	communication
Chapel	○		○	●	PGAS
Charm++	○		○	○	MSG
Cilk++	○	○		●	SMEM
Intel CnC				○	SMEM
Intel TBB				○	SMEM
Legion	○	●	○	●	PGAS
OMPSs		○	○	○●	SMEM
OpenACC		○		○●	SMEM
OpenMP $\geqslant 3$		○		○●	SMEM
PaRSEC	●	●	●	○●	MSG
StarPU	●	●	●	●	MSG
Taggre				??	SMEM
X10	○	○	○	○●	PGAS
X-KAAPI				○●	SMEM

Table 3.2: Classification de supports exécutifs selon le modèle d'expression des taches possible (implicite ○, explicite ●).

support	date introduction	avant derniere sortie	derniere sortie
Chapel	2009	Mai 2018	Sep. 2018
Charm++	1980s	Oct. 2017	Dec. 2018
Cilk	1994		Deprecie
Cilk++	2010		Deprecie
Intel CnC	2008		2018
Intel TBB	2006	Sep. 2017	Sep. 2018
Legion	2012	Mai 2018	Sep. 2018
OMPSs	2014	Juin 2017	Dec. 2017
OpenACC	2011		Nov. 2018
OpenMP	1997	2013 (4.0)	Nov. 2018
PaRSEC	2013	2015	2016
StarPU	2010	Aou. 2018	Sep. 2018
Taggre	N/A		
X10	2004	Juin 2016	Juin 2017
X-KAAPI	2010	Juin 2015	Jan. 2017

Table 3.3: Classification de supports exécutifs selon leur activité récente (le 9 Décembre 2018).

Hétérogénéité Si l'importance et l'intérêt pour un support exécutif de proposer une solution pour pouvoir utiliser des cartes accélératrices (GPGPUs ou Xeon Phis par exemple) ne sont pas approfondis dans le cadre de cette thèse, des efforts ont déjà été portés vers cet horizon et figurent dans les priorités HPC de *Code_Saturne*. Le support de l'hétérogénéité est donc un plus, d'autant plus si le modèle permet d'y accéder de manière simple et pérenne. On dénombre ici de nombreux candidats parmi lesquels nous retiendront Legion, OpenMP, PaRSEC et StarPU. De manière générale (et puisque ce n'est pas le cas pour la reconstruction des gradients), de telles architectures représentent un avantage évident lors de calculs particulièrement intensifs.

Type de communication *Code_Saturne* utilise déjà les bibliothèques OpenMP et MPI, mélangeant déjà ainsi deux types de communication, par mémoire partagée et par message. Ces choix nous pousseraient ainsi à ne pas privilégier ces deux modèles, tout du moins en ce qui concerne les communications par mémoire partagée puisque OpenMP à été historiquement choisi. En ce qui concerne les supports exécutifs de type PGAS, 3 supports se démarquent : X10, Legion et Chapel. Tous trois proposent un langage semblant traiter les problèmes de parallélisme avec élégance. L'aspect non structuré des maillages utilisés par nos études nous promet pourtant avec certitude une remise en question de l'utilisabilité de tels supports une fois sorti du cadre de l'élégance.

Choix du support exécutif L'accumulation de ces divers paramètres ainsi que les affinités du laboratoire Inria Bordeaux Sud-Ouest nous poussent à arrêter notre choix sur trois solutions, OpenMP (qui a le mérite d'être déjà largement présent dans la base de code existante), PaRSEC pour son approche du parallélisme par tâches particulièrement intéressante (et ayant largement fait ses preuves sur une ancienne collaboration EDF/INRIA [80]) et enfin StarPU.

4 | Contributions vers un paradigme de tâches

I Contribution : Transition pour la reconstruction de gradient par cellules

I.1 Présentation de la reconstruction sous forme de graphe de tâches

La reconstruction de gradient, lorsqu'elle est effectuée par itération sur l'ensemble des cellules plutôt que par face (revoir la Figure 2.2 en page 23 pour une comparaison des algorithmes possible pour la reconstruction), offre une possibilité de parallélisme assez élevée : les cellules peuvent être agrégées en bloc, et chaque bloc peut être considéré indépendamment de ses voisins locaux (au sein d'une même machine), car les dépendances de données ne sont alors qu'en lecture et aux frontières de chaque bloc. Une seule contrainte reste : la dépendance en lecture sur les voisins non locaux (positionné sur un autre nœud). Cette dépendance est prise en charge par l'échange d'halos avant tout calcul dans le code original (c.f. Figure 4.1), sans traitement fin quant à la possibilité d'effectuer une majorité des calculs indépendamment de cet échange, qui de fait, n'impacte les cellules qu'aux bords de chaque sous-domaine.

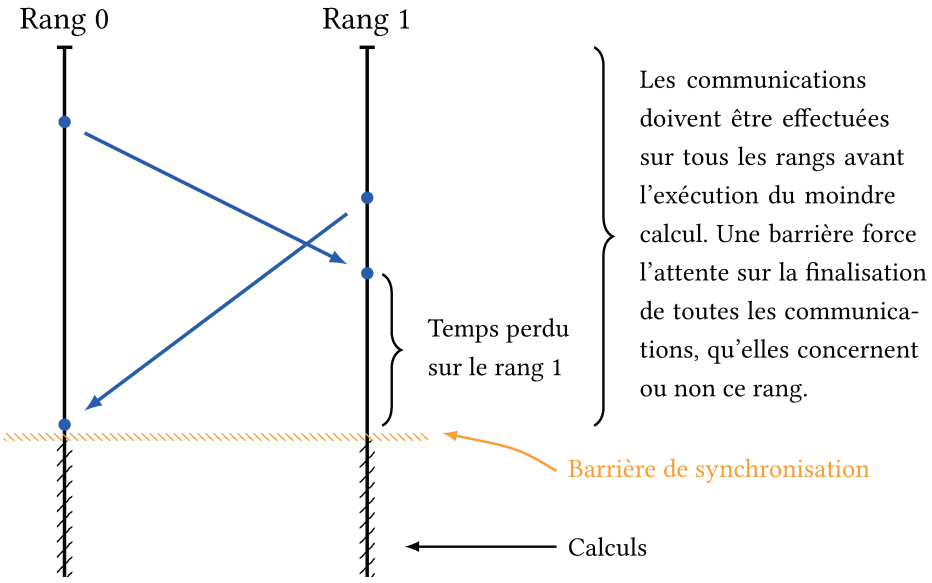


FIGURE 4.1: Scénario de calculs et communications dans le modèle BSP. Sans recouvrement des communications, les processus légers affiliés à chaque rangs seront assujettis aux mêmes contraintes de synchronisation.

On profite ici du paradigme de parallélisme par tâches pour introduire la notion de recouvrement des communications par les calculs dans *Code_Saturne*. Par ce biais, on peut évaluer à la fois l'implémentation d'un concept classique dans *Code_Saturne* en termes de gain de performance et de facilité du support exécutif à réaliser cette première étape.

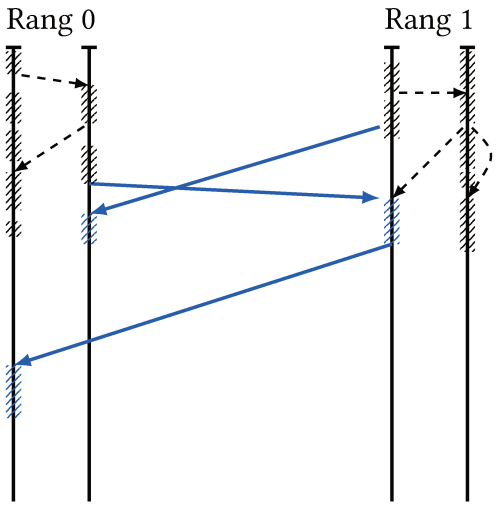
Le code actuel de *Code_Saturne* n'implémente pas de telles optimisations car cela représenterait une modification trop importune du code, nuisant ainsi profondément à la lisibilité de ce dernier.

D'un point de vue purement théorique et en raisonnant sur un modèle de tâches pure, on se retrouve alors avec un ensemble de tâches identiques exécutant la reconstruction de gradients chacune sur un bloc de cellules lui appartenant. Ces tâches ont toutes besoin d'accéder en lecture aux données de leurs tâches voisines, c'est-à-dire les tâches procédant des blocs de cellules voisins à leurs blocs. Ce besoin n'est néanmoins pas une véritable dépendance en lecture au sens propre du

terme. En effet, si la tâche A a besoin d'une donnée qui est possédée par la tâche B, cette dernière ne modifie en l'occurrence pas cette donnée, mais en est juste en possession. Concrètement, la reconstruction de gradient (sur un scalaire et pour une cellule donnée) utilise la valeur de ce scalaire sur l'ensemble des cellules voisines à cette cellule. Le scalaire en lui-même n'est pas modifié.

Recouvrement par tâches En utilisant un support exécutif idéal, on s'attend à pouvoir exprimer ce besoin de manière identique, que la donnée soit locale ou distante, de telle sorte que l'on puisse observer un flux de donnée entre chaque tâche. Ce flux est alors interprété en fonction de la localité des tâches A et B. Si elles sont sur le même nœud, aucun transfert de donnée n'est nécessaire. Dans le cas contraire, un échange de donnée doit avoir lieu avant l'exécution des deux tâches.

En pratique, cette démarche pose deux problèmes majeurs. D'une part le code doit être modifié de façon à stipuler de manière précise ce que chaque tâche possède en matière de donnée, et ce afin de confirmer le transfert de donnée d'une tâche à une autre uniquement lorsque cela est nécessaire. D'autre part en procédant ainsi, on multiplie le nombre d'échanges de données potentiels. Cela peut être bénéfique comme calamiteux en fonction du type de réseau et de la couche logicielle gérant les transferts de données (c.f. Figure 4.2, veut-on définir précisément ce que chaque tâche a besoin en matière de donnée au risque de découper en plusieurs parties un échange d'un voisin à un autre ? Cette question peut être rapidement résolue en fonction des propriétés réseau et de la quantité d'information à échanger : dans notre cas, on s'attend à ce que la latence soit à minimiser plutôt que la bande passante car dans notre cas c'est la quantité de messages plus que la quantité de donnée qui pose des problèmes de synchronisme).



Les calculs et les communications sont intégrés dans des ensembles cohérents appelés tâches. Les mouvements de données nécessaire à la réalisation des flux de données entre tâches sont effectuées par le support exécutif et sont transparents pour le développeur.

La granularité des tâches et leur distribution définissent la quantité et la nature des échanges effectués.

FIGURE 4.2: Scénario d'exécution d'un algorithme sous forme de tâches, les communications deviennent implicites et sont le résultats de dépendance en donnée d'une tâche vers une autre. En pointillés, les dépendances en données locales, en bleues celles nécessitant un échange inter nœud. La connaissance de la répartition des données pour chaque processus est nécessaire (c.f. graphe global).

Recouvrement statique Notre solution se positionne à mi-chemin entre les deux, de manière à profiter du recouvrement des communications grâce au calcul de tâches non dépendantes sur les données voisines tout en effectuant qu'un seul échange de donnée par couple de processus (c.f. Figure 4.3). Cette méthode facilite l'expression de nos tâches de calcul et s'associe plus aisément

à l'échange de halo, qui ne représente pas en soit une dépendance d'une tâche à une tâche, mais simplement la nécessité de recevoir une donnée (sans qu'aucune modification n'ait été effectuée dessus par une tâche). On utilise alors la connaissance que l'on a, à la fois des besoins de chaque tâche en matière de donnée en entrée et du statut des communications, pour permettre aux tâches qui n'ont pas besoin de l'échange des halos (effectué de manière standard) d'être exécutées au plus tôt.

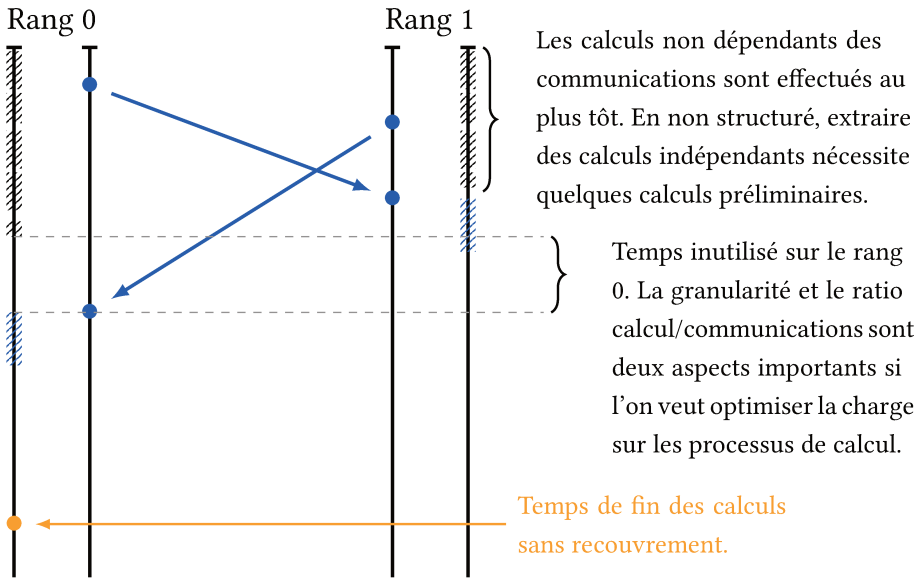


FIGURE 4.3: Exemple d'une exécution théorique d'un ensemble de tâche en fonction de leur dépendance ou non aux halos (respectivement en couleur bleue et noire) lorsque l'on peut exploiter cette information.

Afin de pouvoir procéder de la sorte, il nous est nécessaire de récupérer pour chaque cellule son voisinage afin de déterminer son niveau de localité. On dit alors qu'une cellule est « asociale » si l'ensemble de ses cellules voisines est inclu dans les cellules locales. C'est-à-dire que l'on a : $\forall\ n \in cell\_cells_i$ si $n < n\_local\_elts$ alors $\alpha_i = 1$ avec α_i l'asociabilité de la cellule i, et $cell\_cells_i$ l'ensemble des cellules voisines à i. On peut alors compresser cette information par blocs, de telle sorte que l'on sache pour un bloc de cellule donnée, si l'on doit attendre ou pas la réception des halos.

Dans un second temps, avec la connaissance de la structuration des halos dans *Code_Saturne*, on peut affiner notre connaissance en incluant pour chaque cellule, ou chaque bloc, la liste des rangs MPI dont il dépend, permettant ainsi de réduire pour les cas avec un fort nombre de voisins, le temps d'attente effectif de chaque bloc de cellules. Les figures 4.4 4.5 et 4.6 montrent pour le cas d'un maillage réparti sur 4 rangs MPI, la construction de l'index des cellules locales et non locales du rang 0. On peut en déduire alors avec précision les dépendances en halos de chaque bloc (c.f les figures citées, les dépendances y sont détaillées sur chacune des figures, les blocs ayant des dépendances sont de couleur différentes à ceux n'en ayant pas).

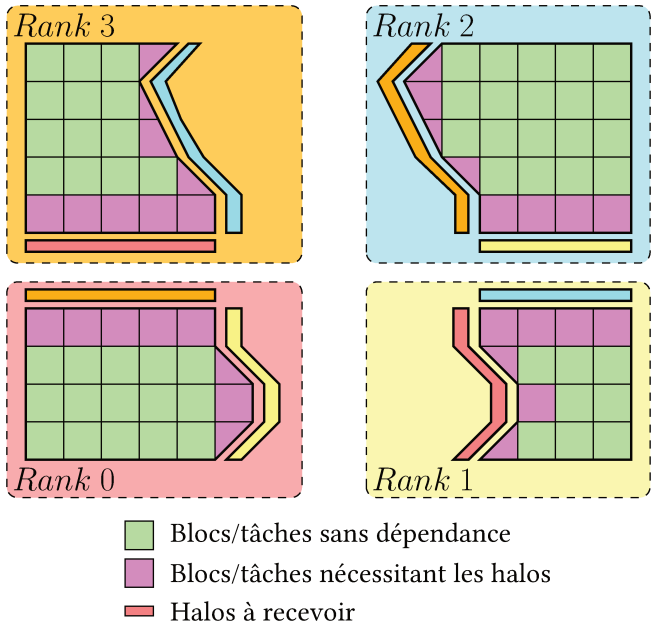


FIGURE 4.4: Vue d'un maillage partitionné pour 4 rangs MPI. On fait une correspondance entre tâche et bloc pour la reconstruction de façon à ce qu'une tâche de reconstruction corresponde à un bloc de cellule. On peut alors déterminer les dépendances entre tâches en fonction du voisinage de son bloc de cellule.

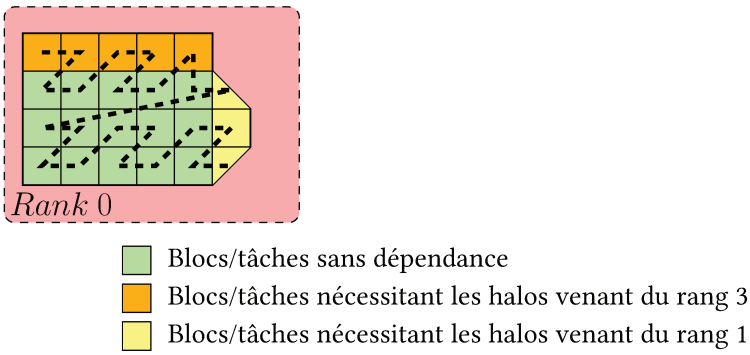


FIGURE 4.5: Spécification pour chaque bloc de cellules des halos nécessaire, de manière numérique et en accord avec les indexes des cellules hors locales (c.f. la figure 4.6). En traçant par dessus une SFC grossière, on visualise aisément l'aspect irrégulier des accès aux halos (et aux cellules en général). Une version raffinée de cette courbe (qui passerait de cellule en cellule) respecterait la même géométrie générale. On constate les allers retours constants entre partie purement locale et partie nécessitant les halos. Une transposition 1D de la numération qui en découle nous montrerait la disparité de la répartition des cellules en bord de maillage.

