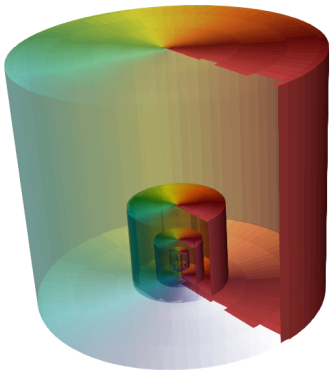
Décomposition de domaines et ordonnancement de tâches pour la simulation en mécanique des fluides

Alice Lasserre

Rapport de Stage

Décomposition de domaines et ordonnancement de tâches pour la simulation en mécanique des fluides.



Étudiant

Alice Lasserre
Université Sciences et Technologies de Bordeaux

Maîtres de stage

Raymond Namyst
Inria Bordeaux Sud-Ouest

Abdou Guermouche
Inria Bordeaux Sud-Ouest

Jean-Marie Couteyen-Carpaye
Airbus

2021 - 2022

Table des matières

Remerciements

Je souhaite remercier tout d'abord mes trois encadrants de stage M. Jean-Marie Couteyen-Carpaye, M. Raymond Namyst et M. Abdou Guermouche pour m'avoir donnée l'opportunité de travailler sur ce sujet particulièrement intéressant. Je tiens aussi à témoigner ma reconnaissance pour la qualité de leur encadrement, leur disponibilité à tout instant ainsi que l'aide apportée à la rédaction de ce rapport.

Je remercie également tous les membres de l'équipe STORM pour leur acceuil.

Enfin, je remercie mes parents et mon frère pour leur soutien indéfectible.

Première partie

Introduction

Les domaines de la recherche et de l'industrie, en particulier appliqués aux domaines de l'aérospatiale et de l'aéronautique, nécessitent très souvent la réalisation d'expériences, parfois à échelle réduite. Celles-ci se trouvent être très coûteuses et complexes à mettre en place, voire parfois nuisibles et dangereuses pour l'homme. *ArianeGroup*, pour concevoir ses lanceurs de satellites et vérifier certaines de leurs propriétés, utilise parfois des maquettes en soufflerie. Cependant, ces expériences ne permettent pas de couvrir l'ensemble des paramètres qui ont une influence sur la trajectoire et sur la structure d'un lanceur lors d'un vol réel. *Airbus* a aussi recours à des essais en vol pour valider les performances de ces avions. Ces essais sont très coûteux et réduire leur nombre est un enjeu important. C'est pourquoi les ingénieurs utilisent de plus en plus des simulations numériques sur ordinateur, qui s'efforcent de calculer le plus précisément et le plus fidèlement possible les phénomènes physiques de manière virtuelle. L'utilisation de ces simulations s'intégre en outre dans une politique environnementale d'*Airbus* afin de passer à des énergies plus propres et réduire l'impact sur l'environnement. La simulation permet ainsi d'évaluer les nuisances acoustiques des avions afin de les réduire et de préparer le passage à l'hydrogène afin de décarboner le secteur aérien. De plus, concevoir des codes de simulation très efficaces permet une réduction de l'empreinte énergétique de ces expériences virtuelles.

Afin de retranscrire au mieux la réalité et de conserver un temps d'exécution raisonnable (de l'ordre de quelques jours de calcul), ces simulations doivent nécessairement s'exécuter sur des ordinateurs très puissants. Ces machines, appelées supercalculateurs [1], ont la particularité de posséder une architecture « parallèle » où plusieurs milliers (voire millions) de cœurs exécutent des flots d'instructions distincts. Pour en tirer parti, le code des applications doit être décomposé en blocs d'instructions dont certains peuvent s'exécuter en même temps.

Avec de telles puissances de calcul, on peut rêver de simulations numériques s'approchant au plus près de la réalité. Néanmoins, il devient de plus en plus difficile d'écrire les codes de simulation à destination de ces architectures toujours plus complexes. En effet, à la difficulté d'extraire le maximum de parallélisme au sein d'une application s'ajoutent celle de générer un code efficace ainsi que celle liée à l'orchestration de l'ensemble des traitements de manière optimale (ordonnancement). Le premier point nécessite de choisir une technique de parallèlisation appropriée à l'application, c'est-à-dire exprimer les calculs sous une forme qui génère le plus de parallèlisme possible. Le second requiert l'utilisation d'une stratégie d'ordonnancement et d'équilibrage des calculs qui soit capable de minimiser les temps d'inactivité des cœurs. Enfin, certains calculs vont dépendre d'autres calculs, parfois distants, et vont donc générer des mouvements de données. Ces derniers, très coûteux pour la simulation, nécéssitent d'être fortement minimisés.

Par conséquent, le fossé grandit entre la puissance théorique (aussi appelée puissance crête) annoncée par les constructeurs et les performances atteintes en pratique. Aujourd'hui, il devient rare de pouvoir dépasser 60% de la puissance crête ! En effet, adapter ces applications utilisées dans la recherche et l'industrie depuis une dizaine d'années sur ces machines très récentes n'est pas une tâche aisée. Des efforts importants en algorithmique parallèle et en optimisation de code sont donc nécessaires afin d'exploiter le plein potentiel de ces architectures.

1. Ces machines atteignent aujourd'hui des puissances de l'ordre 10^{18} flop/s (floating point operations per second), c'est-à-dire un milliard de milliards d'opérations en virgule flottante par seconde.