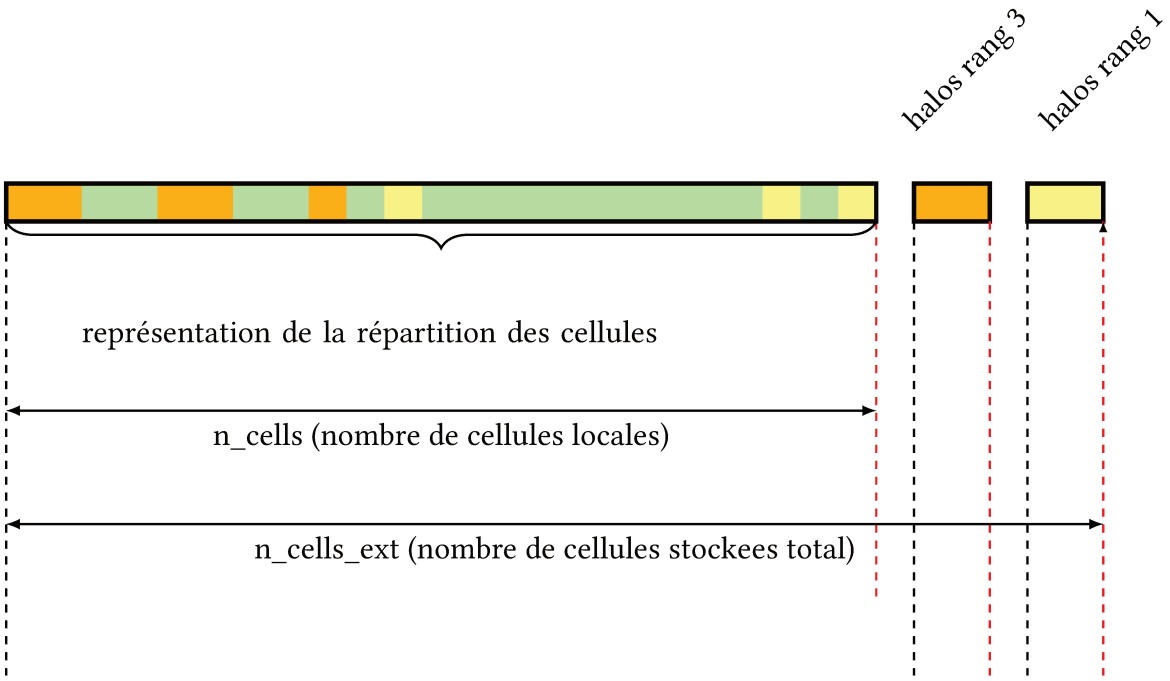


FIGURE 4.6: Vue de la localisation des cellules nécessitant les halos, en fonction des voisins. On observe une certaine disparité dans leur disposition du fait de la numérotation (c.f. Courbe de remplissage fractale en figure 4.5). La distance entre cellules locales et distantes lorsqu'elles sont en relation directe est maximale pour les cellules locales indexées en tout début. Si la quantité de données n_cell_ext ne rentre pas en cache, on va alors perdre beaucoup de temps en défauts de cache.

I.2 Implémentation avec PaRSEC

I.2.1 Définir un graphe global

Domaine d'exécution PaRSEC utilise le modèle de tâche paramétré. Ce sont donc les flux de données entre tâches qui permettent de déterminer les dépendances entre chaque tâche. Ces dépendances permettent alors à PaRSEC de maintenir un graphe global des tâches représentant notre algorithme. Comme vu précédemment, ce modèle de tâches oblige le développeur à devoir prédéterminer l'ensemble des variables définissant son graphe. On doit ainsi détailler avant compilation un ensemble de paramètres tels que les données en entrée et en sortie nécessaires à l'exécution de la tâche, mais aussi son domaine d'exécution, défini par exemple dans la Tâche 1 à la page 61 via les variables my_rank et n (lignes 1, 3 et 4). Ici ces deux paramètres définissent ainsi un set de tâches de taille $my\_rank \times n$. Ce set correspond à l'ensemble des tâches d'envoi de notre algorithme, tous processus confondus. On prendra note de leur déclaration sous la forme "0..n". Cette expression est importante car c'est elle qui définit la taille de notre set de tâches.

```
1  SEND(my_rank, n)
2
3  my_rank = 0 .. WORLD−1
4  n = 0 .. WORLD−1
5  n_idx = %{ return find_neighbor(halo, n); }
6  c = %{ return get_send_count(halo, n_idx); %}
7
8  : cs_halos(my_rank, n, MESH, PVAR)
9
10 READ HALO <− (n == my_rank) ? NULL
11              <− (n != my_rank && !c) ? NULL
12              <− (n != my_rank && c) ? NEW [ count = c ]
13              −> (n != my_rank && c) ? HALO RECV (n, my_rank) [ count = c ]
14
15 BODY
16   cs_real_t *h = (cs_real_t*) HALO;
17   if (n != my_rank)
18     // fill halo send buffer in h[]
19 END
```

Tâche 1: Envoi des halos via PaRSEC (format de représentation d'un graphe appelé JDF propre à PaRSEC). Les routines permettant de calculer la liste des voisins pour un rang donné ainsi que la quantité de donnée à recevoir pour le dit voisin ne sont pas détaillées dans un soucis de gain de place.

Distribution Puisque l'on définit de manière générale notre set de tâches, c'est à dire que chaque processus a la même connaissance de ce set, il est ensuite requis de fournir à PaRSEC les éléments nécessaire lui permettant de déterminer à quel processus chacune des tâches comprises dans ce set appartient. On parle de fonction de distribution des tâches (voir ligne 8, de la Tâche 1). Cette fonction de distribution doit aussi définir le segment des données sur lequel chacune des tâches sera amenée à travailler. Dans notre cas, nous répartissons chacune des tâches $SEND_{0..N,0..N}$ sur le processus de rang correspondant à my_rank de telle sorte que chaque processus se voit attribué une liste de tâche d'envoi de taille N c'est à dire de la taille de "MPI_Comm_size()". On peut d'ores et déjà noter ici

le manque de flexibilité d'une telle définition dans notre cas. En effet, chaque processus possède un nombre de voisin variable auquel correspond notre nombre de tâches d'envoi par processus. Puisque l'on est obligé de déterminer un domaine d'exécution statique pour notre tâche, il nous faut élargir ce set à l'ensemble des tâches possibles [1]. On se retrouve donc potentiellement avec un nombre de tâches surnuméraire par rapport à la réalité, puisque le nombre de processus total, lorsque ce dernier est grand, est largement supérieur au nombre de voisins moyens de chaque processus. Ce dérangement ne s'arrête pas là puisqu'il s'agira ensuite de devoir faire le tri pour chaque tâche de ses actions, et donc dans notre cas, d'évincer une majorité de tâches. On peut observer ce mécanisme en ligne 6 et 10-13 de la Tâche 1. On y récupère dans un premier temps en ligne 6 la quantité de données à échange du rang i à j pour la tâche $SEND_{i,j}$. On détermine ensuite la liste des flux d'entrées/sorties en fonction de i, j et c, notre quantité de donnée. On alloue de cette manière un tableau tampon à envoyer à n lorsque c existe (plus de détails dans la sous-section suivante).

Contrôle de dépendance Lorsque les flux de données ne suffisent pas à définir un graphe de tâches cohérent à notre algorithme, PaRSEC permet la déclaration de contraintes définissant l'ordre dans lequel les tâches doivent s'exécuter. Dans notre cas, cette fonctionnalité est largement mise à contribution pour la libération des dépendances des tâches de reconstruction ayant besoin des halos (c.f. ligne 12 de la Tâche $RECV$ 2 en page 63). La ligne 12 spécifie que chaque tâche $RECV_{i,0..n}$ du rang i libèrera une dépendance à chacune de ses tâches de reconstruction $GRADIENT\_COMPUTATION_{0..nblk}$.

I.2.2 Flux de données et expression des échanges de halos

En modèle de tâche paramétrée, la notion d'échange de donnée s'exprime par flux d'une tâche vers une autre. Dans notre cas, la reconstruction des halos n'a pas de prédécesseur : il n'y a aucune tâche en amont qui modifierait/produirait ces données. L'échange des halos ne peut donc pas se faire sous la forme d'un flux d'une tâche à une autre. De plus, les halos reçus d'un rang vers un autre ne contribuent pas à une tâche spécifique mais à un ensemble de tâches dont la taille varie en fonction de la granularité (pour rappel, on assimile des blocs de cellules à des tâches de reconstruction, or le maillage étant non structuré, on ne peut garantir un nombre fixe et/ou un index spécifique de tâches nécessitant tel ou tel halo). On ne peut donc régir l'échange des halos comme étant un flux d'une tâche de calcul envers une autre. A l'inverse il nous faut sur-représenter ces échanges en définissant des tâches d'envoi et de réception spécifiques (c.f. Tâche 1 ($SEND$) et 2 ($RECV$)).

Cette sur-représentation à elle seule ne suffit pas, puisqu'il faut enfin exprimer la dépendance de chaque tâche de réception vers les tâches de reconstruction correspondantes. Or le modèle paramétré de PaRSEC nous limite ici puisqu'il ne nous permet pas d'exprimer une liste de dépendance complexe [2] (chaque tâche de reconstruction peut dépendre de 0 à N halos, et leurs indexes peuvent ne pas se suivre). Nous recourrons donc ici à la déclaration de dépendance par contrôles tels qu'ils sont explicités ci-avant.

Enfin, l'échange des halos dans *Code_Saturne* est régie via une liste de rang dont l'ordre diffère pour chaque rang. On doit ainsi calculer en ligne 5 de la Tâche 2 l'indice correspondant au rang n de la tâche $RECV_{i,n}$ ainsi que la quantité de données à échanger correspondantes (via l'appel à du code C externe grâce aux balises d'échappements "%{" et "%}"). Si le code à exécuter n'est pas complexe et même si l'utilisation de PaRSEC en tant que support exécutif tendra à réduire de manière drastique

1. On peut constater assez rapidement que pour deux processus A et B, chacun ayant respectivement x et y voisins, qu'il devient compliqué avec le langage JDF utilisé par PaRSEC de spécifier dans une même définition de tâche que A aura x tâches d'envoi tandis que B en aura y. Si l'on définit ici le domaine d'exécution de nos tâches sur $[0 : N]$ par simplicité, il nous faudrait dans le meilleur des cas le définir sur $[0 : max\_N]$, max_N étant le nombre maximal de voisins constaté sur l'ensemble de nos rangs.

2. Ladite liste peut être variable mais doit être fonction des paramètres et exprimable de manière affine, critères auxquels ne correspondent pas les nôtres

```
1  RECV(my_rank, n)
2
3  my_rank = 0 .. WORLD−1
4  n = 0 .. WORLD−1
5  n_idx = %{ return find_neighbor(halo, n); }
6  c = %{ return get_send_count(halo, n_idx); %}
7
8  n_blk = %{ return cs_glob_n_blocks−1 %}
9
10 : cs_halos(my_rank, n, MESH, PVAR)
11
12 CTL ctl_recv −> ctl GRADIENT_COMPUTATION(0..nblk)
13
14 READ HALO <− (n == my_rank) ? NULL
15         <− (n != my_rank && !c) ? NULL
16         <− (n != my_rank && c) ? HALO SEND(my_rank, n) [ count = c ]
17         −> (n != my_rank && c) ? cs_halos(my_rank, n, MESH, PVAR) [ count = c ]
18
19 BODY
20   return PARSEC_HOOK_RETURN_DONE;
21 END
```

Tâche 2: Réception des halos via PaRSEC (JDF)

le nombre de processus (puisque les recommandations sont de n'avoir qu'une instance de PaRSEC par nœud), il reste que ce code devra être exécuté pour chacune des tâches (puisqu'il fait partie des paramètres de la tâche) ce qui ne peut qu'entraîner un surcoût.

I.2.3 Raffiner la gestion des dépendances

La tâche 3 décrit la reconstruction des gradients de manière grossière (dans le sens où elle ne permet aucune finesse de contrôle sur l'ordonnancement des tâches ayant besoin des halos, c.f ligne 17, on bloque l'ordonnancement d'une telle tâche sur la réception de l'ensemble des halos). Afin d'exprimer la dépendance entre tâches de reconstruction et tâches de réception de manière un peu plus fine, on peut prendre en compte la pleine connaissance pour chaque bloc, de ses besoins et non pas de se limiter à la condition unique de réception de tous les halos définie en amont. On fait alors évoluer nos tâches $RECV$ et $GRADIENT\_COMPUTATION$ pour refléter ces dépendances plus fines (visible respectivement en tâche 4 et 5). Il nous faut pour cela maintenir une liste de dépendances envers les halos distants pour chaque bloc (procédure détaillée en page 57). Il nous faut ensuite contrôler la réception des halos pour chacun des blocs (c.f. Tâche 4 $BLOCK\_RECV\_CTL_{my\_rank,b,n}$). Cette tâche se charge de libérer une dépendance sur la tâche $GRADIENT\_COMPUTATION_b$ correspondante (c.f. la nouvelle définition de $GRADIENT\_COMPUTATION_b$ en Tâche 5). Elle prend en lecture la sortie de la tâche $RECV_{my\_rank,n}$ correspondante lorsqu'il convient d'attendre une réception mais est exécutée directement si ce n'est pas le cas. On peut vérifier en lignes 11-12 de la Tâche 5 le nombre de contrôles nécessaires à l'ordonnancement de cette tâche (égal au nombre total de rangs de la simulation).

En considérant les dépendances de chacun des halos séparément, et ce pour chaque bloc de cellule, on introduit ainsi un grand nombre de tâches de contrôle $BLOCK\_RECV\_CTL$ supplémentaires, dont potentiellement une majorité qui sont vides ($n \times b$ tâches de contrôle par rangs).

```
1  GRADIENT_COMPUTATION(blk_id)
2
3  blk_id = 0 .. %{ return cs_glob_n_blocks − 1; %}
4  end_shift = 1
5  start = %{ return (blk_id ∗ GRADIENT_BLK_SIZE); %}
6  stop = %{ int _end = (start + GRADIENT_BLK_SIZE) < MESH−>n_cells
7                                        ? start + GRADIENT_BLK_SIZE
8                                        : MESH−>n_cells;
9                          return _end; %}
10
11 need_halos = %{ return MESH−>blocked_halo_deps[blk_id]; %}
12
13 : taskdist( blk_id )
14
15 READ A <− NULL
16
17 CTL ctl <− ( need_halos ) ? ctl_recv RECV(RANK, 0..WORLD−1)
18
19 BODY
20    _lsq_psc_blk_scalar_grad(start, stop, INC, COEFBP, COEFAP, EXTRAP, GRAD,
21                                    PVAR, FVQ, MESH, MESH_ADJ, GRADIENT_BLK_SIZE);
22 END
```

Tâche 3: Reconstruction des gradients via PaRSEC (JDF)

L'accumulation créée par l'ensemble des définitions de tâches (permettant l'obtention d'un graphe de tâches global) rend l'utilisation de PaRSEC pour *Code_Saturne* assez peu élégante.

I.2.4 Interfacer PaRSEC avec *Code_Saturne*

Afin d'utiliser PaRSEC, il ne suffit pas d'écrire des tâches. Les fonctions de distribution des tâches ainsi que d'attribution des données à chaque tâche doivent être spécifiées par l'utilisateur. Si dans notre cas ces dernières se limitent à une seule version, elles doivent être en réalité multiples en fonction des divers types de données ainsi que de leur structure interne. De manière générale, la réflexion apposée par PaRSEC est de partir de l'idée que les données ne sont pas partitionnées avant son utilisation. Dans ce cadre, spécifier à qui appartient telle ou telle partie d'une matrice prend tout son sens, puisque PaRSEC prendra en charge la répartition des données sur l'ensemble des processus. Hors dans notre cas, la répartition des données est déjà faite. Il nous faudra cependant spécifier pour l'ensemble de nos tâches locales, quelle partie de la mémoire elles se voient attribuées. Cette définition est notamment importante pour l'échange des halos puisque ce dernier est exprimé sous la forme de tâches. Ce mécanisme est présenté en annexe en pages 101-103.

```
1  BLOCK_RECV_CTL(my_rank, b, n)
2
3  my_rank = 0 .. WORLD−1
4  n = 0 .. WORLD−1
5  b = 0 .. %{ return cs_glob_n_blocks − 1; %}
6  n_idx = %{ .. %}
7  c = %{ .. %}
8  need = %{ return block_need_halos_from_n(b, n_idx); %}
9
10 : cs_halos(my_rank, n, MESH, PVAR)
11
12 CTL ctl_n_for_blk −> ctl GRADIENT_COMPUTATION(b)
13
14 READ HALO <− (n == my_rank) ? NULL
15       <− (n != my_rank && !c) ? NULL
16       <− (n != my_rank && c && !need) ? NULL
17       <− (n != my_rank && c && need) ? HALO RECV(my_rank, n) [count = c]
18
19 BODY
20   return PARSEC_HOOK_RETURN_DONE;
21 END
```

Tâche 4: Réception des halos via PaRSEC (JDF)

```
1  GRADIENT_COMPUTATION(blk_id)
2
3  blk_id = 0 .. %{ return cs_glob_n_blocks − 1; %}
4       {...}
5  need_halos = %{ return MESH−>blocked_halo_deps[blk_id]; %}
6
7  : taskdist( blk_id )
8
9  READ A <− NULL
10
11 CTL ctl <− ( need_halos ) ? ctl_n_for_blk
12                  BLOCK_RECV_CTL(RANK, blk_id, 0..halo−>n_c_domains−1)
13
14 BODY
15  {...}
16 END
```

Tâche 5: Reconstruction des gradients via PaRSEC (JDF)

I.3 Implémentation avec StarPU

On détaille ici les possibilités de StarPU et leur adéquation au développement CFD en maillage non structuré. Dans le cas de la reconstruction des gradients par les cellules, on donne une grande importance à la gestion des échanges des données (dans notre cas des halos) car c'est cette notion qui a le plus d'impact, non seulement dans notre cas pour la reconstruction des gradients mais aussi de manière plus générale sur le design d'un algorithme à paradigme de tâches pour *Code_Saturne*. On attend en effet lors de l'expression d'un algorithme sous forme de graphe de tâches à une certaine abstraction de l'échange des données. Cet échange est généralement transparent (car pris en charge par le support exécutif) lorsqu'il est le signe d'une dépendance d'une tâche vers une autre (une tâche ayant besoin des données d'une tâche précédente notamment). On verra dans notre cas, où cet échange de donnée est la seule contrainte et donc seule dépendance à exprimer, que cette logique peut poser certaines contraintes qui rendent son utilisation complexe voire non souhaitable. On présente donc diverses implémentations en graphe local ou global, ainsi que leurs avantages et inconvénients.

Graphe global : échanges de données implicites StarPU permet d'exprimer les échanges de manière implicite via la description de structures de données. Ces dernières sont attachées à nos tâches, et l'on spécifie la nature de l'action de notre tâche sur chacune d'entre elles, que ce soit en lecture ou en écriture. Cette solution a l'intérêt de pouvoir proposer l'approche la plus claire en terme de design. Elle demande en pratique un certain nombre d'opérations supplémentaires afin d'avoir les connaissances nécessaires sur chaque rang. Notamment :

— pour chaque tâche, la liste des structures de données distantes nécessaires,
— pour chaque rang voisin, les tâches qui dépendent de nos cellules

La première condition est remplie grâce à l'appui des échanges de halos tels qu'ils sont définis dans *Code_Saturne* de manière initiale (c.f. Codelet 1). Chaque rang déclare ainsi de manière locale et préalable à toute reconstruction de gradients, l'ensemble des halos que l'on possède ainsi que l'ensemble des halos qui nous sont nécessaires (lignes 14 et 24). Cette déclaration doit ensuite être propagée sur l'ensemble des instances de StarPU via MPI (ligne 39) de telle sorte qu'une structure déclarée comme étant non locale mais nécessaire sur le rang x soit déclarée avec le tag correspondant sur le rang y. Ce code étant exécuté en parallèle par MPI, il faut prévoir l'ensemble des déclarations pour chaque rang en un ensemble homogène. On peut remarquer en ligne 14 l'emplacement de chaque halo positionné sur le tableau tampon de construction des halos de *Code_Saturne* tandis que la ligne 29 s'assure que les structures distantes verront bien leurs données stockées à l'endroit initial prévu dans *Code_Saturne* (ce qui nous évite une recopie supplémentaire).

La deuxième condition est néanmoins plus gênante dans notre cas car elle ne correspond pas à une contrainte de notre graphe de tâches réel. Premièrement, d'un point de vue local la grande majorité des tâches ne dépendent pas des halos et deuxièmement, la dépendance exprimée est en théorie celle d'une tâche vers des données que l'on doit recevoir une seule fois, et pas d'une communication accrue entre les tâches. De ce fait la connaissance pour chaque rang, de l'ensemble des tâches de ses voisins nécessitant ses données n'a que peu d'utilité pour l'ordonnancement des tâches sur ce rang, mis à part pour savoir qu'il doit effectuer un seul échange de donnée. Stocker un tel ensemble de tâches vides représente néanmoins un surcoût en mémoire non négligeable au fur et à mesure que le nombre de tâches par rang augmente. Pour pallier ce problème, on fait appel à la déclaration d'une fausse tâche servant simplement à spécifier la dépendance d'une tâche vers des données distantes. On réduit ainsi les besoins en déclaration de tâches globales au minimum tout en garantissant la réalisation des échanges de halos. Le Codelet 2 détaille cette partie.

On doit ensuite construire un graphe qui prend en compte la variabilité du nombre de dépendances pour chacune des tâches de reconstruction des gradients. Cette gestion n'est pas innée puisque la tâche est préférablement définie de manière statique. Or dans notre cas, non seulement

```
int x = my_rank;
for (y = 0; y < cs_glob_n_ranks; ++y) {
  if (y != x)
  {
    for (rank_id = 0; rank_id < halo->n_c_domains; rank_id++) {
      if (halo->c_domain_rank[rank_id] == y) {

        start = halo->send_index[2*rank_id];
        length = halo->send_index[2*rank_id + end_shift]
                 - halo->send_index[2*rank_id];

        // we own data needed by rank y
        if (length > 0)
          starpu_vector_data_register(&cs_halo_handles[x][y],
                                      STARPU_MAIN_RAM,
                                      (uintptr_t)&build_buffer[start],
                                      length, sizeof(CS_MPI_REAL));

        start = halo->index[2*rank_id];
        length = halo->index[2*rank_id + end_shift] - halo->index[2*rank_id];

        // we don't own that data but we will need it
        if (length > 0)
        {
          starpu_vector_data_register(&cs_halo_handles[y][x], -1,
                                      (uintptr_t)&var[halo->n_local_elts + start],
                                      length, sizeof(CS_MPI_REAL));

          // force data to be stored where code_saturne needs it
          starpu_vector_ptr_register(cs_halo_handles[y][x],
                                     STARPU_MAIN_RAM,
                                     (uintptr_t)&var[halo->n_local_elts + start],
                                     NULL, NULL);
        }
}}}}

for (x = 0; x < cs_glob_n_ranks; ++x)
  for (y = 0; y < cs_glob_n_ranks; ++y)
    if (cs_halo_handles[x][y])
      starpu_mpi_data_register(cs_halo_handles[x][y], x*cs_glob_n_ranks+y, x);
```

Codelet 1: Déclaration globale des structures de données pour StarPU

```
1  for (i = 0; i < halo->n_c_domains; ++i)
2    // set arguments parameters for buffer building task
3    start = halo->send_index[2*i];
4    length = halo->send_index[2*i + 1] - halo->send_index[2*i];
5
6    // tell starpu this task will produce data for neighboring ranks
7    bbuff_task[d] = starpu_mpi_task_build(cs_glob_mpi_comm, &cl_build_buffer,
8                                          STARPU_RW,
9                                          cs_halo_handles[my_rank][other_rank],
10                                         STARPU_EXECUTE_ON_NODE, my_rank, 0);
11
12   // tell starpu that another rank will need the handle its tasks
13   fake_task[d] = starpu_mpi_task_build(cs_glob_mpi_comm, &cl_compute,
14                                        STARPU_R,
15                                        cs_halo_handles[my_rank][other_rank],
16                                        STARPU_EXECUTE_ON_NODE, other_rank, 0);
```

Codelet 2: Déclaration globale de la construction des buffers d'envoi et des besoins en réception via StarPU

la tâche à un nombre de dépendances qui ne peut être défini avant l'exécution du code, mais il est variable pour chaque instance de cette tâche (c.f. Codelet 3). On remarque que quelque soit sa dépendance aux halos, celle-ci n'est intégrée qu'à l'échelle locale du graphe, sa dépendance étant déjà gérée par la déclaration d'une fausse tâche (Codelet 2).

Il nous reste alors plus qu'à soumettre nos tâches (c.f. Codelet 4). Dans ce cas, on considère deux types de tâches différentes : des tâches de construction des tampons d'envoi dont on ne peut s'abstraire, et des tâches de reconstruction des gradients sur un ensemble de cellules données.

On peut d'ores et déjà observer un certain manque d'intérêt dans notre cas de l'utilisation d'un graphe global quant à l'expression par tâche de la reconstruction de gradients sur les cellules. Cet intérêt réduit est notamment dû à la nature de notre algorithme qui n'exhibe un graphe que très simpliste puisque les tâches de reconstruction à proprement parler ne contribuent pas à d'autres tâches par la suite. Il est tout à fait envisageable d'établir ou de trouver une autre partie de *Code_Saturne* sur laquelle la notion de graphe global aurait un intérêt supérieur. Cette solution permet néanmoins à l'exception de la déclaration d'une tâche factice d'exprimer la reconstruction des gradients sans expression des échanges de données. Cependant, cette dépendance étant restreinte aux échanges purement inter-nodaux et non pas de tâche à tâche, le cas de la reconstruction des gradients peut bénéficier d'une expression explicite des échanges en addition d'un graphe local.

Graphe local : échanges de données explicites

StarPU-MPI et dépendance de tâche explicite D'un autre côté, garder l'explicitation des échanges permet une transition peut être plus facile d'un point de vue itératif de *Code_Saturne* vers un paradigme de tâches. Dans ce cadre, StarPU offre une évolution rapide entre un code MPI et sa version StarPU et ce à l'aide de directives sous la forme starpu_mpi_isend[...] qui remplacent tout appel MPI. Il faut cependant ajouter à cela la nécessité de création des structures de données descriptives de StarPU à l'image de la version avec graphe global. Si l'on s'en passerait volontiers dans cette version, puisque la similarité entre l'usage de StarPU et MPI est grande, cette obligation peut être vue comme un premier pas vers une transition à un graphe de tâches global. Elle permet néanmoins de la même manière de pouvoir gérer localement la dépendance vers une donnée distante

```
for (b = 0; b < n_blocks; ++b)
  // set task arguments {...}
  // count number of handles needed for this task {...} in bn_handles
  // dynamically building of handles dependency list for each block
  int c = 0;
  for (i = 0; i < halo->n_c_domains && bn_handles; ++i) {
    if (halo->c_domain_rank[i] != local_rank) {

      descrs[b][c].handle = cs_glob_halo_starpu_handle[c].recv_handle;
      descrs[b][c].mode = STARPU_R;
      c++;
  }}

  compute_block_tasks[b] = starpu_task_build( &cl_compute,
                                  STARPU_DATA_MODE_ARRAY, descrs[b], bn_handles,
                                  STARPU_EXECUTE_ON_NODE, my_rank, 0);

  // keep this task for resubmission
  compute_block_tasks[b]->destroy = 0;
  compute_block_tasks[b]->cl_arg_free = 0;
```

Codelet 3: Construction de dépendances globales en nombre variable pour chaque bloc de cellule avec StarPU.

```
for (i = 0; i < cs_glob_n_halos; ++i)
  starpu_task_submit(&bbuff_task[i])

for (i = 0; i < n_blocks; ++i)
  starpu_task_submit(&compute_block_tasks[i]);
```

Codelet 4: Processus de soumission des tâches dans le cas d'un graphe global.

```
1   for (rank_id = 0; rank_id < halo->n_c_domains; rank_id++)
2
3     start = halo->index[2*rank_id];
4     length = halo->index[2*rank_id + end_shift] - halo->index[2*rank_id];
5
6     if (halo->c_domain_rank[rank_id] != local_rank)
7       if (length > 0)
8         #if STARPU
9         struct cs_halo_handle *handle = &cs_glob_halo_starpu_handle[n];
10        starpu_mpi_irecv_detached(handle->recv_handle, handle->other_rank,
11                                  TAG(handle->other_rank, local_rank),
12                                  cs_glob_mpi_comm, recv_done,
13                                  &cs_glob_halo_status[n]);
14        #else
15        MPI_Irecv(var + halo->n_local_elts + start, length,
16                  CS_MPI_REAL, halo->c_domain_rank[rank_id],
17                  halo->c_domain_rank[rank_id], cs_glob_mpi_comm,
18                  &(_cs_glob_halo_request[request_count++]));
19        #endif
20
21  // [...] building buffer
22  // [...] similar pattern for sending buffer
```

Codelet 5: Comparaison côte à côte de l'échange des halos entre la version StarPU-MPI et MPI dans *Code_Saturne*

ce que l'on verra dans le sous paragraphe suivant. On détaille ici la première solution développée via StarPU qui ne fait pas appel à la déclaration dynamique des dépendances sur flux de données (tel qu'on a pu les voir dans le Codelet 3). On peut donc trouver un cheminement complet de transition pas à pas d'un code MPI + X vers StarPU qui exprime la capacité de StarPU à faciliter le déploiement itératif sur un code de l'envergure de *Code_Saturne*.

En premier lieu cette solution permet une transition quasi immédiate pour l'échange des halos dans *Code_Saturne*, à l'instar d'autres versions détaillées ci-après où l'on intègre MPI en tant que tel dans notre modèle. On peut ici constater une divergence dans l'expression de l'échange des halos très faible (c.f. Codelet 5).

L'asynchronisme intégré doit néanmoins être explicitement géré ensuite. En effet, nos tâches de reconstruction dépendant des halos doivent ici être déclenchées au bon moment, c'est à dire à la fin de la réception du ou des halos correspondants. Dans notre cas, on a pour cela mis à contribution le rappel de fonction effectué par les routines "starpu_mpi_irecv_detached" et "...isend..." afin de notifier nos tâches de calculs (c.f. Codelet 6). On se charge par ce biais de soumettre une tâche vide par réception de telle sorte que les tâches de reconstruction qui en dépendraient voient leur dépendance se libérer. Si cette approche ne semble pas être des plus directe, elle est relativement logique si l'on se place du point de vue d'une modification minimale du code d'origine. Elle a notamment le bénéfice pour le développeur de ne pas avoir à se soucier de l'ordre de soumission des tâches [3]. Le Codelet 7 présente de manière similaire au Codelet 3 l'expression explicite des dépendances entre tâches (ici

3. Lorsque l'on soumet des tâches dont les dépendances sont gérées de manière implicite, c'est à dire sur le flux de données qu'elles partagent, c'est l'ordre de soumission qui établit leur ordonnancement futur. Ainsi si deux tâches A et B accèdent toutes les deux à une donnée λ, ces tâches seront exécutées dans l'ordre dans lesquelles elles auront été soumises. Cette manière de gérer les dépendances simplifie largement leur expression de manière générale.

```
1  static void
2  recv_done(void *arg)
3  {
4    int *p = (int*) arg;
5    starpu_task_submit(&recv_control_tasks[*p]);
6  }
```

Codelet 6: Rappel de fonction pour chaque starpu_mpi_irecv_detached arrivé à terme. La tâche dite de contrôle ainsi soumise et vide libérera directement les tâches en attente de la réception correspondante.

```
1  for (rank_id = 0; rank_id < halo->n_c_domains; ++rank_id)
2    length = halo->index[2*rank_id + end_shift] - halo->index[2*rank_id];
3
4    if (halo->c_domain_rank[rank_id] != local_rank)
5      if (length > 0)
6        if (m->blocked_halo_deps_per_halo[b][rank_id])
7          starpu_task_declare_deps_array(&compute_block_tasks[b], 1,
8                                         &recv_control_tasks[c]);
```

Codelet 7: Construction des dépendances explicites entre tâches de contrôles et tâches de reconstruction nécessitant les halos.

l'ordre de soumission n'influe plus sur l'ordre d'ordonnancement des tâches puisque la dépendance est exprimée sur les tâches et non pas sur les données qu'elles manipulent).

StarPU-MPI et dépendance de tâche implicite Afin de simplifier le code, on peut faire appel aux notions de dépendances sur flux de données précédemment décrites et s'abstraire des complexités de la solution précédente. On retrouve alors une déclaration des dépendances similaire au Codelet 3 en addition à une gestion des échanges explicites tels que décrits dans la version StarPU du Codelet 5. En ce qui concerne l'adaptation de la reconstruction des gradients dans *Code_Saturne* sous forme de tâche, c'est cette solution qui se prête le plus à la nature du problème. Du fait de sa représentativité on peut néanmoins généraliser cette conclusion à l'ensemble de *Code_Saturne* si le critère de sélection retenu est la minimisation des modifications nécessaires à apporter pour la transition du code.

StarPU + MPI Pour des raisons de performances et/ou par simplicité on peut être amené à considérer l'utilisation de StarPU en le cloisonnant à la gestion de tâches locales et avec une implémentation "à la" MPI + X. En l'occurrence puisque notre graphe de tâches est ici très simple (notre seule dépendance ici est une dépendance de certaines tâches de reconstruction sur la réception des halos distants) l'utilisation des structures de données propres à StarPU est discutable. On peut ainsi dans notre cas gérer nos communications MPI de manière non bloquante et instaurer dans le processus MPI un point de contrôle sur chacune de nos requêtes. Lorsqu'une requête est répondue, on libère les tâches de reconstruction dépendant des halos via la soumission d'une tâche factice (c.f. Codelet 8). Les tâches de reconstruction sont alors déclarées de manière similaire au Codelet 3.

StarPU + MPI et échanges bloquants Encapsuler des échanges de données à l'intérieur de tâches peut sembler viable lorsque le problème à résoudre est similaire à un stencil tel que la résolu-

```
1  for (b = 0; b < n_blocks; ++b)
2        starpu_task_submit(&compute_block_tasks[b]);
3
4  cs_halo_sync_var_noblocking(halo, halo_type, pvar);
5  // {...}
6
7  while(ctl_task_submitted < cs_glob_n_halos)
8  {
9    for (i = 0; i < cs_glob_n_halos; ++i)
10   {
11     MPI_Test( &(_cs_glob_halo_request[i]), &flag, &status);
12
13     if(flag && !ctl_task_submitted_flags[i])
14     {
15       starpu_task_submit(&recv_control_tasks[i][0]);
16       ctl_task_submitted_flags[i] = 1;
17       ctl_task_submitted += 1;
18  }}}
19
20 starpu_task_wait_for_all();
```

Codelet 8: Code principal de la reconstruction des gradients avec appels MPI hors-StarPU et recouvrement des communications.

tion de l'équation de Poisson sur une grille 2D cartésienne. Dans ce cas, avoir un nombre fixe et/ou réduit de voisins pour chaque domaine est une métrique très importante. En effet, avoir quelques échanges bloquants sous forme de tâche n'est pas problématique si :

- leur nombre est largement inférieur au nombre de tâches par nœud pouvant être effectuées simultanément
- le pattern de communication est d'une forme simple, permettant ainsi de produire un schéma de communication qui nous garantit l'impossibilité de se retrouver dans une situation d'interblocage.

Dans notre cas, le nombre de voisins par domaine est non seulement variable, mais peut atteindre une soixantaine de voisins dans des cas conséquents mais néanmoins fréquents. Il est assez facile d'imaginer un cas dans cette configuration où plusieurs voisins puissent être bloqués dans l'attente de la résolution d'une communication. S'il est facile d'établir une version étalon sur peu de nœuds et dans laquelle une telle implémentation nous permettrait de gagner en performance sur celles détaillées ci-avant, sa viabilité dans un code de performance est largement mise en doute.

Tâches rémanentes La reconstruction de gradient est une étape qui sera répétée un grand nombre de fois au cours de la simulation. En partant du principe que le maillage n'évolue pas au cours du temps, on peut tirer profit de l'établissement du graphe de tâches de la première fois pour le garder en mémoire. En effet, on sait ainsi que le nombre de tâches de reconstruction sera constant, puisque les calculs effectués se font sur les cellules et faces du maillage. De plus, mis à part la variable physique mise à jour durant la reconstruction qui évolue, l'échange des halos garde la même structure : on échange le même nombre de données avec les mêmes voisins. On peut donc garder les mêmes dépendances sur notre graphe de tâches. Il faudra alors simplement mettre à jour les structures de gestion des données de StarPU pour que les échanges s'effectuent sur le bon tableau. On va ainsi, d'une part limiter à une fois par simulation la déclaration de notre graphe, et espérer un temps d'ini-

tialisation très restreint. La quantité de mémoire supplémentaire utilisée pour le stockage d'un tel graphe peut être légitime d'une part quant à sa réutilisation fréquente mais aussi par sa présence une fois par nœud seulement (tout en sachant qu'en passant à un seul processus MPI par nœud au lieu de un par cœur, on peut réduire d'autant la mémoire utilisée, selon si les bibliothèques ont été compilées en statique ou en dynamique).

StarPU permet une telle stratégie via l'utilisation des routines "starpu_task_build" et le paramètre "task->destroy=0" tels qu'on peut les retrouver en ligne 14 et 19 du Codelet 3 en page 69. On pourra ainsi soumettre autant de fois cette tâche que l'on le souhaite. La mise à jour des données sur lesquelles s'effectueront le graphe devra être établie de la même façon qu'à la première instanciation via les routines de type "starpu_<type>_data_register".

Placement des processus légers et support NUMA Nous savons déjà que *Code_Saturne* est largement parallélisé via MPI+OpenMP avec le constat qu'il est assez difficile de mettre à profit la localité des cœurs pour gagner en performance, c'est à dire que les performances du code se dégradent au fur et à mesure que l'on utilise des processus légers à la place de processus MPI. Le profil type d'une exécution optimale de *Code_Saturne* pour une machine de 24 cœurs non hyperthreadés étant de 12 ou 24 processus MPI (selon les cas) avec l'addition de 2 processus OpenMP par processus MPI, on est très loin des recommandations des supports exécutifs tels que PaRSEC ou StarPU qui préconisent une instance de support exécutif par nœud (ce qui pour MPI+OpenMP reviendrait ici à 1 processus MPI $\times$ 24 processus OpenMP). Le support NUMA de StarPU n'étant pas encore proposé par défaut, il nous paraît important d'évaluer les performances de StarPU dans un contexte similaire au nôtre. Puisque chaque instance de StarPU n'a pas conscience du placement des autres, si tant est qu'ils soient sur le même nœud, il s'avère nécessaire d'effectuer un placement efficient à la fois des processus MPI mais aussi des processus de chaque instance de StarPU. La figure 4.7 montre notre stratégie de placement.

On vérifie pour cela dans un premier temps le nombre de processus MPI locaux (sur le même nœud) ainsi que le nombre de cœurs sur ce nœud. On force alors chacun de ces processus de façon à ce qu'ils soient « équidistants » sur le nœud. Reste alors à l'utilisateur de fournir un couple [#MPI : #STARPU/MPI] cohérent de telle sorte à ce que le nœud soit utilisé à son maximum. On place alors automatiquement le processus de communication interne à StarPU sur le même cœur que son processus MPI maître, les communications étant dans notre cas, soit entièrement faites via MPI, soit uniquement appelées par StarPU (lors de la reconstruction des gradients, seule étape de la simulation où StarPU est actif, c.f. "starpu_pause" et "starpu_resume").

I.3.1 Gestion de la périodicité du maillage

En cas de périodicité dans notre maillage, une partie de nos valeurs locales doivent être recopiées en même temps que l'échange des halos. Cette action n'est faite qu'une fois par rang MPI. Dans le cas des versions par tâches, cette opération doit alors être effectuée avant les tâches de reconstruction qui agiraient sur cette partie du maillage. La nature indexée des structures n'impose ici aucune contrainte car la recopie se fait de manière symétrique. On pourrait donc considérer deux solutions, l'une où la recopie se fait sous forme de tâche, auquel cas il faudra exprimer une dépendance explicite entre les tâches correspondantes, ou l'exprimer sous forme de dépendance de flux entre deux tâches. Dans ce dernier cas il faudrait néanmoins que les données de périodicité soient le résultat d'une tâche. Dans la mesure où ce n'est pas notre cas (puisque le graphe de tâche commence par l'échange des halos et n'est pas branché à un graphe précédent), aucune des deux solutions ne semble meilleure que l'autre.

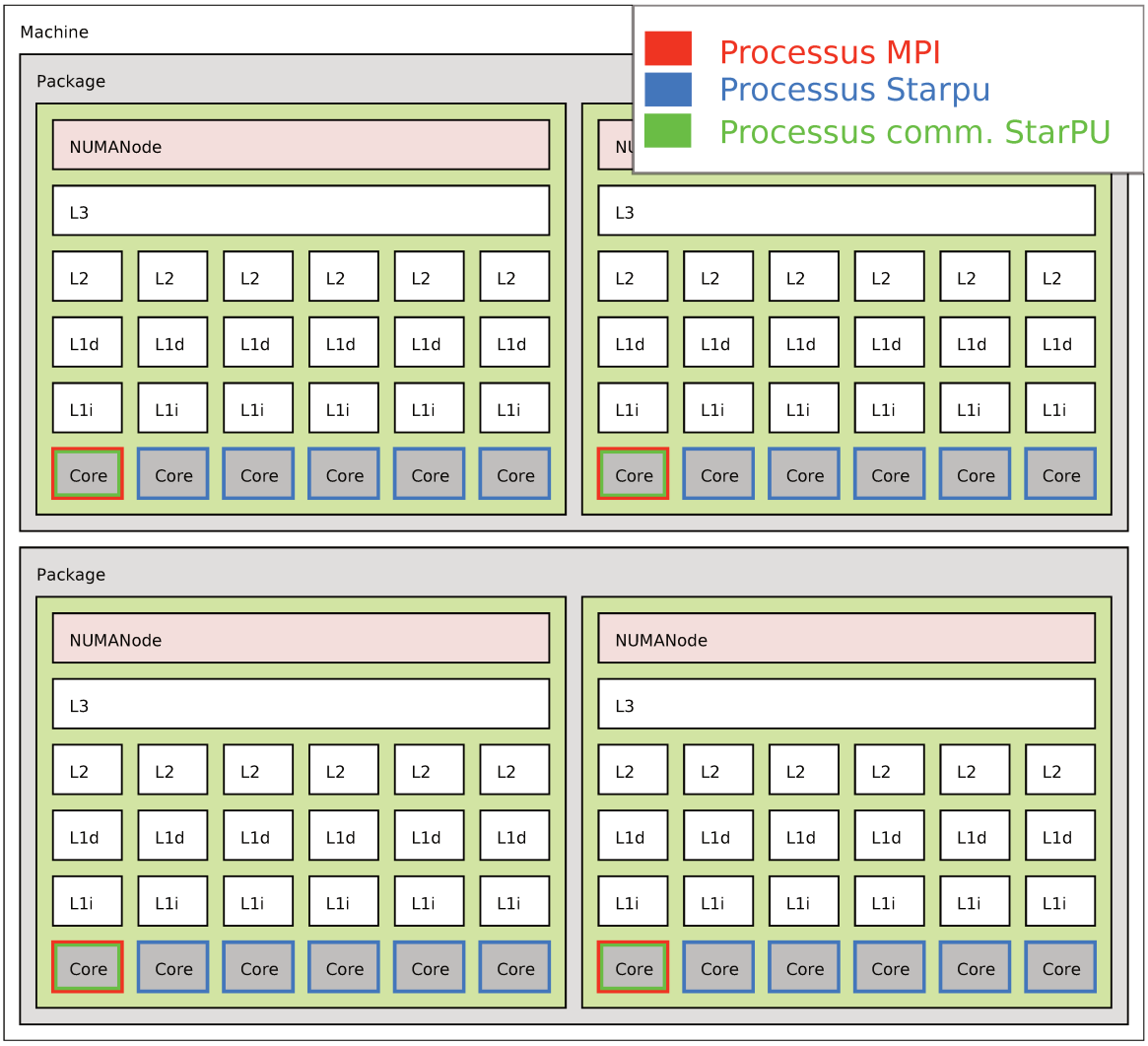


FIGURE 4.7: Exemple du placement des 4 processus MPI + StarPU sur un nœud Miriel (4 nœuds NUMA de 6 cœurs chacun, Plafrim).

II Contribution : Transition pour la reconstruction de gradient par faces

On a déjà vu dans le chapitre précédent que la parallélisation de la reconstruction de gradient par face est plus complexe lorsque l'on souhaite utiliser une bibliothèque de processus légers. On rencontre alors des problématiques de partage de données inter-processus ainsi que le risque de faux partage. Les mécanismes mis en place dans *Code_Saturne* actuellement (voir figure 2.4 page 25) permettent d'adresser cette problématique avec plus ou moins de réussite. Notamment le découpage proposé se fait en plusieurs étapes, chacune n'ayant pas le même équilibre (on opère n passes de n processus légers, avec une quantité totale de cellules/faces par phase qui diminue drastiquement), la dernière étape étant largement dominée par les latences car se composant d'une infime quantité de cellules/faces. La méthode est difficile à manipuler pour ajuster la granularité et espérer pouvoir en améliorer l'équilibrage de charge. Qui plus est il est difficile visuellement d'en déterminer le comportement sur cas réel, ce qui rend toute expérience complexe : la difficulté à lier les stratégies de groupement des cellules avec les variations de performances (potentiellement infimes mais cumulées) s'avère prohibitive. Il serait cependant possible d'utiliser cette approche pour une parallélisation via StarPU ou PaRSEC, mais cela nécessiterait un certain effort. Pour PaRSEC, la complexité du code initial (c.f. Codelet 9) laisse à penser que l'implémentation serait non triviale. Puisque nous nous sommes concentrés sur la reconstruction par face que plus tard dans la thèse, et pour apporter une approche permettant de comparer aisément StarPU et PaRSEC, nous avons choisi une autre méthode de découpage : la coloration de graphe.

```
1   for (g_id = 0; g_id < n_b_groups; g_id++) {
2
3     # pragma omp parallel for private(face_id, ii, ll, mm, ...)
4       for (t_id = 0; t_id < n_b_threads; t_id++) {
5
6         for (face_id = b_group_index[(t_id*n_b_groups + g_id)*2];
7             face_id < b_group_index[(t_id*n_b_groups + g_id)*2 + 1];
8             face_id++) {
9
10            ...compute...
11            }
12      }
13  }
```

Codelet 9: Boucle principale de calcul des gradients par face dans le code actuel.

II.1 Exclusion de tâches

La capacite à pouvoir exprimer les dépendances de tâches par exclusion est une problématique cruciale en ce qui nous concerne puisque il n'y a pour la plus grande partie de notre problème, pas de réelle dépendance. La logique d'exclusion est l'expression la plus synthétique de notre algorithme de tâche.

La logique est simple, il s'agit de pouvoir exprimer une relation de dépendance spéciale entre deux tâches A et B, de telle sorte que celles-ci ne puissent pas être ordonnancées en même temps (et sans que le développeur ait à se soucier de l'ordonnancement au-delà de la définition de cette contrainte).

Il n'y a pas à notre connaissance et à ce jour de solution permettant d'exprimer cela que ce soit dans ParSEC ou dans StarPU.

Une approche permettant de simuler cela serait de passer par une coloration de notre graphe de dépendance. En insérant ainsi les tâches par type de couleur et en imposant une barrière de synchronisation entre chaque couleur, on s'assure ainsi que notre exclusion soit respectée. L'approche pose la problématique de l'ajout de synchronisations supplémentaires, imposant à l'utilisateur de bien choisir la granularité des tâches et/ou au support exécutif de savoir adapter la charge sur chacun de ses processus de manière équitable. On propose par la suite une solution pour l'implémentation d'un tel algorithme.

II.2 Coloration de graphe

La coloration de graphe est une discipline sous partie de la théorie des graphes qui trouve son fondement dans la nécessite de colorier des cartes avec un nombre minimal de couleur, sans qu'aucun pays ne possède la même couleur qu'un de ses pays limitrophes. Une introduction plus détaillée est hors du sujet d'intérêt de cette thèse, mais il est toutefois intéressant de noter l'intérêt de cette pratique dans le cadre de la programmation parallèle : en affiliant à des zones de notre maillage une couleur selon la même règle, on s'assure ainsi l'indépendance du calcul de chacune des couleurs (puisque aucune zone ayant la même couleur ne sera en contact, elles ne partageront ainsi pas de donnée). La qualité de la coloration dépendra de l'algorithme choisi [4]. On souhaite explorer ici avant tout notre capacité à exprimer la reconstruction de gradient par face de manière simple et efficace.

Afin de faciliter la création de notre coloration de graphe, on peut se baser sur l'étape de prénumérotation des cellules de *Code_Saturne*. Cette dernière utilise une version à grande granularité du maillage. En utilisant ce dernier avec un ratio nombre de cellules sur nombre de blocs approprié (on souhaite comme auparant avoir une taille de bloc réduite, c'est à dire bien en dessous du nombre optimal de cellules par processus MPI tel qu'il est actuellement. Une taille de bloc supérieure ne pourrait pas permettre d'observer une quelconque différence dans l'équilibrage de charge) on obtient alors une description de notre maillage sous une forme parfaite pour nos besoins, c'est à dire un ensemble de blocs de cellules et un ensemble de couleurs, tel que chaque bloc ait une couleur associée. Le pseudo code 4 montre une vue généraliste de l'algorithme dans ce cadre. On peut d'ores et déjà observer comment l'algorithme se positionne naturellement dans la catégorie du parallélisme de tâches de type boucle.

```
1: procedure Gradient Reconstruction
2:     sort(colors, byHighestLength)
3:     for each c in color group do
4:         for each b in block of c do
5:             T[c,b] ← InsertTask(compute_gradient, c, b)
6:         Wait(T[c,b])
```

Pseudo-code 4: Calcul parallele des gradients en utilisant une coloration de graphe

L'algorithme de coloration choisi est l'algorithme DSATUR[28] proposé initialement en 1979. Cet algorithme est séquentiel et de multiples variantes potentiellement plus adaptées ont été proposées depuis. L'objectif pour nous est d'évaluer notre capacité à exprimer la reconstruction de gradients

4. La caractérisation de la qualité d'une coloration est propre à l'utilisation que l'on en fait. On peut dénombrer comme critères de qualité le nombre de couleurs obtenues par notre algorithme mais aussi l'équilibrage des couleurs ainsi obtenues. C'est ce dernier critère qui nous importe le plus pour nous, car la problématique initiale de la reconstruction de gradient par face vient en effet d'un découpage du maillage en zones indépendantes avec une dernière phase de calcul trop petite par rapport à son coût de synchronisation.

à partir d'une coloration déjà existante, et non pas d'en obtenir une qui soit parfaite, bien que cette étape peut sembler primordiale, elle est néanmoins à pondérer : une granularité assez fine alliée à la gestion de l'équilibrage de charge de PaRSEC ou StarPU aura un impact moins sujet à la qualité du maillage. D'autre part, la coloration étant effectuée sur une vue à gros grain du maillage, les performances n'ont que peu d'importance. L'auteur en a qui plus est démontré l'efficacité et la qualité dans 90% des cas[28].

II.3 Implémentation avec ParSEC et StarPU

Une fois la coloration effectuée, les évolutions de notre implémentation initiale via StarPU sont moindres et très similaires au pseudo code écrit en amont. Il en est relativement de même pour ParSEC. La facilité d'expression de l'algorithme ne diverge au final pas spécialement de la reconstruction des gradients par face initiale et met en avant encore un fois la plus grande facilité d'utilisation de StarPU.

II.4 Gestion des halos

La gestion des halos peut s'opposer en partie à l'utilisation d'une coloration, et si l'on souhaite pouvoir recouvrir les communications par des calculs, il nous faut s'assurer de pouvoir ordonnancer en priorité les tâches n'ayant pas de dépendance avec d'autres processus. Il vient alors deux choix possibles,

- utiliser/proposer une nouvelle coloration qui permettrait d'isoler les blocs ayant des halos sur un minimum de couleur et ordonnancer les tâches de ces couleurs en fin [5]
- créer un index supplémentaire pour lister les sommets dans chaque couleur qui a besoin d'un échange de halos. On peut alors trier les couleurs par quantité de blocs ayant une dépendance et par ordre croissant et trier chaque couleur de telle sorte que ses blocs dépendants soient en fin de liste. On limite de cette façon les chances d'ordonnancer une tâche nécessitant des halos par encore reçus.

5. En 2D, le problème semble assez simple, il s'agirait de faire en sorte que les sommets du graphe positionnés à son extrémité se voient assignés une couleur de telle sorte que le nombre de couleurs assignées à ces sommets là soit de 2. Ces sommets peuvent être discernés des autres pas leur degré inférieur.

III Travaux existants

Les travaux présentés dans [46] mettent en avant des résultats intéressants en milieu intra-nœud. Les auteurs s'attachent à la comparaison d'une implémentation par tâches "maison" [63] de leur solveur avec l'essai des supports OpenMP et StarPU. Les résultats montrent une plus grande efficacité d'OpenMP mais les auteurs appuient l'importance du support intra-nœud offert par StarPU pour un futur proche. Des mécaniques d'optimisation de la gestion des graphes sont proposés avec un certain succès en milieu intra-nœud. Il y est mis aussi en avant un modèle de calcul de l'efficacité d'un support par la mise en relation de plusieurs métriques d'efficacité : le parallélisme exploité, l'efficacité des tâches, du support, et du pipeline. Ce modèle est plus largement présenté dans [3, 74]. Ce modèle permet notamment de constater de l'augmentation de l'efficacité générale avec l'augmentation de la taille du problème d'entrée. Le modèle ne permet cependant pas de comparer en profondeur différents supports puisque ne prenant pas en compte les aspects de praticité/maintenabilité et autres aspects présentés dans ce manuscrit et se limite à la mise en relation des aspects GPGPUs avec un support exécutif de tâches. Les auteurs poussent dans [47] leurs expériences plus loin en y ajoutant l'étude du modèle paramétré via PaRSEC (toujours en intra-nœud), rendant leur retours très intéressant vis à vis de nos propres expériences. On constate de manière similaire des résultats encourageants en intra-nœud et peu différents pour les trois supports (OpenMP, PaRSEC et StarPU). Quelques directives sont établies pour un passage en intra-nœud, notamment sur les aspects de distribution des données et la réduction des échanges. Cet aspect est à pondérer par la nécessité des échanges dès lors que l'on souhaite passer d'un parallélisme en mémoire partagée à un parallélisme par communication (MPI).

De nombreux travaux proposent une étude du paradigme de tâches sur des solveurs directs creux et leurs portages sur des plateformes faites de GPGPUs. La présence des supports exécutifs StarPU et PaRSEC y est largement représentée, notamment avec les travaux de Lopez *et al.* [74] ou encore Agullo *et al.* [6] et [23, 25, 22]. On peut d'ores et déjà noter la grande proportion des travaux s'attachant à l'importance de l'hétérogénéité des architectures (usant de GPGPUs [96, 5, 4, 2]) ou multi-cœurs et son impact sur les performances des codes scientifiques. La quantité de calculs dégagée par les solveurs directs ainsi que les récentes avancées sur les factorisations des matrices (e.g Cholesky[63, 46], QR[31, 84], LU[4]) offrant une grande opportunité de gains avec une approche de parallélisme par tâche.

Agullo *et al.* présentent dans [6] une étude de la scalabilité d'un solveur creux (calcul de gradient conjugué) via l'utilisation de StarPU. Leurs travaux se concentrent uniquement sur l'étude de la scalabilité avec une configuration de mono à multi-GPUs. Ils proposent un modèle de granularité intéressant en découpant le gradient conjugué en plusieurs tâches (8), et font appel à des regroupements et partitionnements réguliers des données afin de pouvoir répartir correctement la charge. Ces mouvements limitent en conséquence les performances de l'implémentation (notamment par l'imposition de barrières de synchronisation, et un passage potentiel des données des GPUs à la mémoire principale (CPU) avant d'être redistribuée), limitations qui sont adressées avec succès dans un second temps. Ces travaux se détachent de notre approche par leur concentration sur le développement en milieu multi-GPGPU et de cette spécificité. Une partie des mécaniques reste toutefois valable dans le cadre d'une exécution multi-cœurs, et apportent un éclairage sur la méthodologie à mettre en place pour une transition vers une simulation via StarPU sur des GPGPUs. Enfin, l'importance de la granularité dans cette approche doit être remise en situation vis à vis de nos propres contraintes [6],

6. L'intérêt d'un point de vue performance d'une approche par tâches est pour nous d'arriver à apporter un meilleur équilibrage de charge afin d'espérer gagner en performance. Pour ce faire, il nous faut avant tout arriver à augmenter la granularité (réduire la taille des tâches) de telle sorte à pouvoir affiner la répartition du travail au court du temps. Les résultats présentés dans [6] sont axés sur un gain de performance via une maximisation de l'apport en données par une granularité moins fine. Dans notre cas, l'objectif est d'arriver à descendre en dessous de la granularité optimale de *Code_Saturne* (sur des exécutions MPI (+OpenMP)), soit en dessous de la barre des 15000 cellules par rang.

les résultats ayant été obtenus sur une configuration assez éloignée de celles des supercalculateurs EDF [7].

Pour pallier les difficultés à exploiter les supports exécutifs avec une granularité basse, Rossignon *et al.* propose dans [89, 90] de regrouper les tâches de telle sorte à en créer des plus conséquentes. L'approche est intéressante mais les résultats obtenus et la conclusion de l'auteur se heurtent à des observations similaires aux nôtres : obtenir des performances supérieures à celles de MPI est une problématique complexe. L'auteur propose des mécaniques d'optimisation des placements des tâches pour optimiser les accès mémoires pour un code de simulation usant de maillages structurés. D'autres travaux visent à répondre à la même problématique de granularité en permettant de gérer différents niveaux de granularité, de regroupement de tâches par paramétrisation utilisateur ou encore de paramétrer des tâches comme étant divisibles (adresser la problématique de tâches à granularité trop grande) [78, 57, 56]. De telles approches peuvent être intéressantes dans le cadre d'une simulation sur des unités de calculs de natures différentes (mixe GPGPUs et CPU) de telle sorte que la granularité adaptée à l'une ne soit pas adaptée à l'autre et qu'il faille opérer des ajustements de « dernière minute » sur la granularité des tâches. Hors de ce cadre, l'utilisation d'un modèle de performance ou la simple connaissance des tailles des tâches optimales pour la machine utilisée suffira.

Danalys *et al.* évalue dans [41] la plus-value d'une transition vers PaRSEC depuis un code industriel (NWCHEM). NWCHEM est parallélisé via l'utilisation de MPI et bénéficie de mécanismes poussés comme la mise en place de vol de travail entre unités de calcul et de communications par Global Arrays. L'utilisation de point de synchronisation par Global Arrays (notamment pour la mise en place du vol de travail) est identifiée comme gênante pour un passage a l'échelle. Les auteurs obtiennent un gain de performance d'environ 25% à 50% sur le passage du BLAS gemm [8]. On peut noter la faible scalabilité du code initial qui possède des performances quasi similaires au-delà de 3 processus par nœud (nœuds composés de processeurs Xeon E5-2670 à 8 cœurs hyperthreadés (ce qui est bien en deçà des performances de *Code_Saturne* dans des conditions similaires)).

Boillot *et al.* présentent dans [19] d'excellents résultats pour un portage d'un code industriel sous MPI. L'étude se limite aux performances en intra-nœud et plus précisément sur des architectures "many core" ce qui laisse penser que des difficultés similaires aux nôtres ont été rencontrées pour une exécution avec plus d'un nœud. On peut noter que les performances du code initial sur Xeon Phi sont déjà très bonnes : efficacité de 80% sur 48 cœurs contre 95% avec PaRSEC (l'architecture Xeon Phi fait partie des architectures récentes les plus complexes quand il s'agit d'obtenir des performances satisfaisantes compte tenu de la puissance théorique de cette carte accélératrice).

De la même manière, les résultats présentés dans [80] par Moustafa *et al.* montrent de bons résultats pour la résolution de l'équation de Boltzman via un algorithme de alayage en milieu cartésien que ce soit pour StarPU ou PaRSEC ou Intel TBB. Les travaux se concentrent sur la prédiction des performances dans le but d'optimiser la granularité des tâches. Leurs conclusions pointent notamment une meilleure capacité de PaRSEC à gérer des tâches de petite granularité comparé à Intel TBB et StarPU (résultats à tempérer car présentés sur un seul cœur, et datant de 2015, ce qui est relativement éloigné comparé à la date de création des dits supports exécutifs) et globalement une légère supériorité de PaRSEC dans sa capacité à passer à l'échelle. Notons ici qu'à l'instar de nos travaux, conclure sur les performances respectives des deux supports reste un exercice qui se cantonne d'une part à notre capacité à bien exploiter ces derniers, mais aussi à un instant « t » et un algorithme donné.

Enfin Couteyen *et al.* montrent dans [33] la compétitivité de StarPU dans certaines conditions

7. Pour rappel, l'avant dernier cluster d'EDF EOLE sorti en 2016 est composé de processeurs à 14 cœurs et 32gb de RAM (soit un peu plus de 2gb par cœurs). Comparé aux 6gb de RAM des GPGPUs de l'étude, la quantité de mailles allouables à chaque unité de calcul est quasiment 3 fois moins grande.

8. Le BLAS gemm est largement utilisé dans le monde scientifique, et a une place de choix dans la mise en place des étalonnages de performance de la bibliothèque Linpack. Il est aussi utilisé lors d'une factorisation de Cholesky, algorithme largement étudié pour l'évaluation des performances des supports exécutifs à paradigme de tâche

pour la parallélisation de FLUSEPA. Leurs conclusions sont finalement assez proches des nôtres. Lesdits travaux se distinguent dans la compréhension de la nécessité d'utiliser StarPU comme support exécutif en tant que solution intermédiaire, en cumulant StarPU à granularité haute en addition d'un parallélisme à l'intérieur des tâches via OpenMP. Cette approche est intéressante dans un cadre d'obtention pure des performances. La concentration des auteurs sur des échanges MPI explicites confirme nos conclusions quant aux différentes options offertes pour l'exploitation des ressources distribuées via StarPU. Enfin, les résultats montrent tout de même et à l'instar de nos propres conclusions, une tendance à de meilleures performances pour une solution à base de MPI+OpenMP que de part l'utilisation de StarPU, et ce malgré une implémentation de FLUSEPA via MPI très récente [32] dans le code.

IV Résultats

La figure 4.8 présente une comparaison des performances obtenues sur un seul nœud. Evaluer les performances sur un seul nœud peut sembler léger si l'on souhaite évaluer la scalabilité d'un code mais permet d'évaluer les performances en intra-nœud qui sont primordiales à une bonne scalabilité. L'objectif ici est de montrer comment chacune des implémentations se comporte en fonction du nombre de processus (léger ou non) et l'augmentation de la granularité : le maillage de la simulation ne change pas en fonction des cas, seul son partitionnement (131072 cellules sur deux unités de calculs a 10922 sur 24 unités de calcul).

La performance optimale de l'implémentation initiale de *Code_Saturne* est présentée en orange et n'est pas fonction du nombre de processus (paramètre fixé à 14 processus MPI (soit le nombre de cœur réel) et 2 processus légers OpenMP par processus MPI.

On observe que pour StarPU, les performances optimales se situent aux environs de 8 processus, soit environ 1 tiers des cœurs (si l'on prend en compte l'aspect hyperthreadé, on avoisine 60% de la machine si l'on compte que les cœurs réels) pour un nombre de cellules par processus de 32768. Dans cette configuration, StarPU obtient les performances les plus satisfaisantes et motive l'essai d'une exécution hybride (2 à 3 processus MPI par machine, chacune exécutant StarPU).

Les performances obtenues avec ParSEC montrent une évolution plus proche de celle de MPI, avec un léger gain avec 24 processus légers tout en avoisinant les performances optimales dès 10-14 processus.

Comparés aux performances de *Code_Saturne* lorsqu'usant uniquement d'OpenMP, on peut constater que les résultats sont particulièrement satisfaisants : si les performances locales d'OpenMP sont très médiocres dès que l'on s'appuie uniquement sur eux pour exploiter le parallélisme d'une machine, celles de StarPU et ParSEC s'approchent voire dépassent légèrement celles du cas optimal.

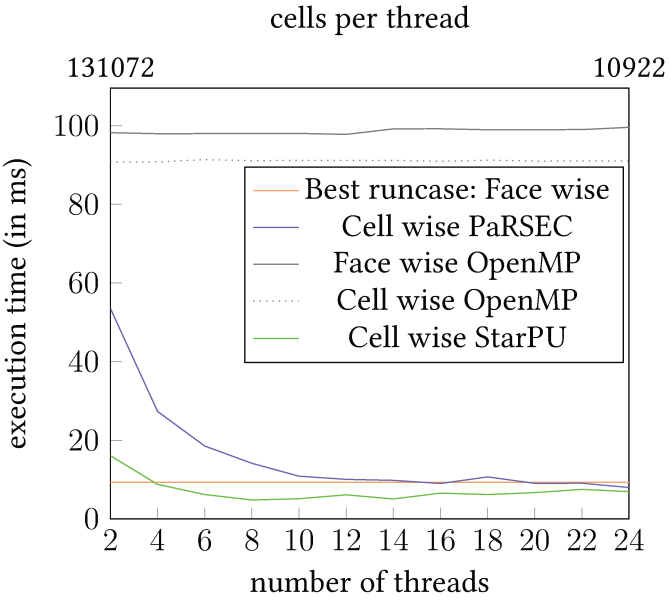


FIGURE 4.8: Comparaison des performances des différentes versions sur un seul nœud.

Les sous-sections suivantes détaillent les résultats des expériences faites avec StarPU et ParSEC.

IV.1 Detail des résultats avec StarPU

La figure 4.1 présente l'ensemble des combinaisons essayées avec StarPU. Les figures 4.9, 4.10 et 4.11 présentent les resultats d'une partie des expériences avec StarPU en inter-nœud.

cas	dépendance	halos
A	implicite	starpu_mpi
B	implicite	starpu graphe global
C	explicite	starpu_mpi
D	explicite	mpi_irecv
E	explicite	mpi_recv

TABLE 4.1: Liste des cas essayés avec Starpu.

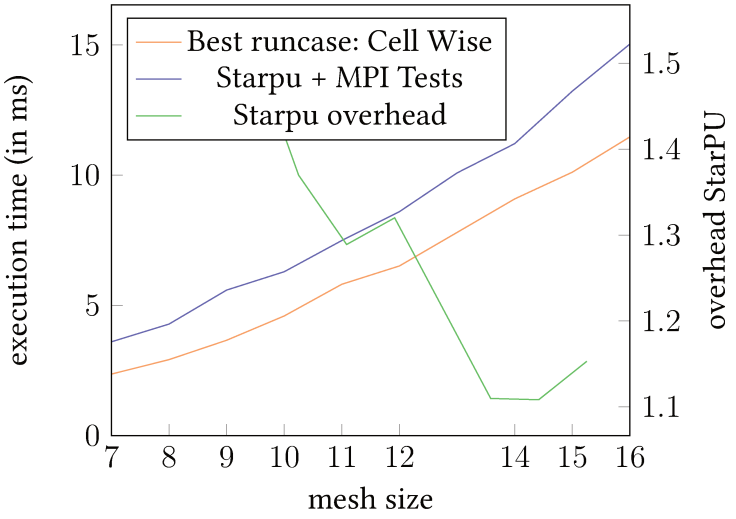


FIGURE 4.9: Comparaison des performances de *Code_Saturne* initiales sur deux nœuds avec StarPU en utilisant des tâches de communications explicites (MPI) en usant de directives MPI_Test. On observe que la version avec StarPU prend plus de temps, mais le ratio de temps supplémentaire diminue avec l'augmentation de la taille du maillage.

La figure 4.10 confirme nos premières observations : on tend à gagner légèrement en performance en augmentant le nombre de processus MPI par nœud, même si cette tendance est bien inférieure au comportement de l'implémentation MPI + OpenMP de *Code_Saturne* (code initial). Contrairement à une exécution sur un seul nœud, on ne gagne pas de performance vis à vis du code initial (on en perd même, de 50 a 10% en fonction des cas, *c.f* présentation du surcoût en figure 4.9). Enfin la figure 4.10 montre le besoin d'utiliser les capacités de StarPU en terme d'expression des échanges (*c.f* StarPU + Global DAG) même si l'on perd en efficacité au fur et à mesure que la taille du maillage augmente [9]. On constate néanmoins qu'une gestion des dépendances en données via des échanges MPI externes permet un écart avec les performances du code initial inférieur.

La figure 4.12 présente un graphe GANTT permettant d'évaluer le comportement de notre implémentation. Chaque bloc bleu représente une tâche de calcul, et sa taille signale le temps qu'elle a pris. Sa position verticale la positionne sur le cœur l'ayant exécutée. Les autres blocs mis à part celui du cœur 0 sont déplacés sur des indexes imaginaires afin d'améliorer la lisibilité et dont voici la correspondance :

— *commt* tache de communication

9. Puisqu'on souhaite garder une granularité similaire, l'augmentation de la taille du maillage signifie l'augmentation du nombre de tâches, et avec ces dernières, l'augmentation du nombre de tâches dépendantes d'un change de données et le nombre d'échanges de données effectuées. Cette augmentation peut devenir gênante en fonction de la granularité essentiellement : StarPU s'occupe d'ordonnancer les tâches en fonctions du relâchement de leurs dépendances via un graphe de dépendance, néanmoins il se peut qu'il n'y ait plus d'autres tâches à ordonnancer que celles en attente de données, ce qui peut arriver si la quantité de travail est inférieure au temps d'échange des données.

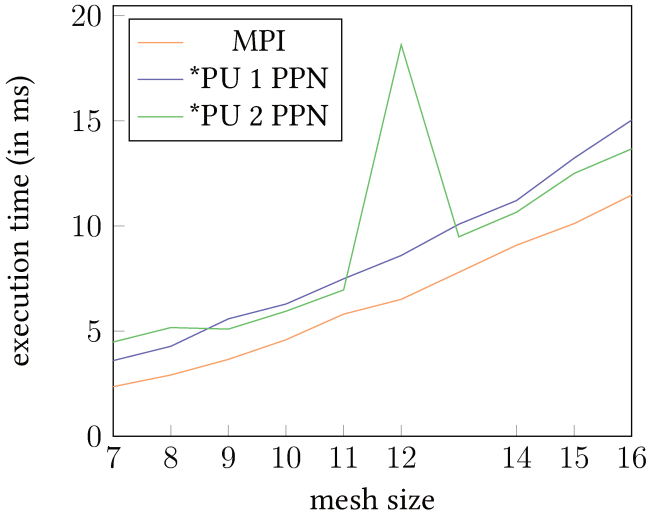


FIGURE 4.10: Comparaison des performances sur 2 nœuds en augmentant le nombre d'instance StarPU par noeud, ici 1 processus par nœud (1 PPN) contre 2 processus par nœud (2 PPN)

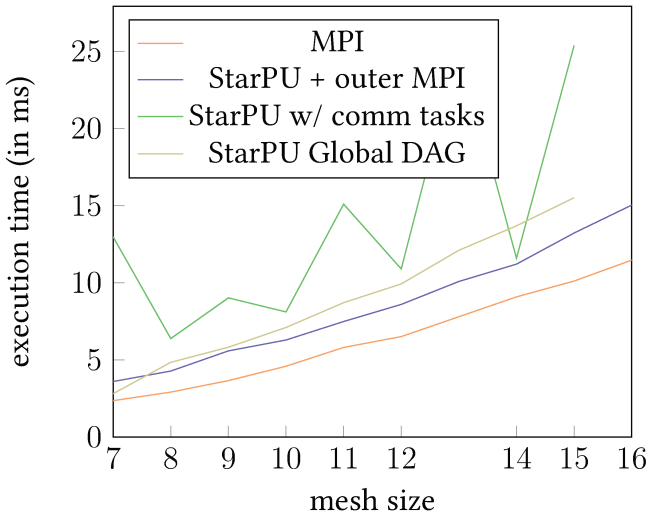


FIGURE 4.11: Comparaison des performances sur 2 nœuds

— $fullr$ chrono prenant en compte la totalité de la reconstruction
— $tillw$ chrono entre le début de la reconstruction et après l'attente de fin des tâches
— $submis$ chrono du début à la fin de la soumission de l'ensemble des tâches

Il est difficile sur ce GANTT de déterminer une problématique liée aux communications. Il semble que la granularité des tâches pourrait augmenter pour espérer gagner légèrement en temps (bien que le gain ainsi espéré ne soit ici pas excessif). Les résultats précédemment présentés laissent à penser que la plus grande problématique reste l'optimisation de la gestion des échanges MPI.

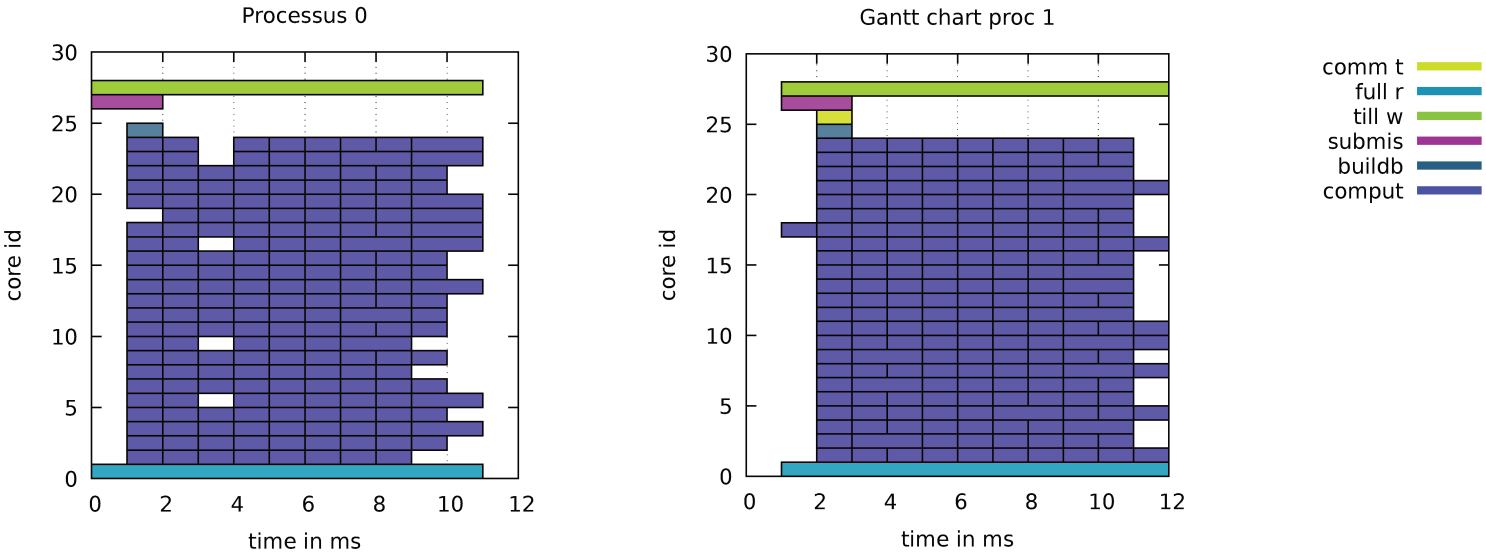


FIGURE 4.12: Graphe GANTT d'une exécution de la reconstruction des gradients avec StarPU avec échanges des halos via MPI_Isend/I_Recv. On constate un bon équilibre de charge. L'échange des halos ne semble pas impacter les performances, voire est transparent.

IV.2 Retour d'expérience

On se propose de résumer dans un tableau (*c.f* 4.2 de manière objective les différents points qui font ou défont le succès de telle ou telle bibliothèque pour le développement d'algorithme via des taches sur *Code_Saturne* (et donc en comprenant les spécificités d'un code de simulation en CFD usant de maillages non structurés). D'autres aspects ne sont ici pas pris en compte, tels que la disponibilité des équipes ou l'existence d'outils de déboguage sur lesquels il est difficile de faire une différence, que ce soit à cause de la représentativité de ce sur quoi la comparaison est faite ou parce que la différence n'est pas significative. La sous-section suivante détaille plus largement le retour d'expérience quant à l'utilisation des deux bibliothèques PaRSEC et StarPU.

IV.2.1 Appropriation du support

PaRSEC : L'utilisation du support PaRSEC impose sur cet aspect le plus de difficulté au développeur. L'addition d'un nouveau langage à maîtriser, en sus d'une documentation allégée semble un frein majeur à l'adoption du support. On peut aussi noter (lignes 5 à 16 de la tâche 1 par exemple page 61) l'intégration de code C/C++ via l'utilisation des séquences d'échappements %{ et %} qui permettent, lorsqu'il n'est pas possible de faire autrement, d'exécuter une suite d'instructions pour déterminer la valeur d'une variable spécifique à une tâche. Dans notre cas, le nombre de voisins et la nécessité de communiquer avec eux ou pas étant indexée (c.f. $halo \rightarrow c\_domain\_rank$ et $halo \rightarrow send\_index$ par exemple, accédés pour la récupération du véritable indice de rang voisin (n_idx) et du nombre d'éléments à lui échanger (c) doivent être préalablement déterminés. On

support	1er contact	documentation	productivité	éloignement
OpenMP $\geqslant 4$	●	●		●●
PaRSEC	●			
StarPU		●●	●	●

TABLE 4.2: Classification de supports exécutifs testés selon différents critères. On entend ici par premier contact le premier sentiment ressenti. Se démarque souvent dans cette catégorie les bibliothèques aux apparences simples. L'éloignement fait lui référence à la différence entre le code obtenu et le code initial. Il est ici estimé qu'un code résultant dont la différence est moindre avec le code initial est un critère important : il représente d'une certaine manière l'investissement nécessaire aux anciens développeurs comme aux futurs pour la transition ou la prise en main du code résultant.

remarque alors que malgré la simplicité de l'algorithme à implémenter, le modèle paramétré via l'écriture de code JDF avec PaRSEC perd une bonne partie de ce qui fait son élégance lorsque mis en usage sur un code gérant des maillages non structurés.

StarPU : Sur cet aspect, StarPU a l'avantage d'être plus facile à s'approprier, et s'interface aisément avec un code C ou Fortran existant. De nombreuses facilités sont mises à disposition pour la gestion des cas simples (majoritairement utiles pour des cas structurés notamment). Dans notre cas, une tâche pouvant dépendre d'un nombre variable et inconnu avant exécution de données voisines, on ne peut bénéficier des outils tels que "starpu_mpi_task_insert". La mise à disposition d'outils pour ce genre de cas n'entache cependant pas la possibilité de gérer des cas plus compliqués comme c'est notre cas [10].

IV.2.2 Une solution tout-en-un

On peut cependant noter que –lorsque utilisées dans le cadre d'un logiciel déjà existant et sur lequel des travaux sur le parallélisme à la fois intra et inter nodal ont été effectués de manière régulière et sur une plage temporelle importante– les deux solutions peuvent apporter une difficulté supplémentaire qui ne serait pas perçue si l'on devait re-implémenter de zéro un logiciel et/ou sa version parallèle. Toute une partie de l'approche répartition des données et de l'attribution de ces dernières à une unité de calcul spécifique est notamment contrecarrée par l'existence d'un partitionnement déjà effectué via des outils tels que SCOTCH ou PARMETIS. Un effort supplémentaire est ainsi demandé aux développeurs pour que l'interfaçage entre les deux méthodes soit bien pensé. Ces deux supports se voulant des solutions tout-en-un, il a été notamment nécessaire de bien spécifier à PaRSEC comment accéder aux données de *Code_Saturne*. Côté StarPU, cette problématique a pu être évitée.

IV.2.3 Performances inter-noeuds

StarPU : Si l'approche offerte par StarPU via les directives starpu_mpi semble être l'approche idéale quelque soit le code, en pratique elle comprend certains défauts.

D'une part les performances obtenues avec StarPU et des échanges MPI purs (explicites) sont largement meilleurs. Cet écart significatif n'a pas pu être clairement expliqué bien qu'il semble être intuitivement lié à des problèmes d'ordonnancement. Il en résulte une attente active de l'ensemble du parc des processus légers de StarPU avant l'ordonnancement des tâches finales nécessitant les dits échanges de données. Malgré de nombreux développements et essais de placement de processus

10. Il semble que les outils proposant des solutions élégantes permettent souvent de résoudre très facilement des problématiques déjà simples au départ. Il peut être plus compliqué d'utiliser de tels outils dans des conditions réelles.

différents et plusieurs itérations avec l'équipe de développement du support exécutif, il n'a pas été trouvé de solutions permettant d'obtenir les performances espérées et en accord avec la qualité des performances obtenues avec StarPU sur un seul nœud. Ce premier défaut laisse à penser à un manque de maturité du support quant à la gestion des communications MPI. La multiplicité des solutions offertes par StarPU ne facilitant pas à la résolution de tels problèmes.

La reconstruction des gradients par méthode de moindres carrés n'étant pas itérative, la libération de dépendances de tâches ne se fait que pour un seul appel à cette reconstruction de telle sorte que l'on ne puisse pas profiter d'un asynchronisme maximal. Ce schéma est néanmoins représentatif de l'ensemble de *Code_Saturne*. Les schémas de résolution mis en place dans *Code_Saturne* imposant des synchronisations fréquentes, il n'est pas envisagé de passer l'ensemble du code en asynchrone. Partant de ce constat, l'utilisation des directives de type "starpu_mpi" ne peut se faire que via la mise en place d'une solution singulière : les tâches qui dépendraient des halos se voient assignées une dépendance vers une tâche de contrôle dont l'exécution est vide. Cette dernière est ordonnancée à la fin d'un "starpu_mpi_irecv_detached" via l'exécution de sa fonction retour. En théorie, le nombre de tâches supplémentaires ainsi soumises est relativement restreint, le nombre de voisins pour chaque processus étant limité.

Cette approche ayant été décevante quant à ses performances, nous nous sommes ensuite attachés à explorer d'autres possibilités en sortant la partie MPI de StarPU, ce qui a permis d'obtenir les meilleures performances bien que l'approche ne soit pas satisfaisantes quant à la volonté initiale d'utiliser un support exécutif tel que StarPU.

La zone de performance optimale de notre implémentation semble aussi décalée par rapport à la zone de performance optimale constatée sur le code initial (besoin d'une granularité 3X plus faible avec StarPU). La sensation que la granularité à laquelle les performances sont optimales avec StarPU étant plus faible que celle avec le code initial laissent à penser que la scalabilité de nos développements sera moindre vis à vis du code initial.

PaRSEC : Les tentatives d'obtention de performances un tant soit peu correct avec PaRSEC dans des conditions inter-nœuds furent toutes vaines malgré un travail en collaboration étroite avec un des membres développeur du support. La complexité de son utilisation couplée avec le temps nécessaire à obtenir des performances en inter-nœuds (compte tenu du temps passé au cours des travaux ici présentés et de la non obtention des dites performances) semblent ici apporter de nombreux éléments prohibitif à son utilisation pour un logiciel aux propriétés similaires à celles de *Code_Saturne*.

IV.2.4 Gestion des échanges

Initialement effectué via la bibliothèque MPI, un des intérêts majeurs que les supports exécutifs peuvent nous apporter est de fournir une solution incluant tout en son sein. On peut ainsi limiter le travail d'optimisation des performances qui peut être conséquent lors d'une implémentation sur une base de MPI + X, (dans notre cas MPI + OpenMP, avec un gain en performance via OpenMP relativement faible). Si on peut par ce biais améliorer ou a minima maintenir les performances de *Code_Saturne* tout en facilitant sa maintenance ou son évolution vers d'autres architectures ou algorithmes ce support exécutif aura alors pour nous une plus-value vis à vis d'un modèle basé sur MPI + OpenMP (on n'évaluera pas cet aspect ici puisqu'il a déjà été abordé auparavant).

Sur cet aspect, les deux supports permettent une gestion implicite des échanges en ne spécifiant que les « flux de données ». PaRSEC en usant le PTG met particulièrement cette notion en jeu et en fait son atout principal. Si l'aspect de la solution est séduisante puisqu'elle met en avant une vision « haut niveau » du développement, son utilisation sous-tend une partie bas niveau plus complexe à mettre en place compte tenu de la complexité des structures de données dans *Code_Saturne* (spécifier de manière à limiter les coûts et de manière simple quelle partie de la donnée doit être envoyée à tel ou tel voisin et ce en fonction de l'identifiant de chaque tâche n'est pas une mince affaire). StarPU

quant à lui a l'avantage de permettre les deux solutions en permettant une approche d'échange automatique des données par définition d'un graphe de tâches globales avec une approche qui facilite, non pas le raisonnement comme c'est le cas de PaRSEC, mais le développement. Le développeur peut qui plus est se retrancher sur l'approche plus « terre à terre » en usant des outils d'explicitations des communications mis à disposition ($starpu\_mpi\_(Send/Recv)$). L'ensemble permet au final d'expérimenter rapidement avec la seconde approche avant de transitionner vers un graphe global [11]. Troisièmement, le support permet comme nous l'avons utilisé, de spécifier les dépendances en données manuellement sans toutefois passer par un graphe global (trop contraignant ici). Cette dernière solution permet de facilement s'extraire des difficultés de la définition d'un graphe global sans toutefois devoir déclarer de manière explicite des échanges de données.

V Perspectives

V.1 Transition vers un graphe de tâches global

La création d'un graphe de tâches global permettrait de maximiser le potentiel d'optimisation qu'un ordonnanceur de tâches pourrait effectuer. Il pourrait par exemple privilégier localement une tâche sur une autre en fonction de ce que cette première pourrait libérer en dépendance sur des tâches distantes. Permettre cette transition depuis les structures de données de *Code_Saturne* faciliterait l'utilisation des supports exécutifs tels que PaRSEC (où cela permettrait d'éviter certaines pratiques qui nous ont été obligatoires comme la création de tâches vides en masse afin de simplement pouvoir définir notre graphe de tâche, pratique qui ne facilite pas la compréhension, le déboggage et est très probablement mauvaise pour les performances globales de PaRSEC pour notre cas [12]).

Dans notre cas définir un graphe global revient à ce que chaque processus ait une vision de comment le maillage a été réparti sur l'ensemble des entités. Or *Code_Saturne* répartit les données au plus tôt sur l'ensemble des processus utilisés lors d'un calcul, et la nature non structurée du maillage empêche toute déduction facile quant à cette répartition, aucun processus ne dispose donc actuellement d'une vue globale de l'ensemble du maillage. Par contre, à chaque élément du maillage (cellule, face, sommet), on associe un numéro global, permettant lors des opérations d'entrées/sorties ou pour le re-partitionnement de disposer d'une vue « globale par blocs » du maillage, où chaque rang MPI (ou un sous-ensemble des rangs) gère un bloc.

Par exemple, pour les cellules du maillage partitionné en figure 4.13, on peut avoir la distribution vers 2 blocs (un bloc / 2 procs) suivante :

La figure 4.14 présente la même opération sur les faces du même maillage afin de maintenir la cohérence des données.

En utilisant ces deux opérateurs sur la base d'une représentation plus grossière du maillage, telle que l'on ait un bloc de cellules/faces par tâche, on pourrait définir une numérotation globale des tâches. La construction des éléments grossiers, et surtout la construction de halos « grossiers » peut être élaborée à partir de l'adaptation du code existant de construction des grilles grossières du multigrille algébrique disponible dans le code (en agglomérant les éléments associés à une même tâche). Cette construction grossière a déjà été adaptée dans la thèse pour le coloriage nécessaire à la version par face.

L'intégration récente mais non disponible durant cette thèse des travaux apportés par [20] sur une nouvelle discrétisation appellée CDO permet une approche plus simple qui consisterait à la construction d'une numérotation globale à partir des halos :

11. On se positionne ici sur l'utilisation de ces deux supports pour l'adaptation d'un code déjà existant.

12. Concernant PaRSEC ou tout autre support utilisant le paradigme de programmation par tâche paramétrique, on peut reconnaitre que c'est le paradigme dans son ensemble qui n'est pas bien adapté pour *Code_Saturne*.

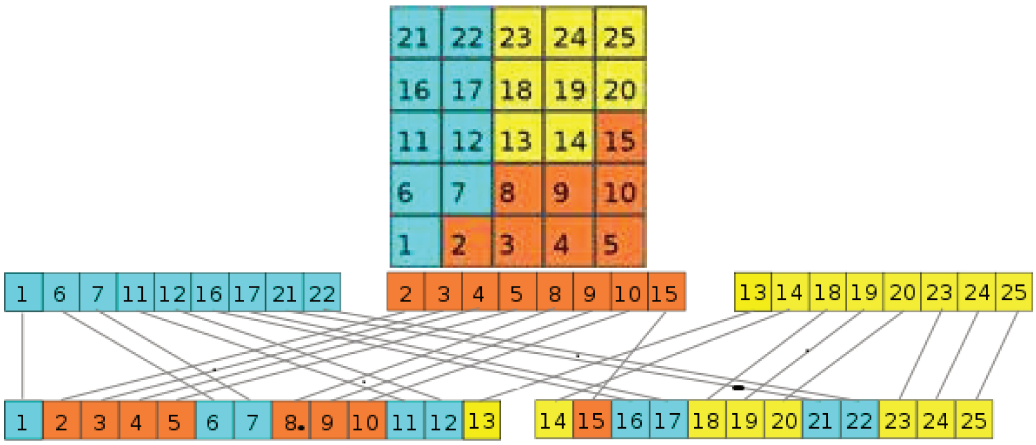


FIGURE 4.13: Un maillage (en haut) réparti sur 3 rangs avec une numérotation globale par bloc et sa distribution initiale vers une redistribution sur deux rangs.

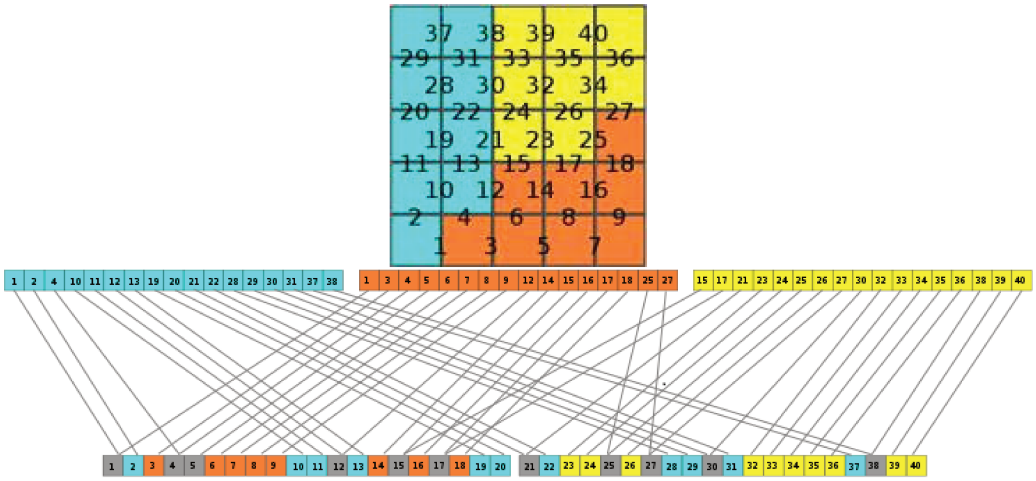


FIGURE 4.14: Explication d'un re-partitionnement des faces d'un maillage de 3 vers 2 processus tel qu'il est effectué dans *Code_Saturne*

```
1  modification() {
2
3    double *a = get_array_m(id) ;
4    // modification de a
5    free_array(&a);
6  }
```

Codelet 10: Codelet de démonstration du schéma type d'accès en écriture avec signalisation de terminaison par libération du pointeur. On utilisera en interne ce mécanisme pour maintenir la cohérence des accès écriture/lecture sur le tableau en question.

- Pour chaque tâche impliquant deux rangs MPI distincts, on associe un rang « propriétaire » (par exemple le rang le plus fort, ou celui qui a besoin des halos de l'autre)
- On numérote localement les tâches de manière à ce que les tâches dont un rang n'est pas propriétaire apparaissent après celles dont on est propriétaire
- On construit une numérotation globale par somme préfixée parallèle (i.e. MPI_Scan) sur les tâches dont chaque rang est propriétaire (i.e. première partie de la liste)
- Pour les tâches associant deux processus, on utilise le halo grossier (basé sur l'adaptation du multigrilles mentionnée ci-dessus) pour que le rang « propriétaire » transmette le numéro global des tâches concernées aux rangs « non-propriétaires »

V.2 Piste d'amélioration de l'aspect "asynchrone"

Le besoin de synchronisation dans *Code_Saturne* se fait par chaque algorithme ayant besoin des données des cellules (par exemple) voisines. Cette synchronisation peut être nécessaire à la fois lors de calcul sur plusieurs machines mais aussi sur une seule (on peut d'ailleurs argumenter que cette problématique pour *Code_Saturne* est souvent la même, la majorité du parallélisme étant exprime via MPI). On pourrait dans certains cas améliorer l'aspect "asynchrone" en faisant en sorte de ne pas nécessiter d'échange de halos en amont d'un opérateur de type "gradient" quand le tableau utilisé a déjà été synchronisé. On peut s'inspirer pour cela de la logique de maintenance de cohérence des caches processeurs (*c.f* le protocole MOESI par exemple). Une difficulté : savoir si un tableau a effectivement été modifié ou non depuis la dernière synchronisation. Cet aspect doit être traité des deux côtés à l'écriture de la donnée et à la lecture. Si la granularité des tâches le permet [13], on pourrait laisser StarPU gérer cette problématique en ayant bien la distinction des besoins en lecture et en écriture de chaque tâche. Il faudrait pour cela définir clairement des tâches spécifiques aux halos (pour la tâche modificatrice comme pour celle qui ne ferait que lire la donnée). La même logique est possible côté PaRSEC bien que l'on s'attende à une complexité accrue de l'expression de cette idée.

Si l'on voulait garder les échanges sous forme de tâches, on pourrait s'inspirer d'une des techniques d'accès aux tableaux utilisées dans PETSc ($VecGetArray/VecRestoreArray$, en l'étendant aux aspects lecture et écriture, ce qui se rapprocherait des mécanismes de cohérence de cache processeur :

- Pour accéder aux valeurs d'un tableau en lecture seule : $double * a = get\_array()$;
- Pour accéder aux valeurs d'un tableau en lecture/écriture : $double * a = get\_array\_m()$;
- Pour arrêter d'accéder à un tableau : $free\_array(\&a)$;

Ainsi, pour modifier les valeurs d'un tableau on opèrerait de la manière telle qu'elle est présentée dans les codelets 10 et 11.

13. Exécuter des tâches de la taille des halos, et cela pour chaque rang signifie potentiellement des tâches de quelques milliers de cellules tout au plus.

```
1   lecture() {
2
3     double *a = get_array(id) ;
4     // lecture de a
5     free_array(&a);
6   }
```

Codelet 11: Codelet de démonstration du schéma type d'accès en lecture avec signalisation de terminaison par libération du pointeur. La terminaison est ici moins importante. La fonction get_array doit gérer la connaissance de la disponiblite de la donnée (a-t-elle été reçue ?).

Dès que la fonction $modification$ a « libéré » le tableau, on peut démarrer la synchronisation des halos associés, sous une forme asynchrone. Lorsque la fonction $lecture$ cherche à accéder au tableau via get_array, on peut vérifier si les éventuelles synchronisations lancées en amont sont terminées, et attendre leur terminaison le cas échéant [14]. Cette approche est plus contraignante que les accès classiques aux tableaux utilisés pour les halos (halos positionnés dans la continuité des tableaux locaux) mais permettrait de bien déterminer quand un tableau est utilisé en modification ou en lecture, et gérer de manière fine et asynchrone les échanges de halos correspondants, permettant de mieux masquer les temps de soumission des échanges. Elle n'aurait par contre pas d'intérêt supplémentaire au sein d'algorithmes itératifs, où les échanges sont fréquents et où l'on a en général besoin de synchroniser immédiatement les tableaux calculés à l'itération précédente.

14. Cette action sous-tend des mécanismes ou des échanges de données supplémentaires afin d'avoir connaissance de chaque intention d'échange de données entre les deux unités de calculs en relation.

Conclusion

La complexification constante des architectures matérielles ont fait du développement scientifique actuel un domaine particulièrement compliqué. Exploiter à la perfection un super-ordinateur d'aujourd'hui hors du simple étalonnage des performances de la machine semble impossible et impose à l'utilisateur de nouvelles préoccupations. Ainsi le code doit être adapté pour prendre en compte l'hétérogénéité des machines et la nature hybride du parallélisme qui s'ensuit. L'arrivée de l'utilisation de cartes accélératrices telles que les architectures Many-cores ou les GPGPUs, dont les spécifications changent rapidement et avec elles les manières de les exploiter rendent tout investissement en elles couteux et peu durable. L'introduction récente de supports exécutifs usant du paradigme de programmation par tâches s'attache à répondre à ces problématiques multiples. D'une part, elle sépare la logique de l'algorithme initial – qui peut être perçu comme un graphe de tâches – de l' implémentation des tâches qui le composent, facilitant d'une part l'expérimentation sur de nouvelles architectures, garantissant ainsi le maintient de la pérennité du code.

Dans cette thèse nous nous sommes focalisés sur *Code_Saturne* qui est un code de simulation largement parallèle, et qui a fait l'objet d'études sur super-ordinateur dans des conditions extrêmes (105 milliards de cellules sur la totalité du super ordinateur Mira à Argonne). L'arrivée des architectures massivement parallèles ainsi que l'augmentation du nombre de cœur par machine ont fait émerger la nécessité d'un parallélisme hybride afin de maximiser les performances locales tout en diminuant la complexité des échanges. Cela se traduit par une réduction de la cardinalité du partitionnement nécessaire au calcul parallèle. La complexité des accès mémoires des algorithmes de *Code_Saturne* initialement développés pour du calcul distribué ainsi que la nature des systèmes d'équation à résoudre impactent largement la transition de *Code_Saturne* vers les nouvelles architectures de calcul. De fait, *Code_Saturne* bénéficie aujourd'hui bien plus du parallélisme distribué que de l'utilisation du parallélisme à mémoire partagée. Si l'on prend l'exemple de la reconstruction des gradients, on constate une large augmentation de la complexité du code suite au rajout d'OpenMP en combinaison de MPI, et cela pour un résultat peu engageant. Les nouveaux algorithmes mettent en avant – avec l'utilisation de l'algorithme de multi-passe – la notion d'équilibrage de charge au cœur de chacune des parties du code itérant sur les faces. On constate que la répartition ainsi obtenue est difficile à interpréter et optimiser pour en améliorer les résultats. Or cette problématique risque d'être de plus en plus impactante au fur et à mesure de l'évolution des machines de calcul scientifique.

En introduisant une nouvelle façon d'exprimer la reconstruction des gradients dans *Code_Saturne*, nous démontrons la nécessité de faire évoluer certains algorithmes afin de faciliter à la fois le support du parallélisme hybride et l'exploration de l'optimisation de l'équilibrage de charge, facilitée par la suppression des dépendances entre faces. Les performances de cette nouvelle implémentation sont encourageantes (10% moins rapide pour quasiment le double de calculs à exécuter) et vont dans le sens des évolutions matérielles (les architectures gagnent depuis les années 80 beaucoup plus en capacité de calcul qu'en vitesse mémoire, c.f "memory wall'). Elles laissent aussi à penser que des travaux d'optimisations supplémentaires pourraient grandement profiter au code. Enfin, la reconstruction des gradients par itération sur les cellules facilite une transition vers l'utilisation de supports exécutifs permettant d'utiliser le paradigme de programmation par tâches.

PaRSEC et StarPU offrent tous les deux une vision différentes du paradigme de tâches, PaRSEC semblant plus proche d'une réflexion algorithmique là où StarPU reste plus proche d'une logique

de développement traditionnelle. À travers nos implémentations avec StarPU et PaRSEC de la reconstruction des gradients par itération sur les cellules, nous établissons une comparaison des deux visions quant à leur compatibilité avec l'évolution d'un code industriel de simulation en mécanique des fluides. On compare ces implémentations avec le code existant en se concentrant sur l'importance des performances en distribué (il n'existe pas à notre connaissance d'étude comparative des performances des supports cités ci-avant vis-à-vis d'un code industriel et dans le cadre d'exécution distribuée, l'accent étant généralement mis sur l'exploitation de cartes accélératrices). On montre dans nos travaux que les performances obtenues en intra-nœud sont prometteuses tandis que celles en inter-nœuds sont plus compliquées à obtenir. Les différences significatives de complexités de développement constatées dans *Code_Saturne* entre PaRSEC et StarPU laissent à penser que le modèle de paradigme de tâche paramétrique est inadapté à l'évolution d'un code tel que *Code_Saturne*. À l'opposé l'utilisation de StarPU demande un effort peu significatif et avec un impact réduit sur *Code_Saturne* (de l'ordre de l'utilisation d'OpenMP, bien que de manière différente).

On apporte enfin les clés pour l'adaptation de la reconstruction des gradients par itération sur les faces par l'utilisation d'une coloration de graphe tout en utilisant des logiques de partitionnement adaptables en granularité déjà accessibles dans *Code_Saturne*.

Ces travaux posent la première pierre pour un passage de *Code_Saturne* vers l'utilisation d'un support exécutif à base de tâches. Nous identifions StarPU comme un support prometteur à une telle transition. Pour aller dans la continuité de ces travaux, nous identifions en premier lieu l'établissement d'une implémentation avec StarPU en parallélisme distribué comme primordiale. Enfin, la continuation de l'implémentation de la reconstruction des gradients par itération sur les faces permettra de toucher une plus grande partie du code de *Code_Saturne*. Une fois ces étapes réalisées, l'exploitation de plateformes massivement parallèles usant d'architectures GPGPUs ou Many-core via StarPU serait particulièrement intéressante, la transition de *Code_Saturne* vers ces architectures étant un des intérêts majeurs de l'utilisation d'un tel support exécutif. L'étude de l'équilibrage de charge doit aussi être continuellement fait afin de déterminer les bénéfices de StarPU sur ce dernier. L'implémentation actuelle sur la problématique de la reconstruction des gradients s'est concentrée sur le recouvrement potentiel des communications par les calculs de manière locales. Généraliser cette approche à plus haut niveau devrait pouvoir permettre de débloquer continuellement plus de parallélisme.

Références

[1] Emmanuel Agullo et al. “Are Static Schedules so Bad ? A Case Study on Cholesky Factorization”. In : *IEEE International Parallel & Distributed Processing Symposium (IPDPS 2016)*. Chicago, IL, United States : IEEE, mai 2016. URL : https://hal.inria.fr/hal-01223573.

[2] Emmanuel Agullo et al. *Faster, cheaper, better–a hybridization methodology to develop linear algebra software for GPUs*. 2010.

[3] Emmanuel Agullo et al. *Implementing multifrontal sparse solvers for multicore architectures with Sequential Task Flow runtime systems*. 2. 2016, p. 13.

[4] Emmanuel Agullo et al. “LU factorization for accelerator-based systems”. In : *Computer Systems and Applications (AICCSA), 2011 9th IEEE/ACS International Conference on*. IEEE. 2011, p. 217-224.

[5] Emmanuel Agullo et al. “QR factorization on a multicore node enhanced with multiple GPU accelerators”. In : *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*. IEEE. 2011, p. 932-943.

[6] Emmanuel Agullo et al. “Task-based Conjugate Gradient : from multi-GPU towards heterogeneous architectures”. Thèse de doct. Inria Bordeaux Sud-Ouest, 2016.

[7] James Ahrens, Berk Geveci et Charles Law. “Paraview : An end-user tool for large data visualization”. In : *The visualization handbook* 717 (2005).

[8] Hasan Metin Aktulga et al. “Optimizing sparse matrix-multiple vectors multiplication for nuclear configuration interaction calculations”. In : *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE. 2014, p. 1213-1222.

[9] Marco Annaratone. “MPPs, Amdahl’s law, and comparing computers”. In : *[Proceedings 1992] The Fourth Symposium on the Frontiers of Massively Parallel Computation*. IEEE. 1992, p. 465-470.

[10] Frédéric Archambeau, Namane Méchitoua et Marc Sakiz. “Code Saturne : A finite volume code for the computation of turbulent incompressible flows-Industrial applications”. In : *International Journal on Finite Volumes* (2004).

[11] Cecil G Armstrong et al. “Common themes in multi-block structured quad/hex mesh generation”. In : *Procedia Engineering* 124 (2015), p. 70-82.

[12] Cédric Augonnet et al. “StarPU : a unified platform for task scheduling on heterogeneous multicore architectures”. In : *Concurrency and Computation : Practice and Experience* 23.2 (2011), p. 187-198.

[13] Eduard Ayguadé et al. “An extension of the StarSs programming model for platforms with multiple GPUs”. In : *European Conference on Parallel Processing*. Springer. 2009, p. 851-862.

[14] Eduard Ayguadé et al. “The design of OpenMP tasks”. In : *IEEE Transactions on Parallel and Distributed Systems* 20.3 (2008), p. 404-418.

[15] Michael Edward Bauer. “Legion : Programming distributed heterogeneous architectures with logical regions”. Thèse de doct. Stanford University, 2014.

[16] Marsha J Berger et Antony Jameson. “Automatic adaptive grid refinement for the Euler equations”. In : *AIAA journal* 23.4 (1985), p. 561-568.

[17] Arthur J Bernstein. “Analysis of programs for parallel processing”. In : *IEEE Transactions on Electronic Computers* 5 (1966), p. 757-763.

[18] Robert D Blumofe et al. “Cilk : An efficient multithreaded runtime system”. In : *Journal of parallel and distributed computing* 37.1 (1996), p. 55-69.

[19] Lionel Boillot et al. “Task-based programming for seismic imaging : Preliminary results”. In : *2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC, CSS, ICESS)*. IEEE. 2014, p. 1259-1266.

[20] Jerome Bonelle. “Compatible Discrete Operator schemes on polyhedral meshes for elliptic and Stokes equations”. Thèse de doct. Université Paris-Est, 2014.

[21] G Bosilca et al. *Distibuted Dense Numerical Linear Algebra Algorithms on Massively Parallel Architectures : DPLASMA. University of Tennessee Computer Science Technical Report*. Rapp. tech. UT-CS-10-660, September 15, 2010.

[22] George Bosilca. “Dense linear algebra on distributed heterogeneous hardware with a symbolic dag approach”. In : *Scalable Computing and Communications : Theory and Practice* (2014).

[23] George Bosilca et al. “DAGuE : A generic distributed {DAG} engine for High Performance Computing”. In : *Parallel Computing* 38.1–2 (2012). Extensions for Next-Generation Parallel Programming Models, p. 37 -51. issn : 0167-8191. doi : http://dx.doi.org/10.1016/j.parco.2011.10.003. url : //www.sciencedirect.com/science/article/pii/S0167819111001347.

[24] George Bosilca et al. “Dense linear algebra on distributed heterogeneous hardware with a symbolic DAG approach”. In : *Scalable Computing and Communications : Theory and Practice* (2012).

[25] George Bosilca et al. “Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA”. In : *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*. IEEE. 2011, p. 1432-1441.

[26] George Bosilca et al. “Parsec : Exploiting heterogeneity to enhance scalability”. In : *Computing in Science & Engineering* 15.6 (2013), p. 36-45.

[27] Timothy Brecht. “On the importance of parallel application placement in NUMA multiprocessors”. In : *Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*. 1993, p. 1-18.

[28] Daniel Brélaz. “New methods to color the vertices of a graph”. In : *Communications of the ACM* 22.4 (1979), p. 251-256.

[29] François Broquedis et al. “ForestGOMP : An Efficient OpenMP Environment for NUMA Architectures”. In : *International Journal of Parallel Programming* 38.5 (2010), p. 418-439. issn : 1573-7640. doi : 10.1007/s10766-010-0136-3. url : http://dx.doi.org/10.1007/s10766-010-0136-3.

[31] Alfredo Buttari et al. “Parallel tiled QR factorization for multicore architectures”. In : *Concurrency and Computation : Practice and Experience* 20.13 (2008), p. 1573-1590.

[32] Jean Marie Couteyen Carpaye. “Contribution à la parallélisation et au passage à l’échelle du code FLUSEPA”. Thèse de doct. 2016.

[33] Jean Marie Couteyen Carpaye, Jean Roman et Pierre Brenner. “Design and analysis of a task-based parallelization over a runtime system of an explicit finite-volume CFD code with adaptive time stepping”. In : *Journal of computational science* 28 (2018), p. 439-454.

[34] Bradford L Chamberlain, David Callahan et Hans P Zima. “Parallel programmability and the chapel language”. In : *The International Journal of High Performance Computing Applications* 21.3 (2007), p. 291-312.

[35] Philippe Charles et al. “X10 : an object-oriented approach to non-uniform cluster computing”. In : *Acm Sigplan Notices*. T. 40. 10. ACM. 2005, p. 519-538.

[36] Cédric Chevalier et François Pellegrini. “PT-Scotch : A tool for efficient parallel graph ordering”. In : *Parallel computing* 34.6-8 (2008), p. 318-331.

[37] Cristian Coarfa et al. “An evaluation of global address space languages : co-array fortran and unified parallel C”. In : *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM. 2005, p. 36-47.

[38] Michel Cosnard et Michel Loi. “Automatic task graph generation techniques”. In : *System Sciences, 1995. Proceedings of the Twenty-Eighth Hawaii International Conference on*. T. 2. IEEE. 1995, p. 113-122.

[39] G Crimi et al. “Early experience on porting and running a Lattice Boltzmann code on the Xeon-Phi co-processor”. In : *Procedia Computer Science* 18 (2013), p. 551-560.

[40] Leonardo Dagum et Ramesh Menon. “OpenMP : an industry standard API for shared-memory programming”. In : *IEEE computational science and engineering* 5.1 (1998), p. 46-55.

[41] Anthony Danalis et al. “PaRSEC in Practice : Optimizing a legacy Chemistry application through distributed task-based execution”. In : *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*. IEEE. 2015, p. 304-313.

[42] Anthony Danalis et al. “PTG : an abstraction for unhindered parallelism”. In : *Proceedings of the Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*. IEEE Press. 2014, p. 21-30.

[43] SD Daxini et JM Prajapati. “A review on recent contribution of meshfree methods to structure and fracture mechanics applications”. In : *The Scientific World Journal* 2014 (2014).

[44] Nicolas Denoyelle et al. “Modeling Non-Uniform Memory Access on Large Compute Nodes with the Cache-Aware Roofline Model”. In : *IEEE Transactions on Parallel and Distributed Systems* (2018).

[45] Mehmet Deveci et al. “Multi-jagged : A scalable parallel spatial partitioning algorithm”. In : *IEEE Transactions on Parallel and Distributed Systems* 27.3 (2016), p. 803-817.

[46] Iain Duff, Jonathan Hogg et Florent Lopez. “Experiments with sparse Cholesky using a sequential task-flow implementation”. In : *Numerical Algebra, Control & Optimization* 8.2 (2018), p. 237-260.

[47] Iain Duff et Florent Lopez. “Experiments with sparse Cholesky using a parametrized task graph implementation”. In : *International Conference on Parallel Processing and Applied Mathematics*. Springer. 2017, p. 197-206.

[48] Alejandro Duran et al. “Ompss : a proposal for programming heterogeneous multi-core architectures”. In : *Parallel Processing Letters* 21.02 (2011), p. 173-193.

[49] Eduardo F D’Azevedo, Mark R Fahey et Richard T Mills. “Vectorized sparse matrix multiply for compressed row storage format”. In : *International Conference on Computational Science*. Springer. 2005, p. 99-106.

[50] Joel H Ferziger et Milovan Peric. *Computational methods for fluid dynamics*. Springer Science & Business Media, 2012.

[52] Charles M Fiduccia et Robert M Mattheyses. “A linear-time heuristic for improving network partitions”. In : *19th Design Automation Conference*. IEEE. 1982, p. 175-181.

[53] Y. Fournier et al. “Optimizing Code Saturne computations on Petascale systems”. In : *Computers & Fluids* 45.1 (2011). 22nd International Conference on Parallel Computational Fluid Dynamics (ParCFD 2010)ParCFD, p. 103 -108. issn : 0045-7930. doi : http://dx.doi.org/10.1016/j.compfluid.2011.01.028. url : http://www.sciencedirect.com/science/article/pii/S0045793011000351.

[54] Edgar Gabriel et al. “Open MPI : Goals, concept, and design of a next generation MPI implementation”. In : *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer. 2004, p. 97-104.

[55] Thierry Gautier, Xavier Besseron et Laurent Pigeon. “Kaapi : A thread scheduling runtime system for data flow computations on cluster of multi-processors”. In : *Proceedings of the 2007 international workshop on Parallel symbolic computation*. ACM. 2007, p. 15-23.

[56] Thierry Gautier et al. “X-Kaapi : a multi paradigm runtime for multicore architectures”. In : *Parallel Processing (ICPP), 2013 42nd International Conference on*. IEEE. 2013, p. 728-735.

[57] Apostolos Gerasoulis et Tao Yang. “A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors”. In : *Journal of Parallel and Distributed Computing* 16.4 (1992), p. 276-291.

[58] William Gropp et al. *Using MPI : portable parallel programming with the message-passing interface*. T. 1. MIT press, 1999.

[59] William D Gropp et al. “High-performance parallel implicit CFD”. In : *Parallel Computing* 27.4 (2001), p. 337-362.

[60] Liu Gui-rong. *Smoothed particle hydrodynamics : a meshfree particle method*. World Scientific, 2003.

[62] Michael A Heroux et al. “Improving performance via mini-applications”. In : *Sandia National Laboratories, Tech. Rep. SAND2009-5574* 3 (2009).

[63] Jonathan D Hogg, John K Reid et Jennifer A Scott. “Design of a multicore sparse Cholesky factorization using DAGs”. In : *SIAM Journal on Scientific Computing* 32.6 (2010), p. 3627-3649.

[64] Emmanuel Jeannot et Guillaume Mercier. “Near-optimal placement of MPI processes on hierarchical NUMA architectures”. In : *European Conference on Parallel Processing*. Springer. 2010, p. 199-210.

[65] Laxmikant V Kale et Sanjeev Krishnan. “CHARM++ : a portable concurrent object oriented system based on C++”. In : *ACM Sigplan Notices*. T. 28. 10. ACM. 1993, p. 91-108.

[66] George Karypis. “METIS and ParMETIS”. In : *Encyclopedia of parallel computing*. Springer, 2011, p. 1117-1124.

[67] B. W. Kernighan et S. Lin. “An efficient heuristic procedure for partitioning graphs”. In : *The Bell System Technical Journal* 49.2 (1970), p. 291-307. issn : 0005-8580. doi : 10.1002/j.1538-7305.1970.tb01770.x.

[68] Dimitri Komatitsch, David Michéa et Gordon Erlebacher. “Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA”. In : *Journal of Parallel and Distributed Computing* 69.5 (2009), p. 451-460.

[71] Christoph Lameter. “Numa (non-uniform memory access) : An overview”. In : *Queue* 11.7 (2013), p. 40.

[72] Charles E Leiserson. “The Cilk++ concurrency platform”. In : *The Journal of Supercomputing* 51.3 (2010), p. 244-257.

[73] Wai-Hung Liu et Andrew H Sherman. “Comparative analysis of the Cuthill–McKee and the reverse Cuthill–McKee ordering algorithms for sparse matrices”. In : *SIAM Journal on Numerical Analysis* 13.2 (1976), p. 198-213.

[74] Florent Lopez. “Task-based multifrontal QR solver for heterogeneous architectures”. Thèse de doct. Université de Toulouse, Université Toulouse III-Paul Sabatier, 2015.

[76] Zoltan Majo et Thomas R Gross. "(Mis) understanding the NUMA memory system performance of multithreaded workloads". In : *Workload Characterization (IISWC), 2013 IEEE International Symposium on*. IEEE. 2013, p. 11-22.

[77] Preeti Malakar et al. "Benchmarking Machine Learning Methods for Performance Modeling of Scientific Applications". In : *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE. 2018, p. 33-44.

[78] Hasnain A Mandviwala, Umakishore Ramachandran et Kathleen Knobe. "Capsules : Expressing composable computations in a parallel programming model". In : *International Workshop on Languages and Compilers for Parallel Computing*. Springer. 2007, p. 276-291.

[79] Salli Moustafa. "Massively Parallel Cartesian Discrete Ordinates Method for Neutron Transport Simulation". Thèse de doct. Université de Bordeaux, 2015.

[80] Salli Moustafa et al. "3D cartesian transport sweep for massively parallel architectures with PARSEC". In : *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. IEEE. 2015, p. 581-590.

[81] Dorit Nuzman, Ira Rosen et Ayal Zaks. "Auto-vectorization of interleaved data for SIMD". In : *ACM SIGPLAN Notices* 41.6 (2006), p. 132-143.

[82] Jeeva Paudel, Olivier Tardieu et José Nelson Amaral. "On the merits of distributed work-stealing on selective locality-aware tasks". In : *Parallel Processing (ICPP), 2013 42nd International Conference on*. IEEE. 2013, p. 100-109.

[83] M Perić. "Analysis of pressure-velocity coupling on nonorthogonal grids". In : *Numerical Heat Transfer* 17.1 (1990), p. 63-82.

[84] Gregorio Quintana-Orti et al. "Scheduling of QR factorization algorithms on SMP and multi-core architectures". In : *Parallel, Distributed and Network-Based Processing, 2008. PDP 2008. 16th Euromicro Conference on*. IEEE. 2008, p. 301-310.

[86] I Z Reguly et al. "Vectorizing unstructured mesh computations for many-core architectures". In : *Concurrency and Computation : Practice and Experience* 28.2 (2016), p. 557-577.

[87] James Reinders. *Intel threading building blocks : outfitting C++ for multi-core processor parallelism*. " O'Reilly Media, Inc.", 2007.

[88] James Reinders, Jim Jeffers et Rob Farber. *High Performance Parallelism Pearls*. 2015, p. 129-142. isbn : 9780128021187. doi : 10.1016/B978-0-12-802118-7.00007-8. arXiv : arXiv:1011.1669v3. url : http://www.sciencedirect.com/science/article/pii/B9780128021187000078.

[89] Corentin Rossignon. "Un modèle de programmation à grain fin pour la parallélisation de solveurs linéaires creux". Thèse de doct. 2015.

[90] Corentin Rossignon et al. "A NUMA-aware fine grain parallelization framework for multi-core architecture". In : *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*. IEEE. 2013, p. 1381-1390.

[91] Yousef Saad. *Iterative methods for sparse linear systems*. SIAM, 2003.

[92] Rahul S. Sampath et al. "Dendro : Parallel Algorithms for Multigrid and AMR Methods on 2 :1 Balanced Octrees". In : *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. SC '08. Austin, Texas : IEEE Press, 2008, 18 :1-18 :12. isbn : 978-1-4244-2835-9. url : http://dl.acm.org/citation.cfm?id=1413370.1413389.

[94] Will J Schroeder, Bill Lorensen et Ken Martin. *The visualization toolkit : an object-oriented approach to 3D graphics*. Kitware, 2004.

[95] Zhi Shang. "Impact of mesh partitioning methods in CFD for large scale parallel computing". In : *Computers & Fluids* 103 (2014), p. 1-5.

[96] Christopher Stone et al. “GPGPU parallel algorithms for structured-grid CFD codes”. In : *20th AIAA Computational Fluid Dynamics Conference*. 2011, p. 3221.

[97] John E Stone, David Gohara et Guochun Shi. “OpenCL : A parallel programming standard for heterogeneous computing systems”. In : *Computing in science & engineering* 12.3 (2010), p. 66.

[98] Xinmin Tian et al. “Effective simd vectorization for intel xeon phi coprocessors”. In : *Scientific Programming* 2015 (2015), p. 1.

[100] John F Wendt. *Computational fluid dynamics : an introduction*. Springer Science & Business Media, 2008.

[101] Samuel Williams, Andrew Waterman et David Patterson. “Roofline : an insightful visual performance model for multicore architectures”. In : *Communications of the ACM* 52.4 (2009), p. 65-76.

[103] Rengan Xu, Sunita Chandrasekaran et Barbara Chapman. “Exploring programming multi-GPUs using OpenMP and OpenACC-based hybrid model”. In : *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*. IEEE. 2013, p. 1169-1176.

[105] Ting Ye et al. “Smoothed particle hydrodynamics (SPH) for complex fluid flows : Recent developments in methodology and applications”. In : *Physics of Fluids* 31.1 (2019), p. 011301.

Appendices

I Extraits de codes PaRSEC/*Code_Saturne*

```
1  int
2  _parsec_task_caller({...})
3  {
4     parsec_context_t* parsec = cs_glob_parsec_context;
5     int rank, world;
6     parsec_ddesc_t taskdist;
7     cs_halo_desc_t halo_desc;
8
9     parsec___parsec_gradients_handle_t *handle;
10
11    parsec_ddesc_init(&taskdist, world, rank);
12    taskdist.rank_of = rank_of;
13    taskdist.vpid_of = vpid_of;
14    taskdist.data_key = data_key;
15
16    cs_halo_desc_init(&halo_desc, world, rank, cs_glob_mesh, pvar);
17    halo_desc.super.rank_of = comm_rank_of;
18    halo_desc.super.vpid_of = vpid_of;
19
20    handle = parsec___parsec_gradients_new(&taskdist, &halo_desc, rank, world,
21                                           inc, coefbp, coefap, extrap, grad,
22                                           pvar, cs_glob_mesh_quantities,
23                                           cs_glob_mesh, cs_glob_mesh_adj);
24
25    parsec_arena_construct(
26       handle->arenas[PARSEC___parsec_gradients_DEFAULT_ARENA],
27       sizeof(cs_real_t), PARSEC_ARENA_ALIGNMENT_SSE,
28       parsec_datatype_double_t);
29
30    parsec_enqueue( parsec, (parsec_handle_t*)handle );
31    parsec_context_start(parsec);
32    parsec_context_wait(parsec);
33
34    parsec_handle_free((parsec_handle_t*)handle);
35
36    parsec_ddesc_destroy(&taskdist);
37    cs_halo_desc_destroy(&halo_desc);
38    return 0;
39 }
40
```

FIGURE 15: Soumission des tâches avec PaRSEC

```
1  static void
2  _lsq_scalar_gradient(const cs_mesh_t *m,
3                       cs_mesh_quantities_t *fvq,
4                       cs_halo_type_t halo_type,
5                       int idimtr,
6                       int hyd_p_flag,
7                       int w_stride,
8                       cs_real_t inc,
9                       double extrap,
10                      const cs_real_3_t f_ext[],
11                      const cs_real_t coefap[],
12                      const cs_real_t coefbp[],
13                      const cs_real_t pvar[],
14                      cs_real_t c_weight[],
15                      cs_real_3_t *restrict grad)
16 {
17   cs_halo_set_index(m->halo);
18   _parsec_task_caller(pvar, idimtr, grad, extrap, coefap, coefbp, inc);
19   _sync_scalar_gradient_halo(m, CS_HALO_STANDARD, idimtr, grad);
20 }
```

FIGURE 16: Appel de la soumission des tâches avec PaRSEC depuis *Code_Saturne*

```
1  void
2  cs_halo_desc_init( cs_halo_desc_t *hdesc, int nodes, int myrank, cs_mesh_t *mesh,
3                     const cs_real_t *var)
4  {
5    int rank_id;
6    cs_real_t *ptr;
7    size_t size;
8
9    parsec_ddesc_t *o = (parsec_ddesc_t*)hdesc;
10   parsec_ddesc_init( o, nodes, myrank );
11
12   hdesc->nb_local_halos = mesh->halo->n_c_domains;
13   o->data_key = cs_halo_data_key;
14   o->data_of = cs_halo_data_of;
15
16   hdesc->data_map = (parsec_data_t**) calloc(cs_glob_n_ranks,
17                                              sizeof(parsec_data_t*));
18
19   for (rank_id = 0; rank_id < hdesc->nb_local_halos; ++rank_id) {
20
21     if ( mesh->halo->c_domain_rank[rank_id] != myrank)
22     {
23       size = mesh->halo->index[2*rank_id+1] - mesh->halo->index[2*rank_id];
24       if (size){
25       cs_lnum_t id = mesh->halo->c_domain_rank[rank_id];
26       ptr = (cs_real_t*) var + mesh->halo->n_local_elts
27                              + mesh->halo->index[2*rank_id];
28
29       parsec_data_t *data = parsec_data_create( hdesc->data_map + id,
30                                                 o, id, ptr, sizeof(cs_real_t) * size);
31       }
32     }
33   } // end of loop on ranks
34 }
```

FIGURE 17: Initialisation du descripteur pour chaque tâche de réception des halos avec PaRSEC.