ArianeGroup developpe depuis plus de trente ans le code de simulation numérique FLUSEPA qui permet de modéliser des phénomènes instationnaires (i.e. qui varient au cours du temps) en mécanique des fluides (CFD). Des phénomènes tels que la propagation d'ondes de souffle au décollage d'une fusée ou des séparations d'étages de lanceurs peuvent ainsi être simulés. Le simulateur FLUSEPA représente les objets en mouvement et les fluides associés à l'aide d'une géométrie qui les modélise dans l'espace. Le code effectue ensuite une discrétisation spatiale sous la forme d'un maillage, c'est-à-dire un découpage de la géométrie en petits volumes au sein desquels les valeurs physiques que l'on calcule sont supposées uniformes. Cela constitue un compromis entre précision acceptable et temps de calcul global. Puisque l'évolution de ces valeurs physiques doit être calculée au cours du temps, une seconde approximation est réalisée en discrétisant le temps : les valeurs sont calculées toutes les Δt secondes, que l'on appelle le « pas de temps » de la simulation.

Le code de simulation est donc un code itératif qui recalcule toutes les valeurs associées aux mailles à chaque pas de temps. La parallélisation s'appuie sur le découpage spatial du domaine, en s'efforçant de calculer simultanément de nombreuses valeurs associées à des mailles distinctes. Évidemment, toutes les valeurs ne peuvent être calculées indépendamment les unes des autres car il existe des dépendances entre les calculs qui proviennent de l'influence des mailles voisines.

Ainsi, de nombreuses synchronisations et communications sont nécessaires entre les cœurs pour assurer que les calculs distribués sont corrects, ralentissant fortement la simulation numérique. Si l'impact de celles-ci peut facilement être minimisé sur une architecture à mémoire commune, il n'en va pas de même sur un supercalculateur où les synchronisations entre cœurs doivent passer par un complexe réseau d'interconnexion externe. Afin de minimiser ces dernières, un modèle de programmation à base de tâches a été mis en place dans FLUSEPA. Le travail généré par l'application est représenté par un graphe orienté où les sommets correspondent aux tâches et les arêtes aux dépendances.

Bien que sensiblement réduite dans une approche par tâches, la présence de ces communications et synchronisations est toujours observable dans l'exécution du code en particulier sous la forme de grandes périodes d'inactivité constatées sur les cœurs. Le phénomène est accru lorsque le nombre de ressources est trop élevée ou le travail généré par l'application est trop faible. Malheureusement, il n'est pas clair que ces périodes d'oisiveté soient exclusivement causées par la présence des communications et des dépendances.

La technique de parallèlisation étant fixe, ce stage se place dans l'optique d'identifier les paramètres influant sur la présence de ces périodes d'oisiveté constatées sur les cœurs ne partageant pas de mémoire commune et ainsi implémenter des solutions adaptées. Je me suis donc intéressée à la résolution de ces problèmes de performance dans le cadre du stage de Master 2 que j'ai effectué au centre de recherche Inria Bordeaux Sud-Ouest au sein de l'équipe STORM, sous la direction de Raymond Namyst, Abdou Guermouche et Jean-Marie Couteyen-Carpaye (*Airbus*). Ma démarche scientifique s'est organisée comme suit.

J'ai commencé par étendre un simulateur développé chez *Airbus* permettant de pouvoir reproduire rapidement et aisément de nombreuses conditions expérimentales sans nécessiter des jours de calcul littéralement. J'ai surtout développé une suite d'outils de visualisation et d'analyse des traces d'exécution permettant d'observer finement ce qui s'est passé sur chaque cœur, de détecter les problèmes de famine et de quantifier les déséquilibres de charge notamment. Surtout, cela m'a permis de pouvoir "scripter" mes expériences afin de faire varier automatiquement certains paramètres et générer aisément des courbes comparatives.

À partir d'un cas test de grande taille fourni par *Airbus*, j'ai pu reproduire les problèmes de

performance dans le cadre du simulateur. Cela a permis de déterminer dans quelles configurations ces derniers impactent le plus négativement le temps d'exécution global.

Ma première contribution a été de mieux guider l'ordre d'exécution des tâches en leur attribuant des priorités de façon à, intuitivement, débloquer plus rapidement de nouvelles tâches "prêtes" sur des nœuds distants. L'idée est de veiller à conserver un niveau suffisant de parallélisme tandis que les échanges de données entre cœurs s'effectuent en tâche de fond.

Bien qu'améliorant les performances pour de nombreuses configurations, les traces d'exécution ont montré que de régulières périodes de pénurie de tâches demeuraient. Aussi, ma seconde contribution a été de remettre en question la décomposition de domaine elle-même, c'est-à-dire la manière dont le maillage d'entrée est découpé et réparti entre les différents nœuds du calculateur.

J'ai donc effectué quelques partitionnements de manière ad-hoc sur le maillage original mais aussi sur un cas test particulier, en suivant une intuition sur l'importance d'avoir des graphes de tâches de largeur homogène sur chaque nœud.

Les évaluations menées grâce au simulateur sur de nombreuses configurations montrent que la combinaison de ces approches est payante : dans certains cas, l'exécution (théorique) a été réduite de 60% !

Deuxième partie

Simulation appliquée à la mécanique des fluides

Développé depuis une trentaine d'année par *ArianeGroup* dont *Airbus* possède une licence d'utilisation avec accès aux sources, le code de simulation numérique FLUSEPA correspond à un code de production de plus de 160.000 lignes, majoritairement écrit en langages C et Fortran. Ce code de mécanique des fluides permet de visualiser des phénomènes 3D instationnaires avec topologie variable, notamment le bruit généré par les hélices d'un avion ou, comme observé sur la figure 1, la pression que subissent les propulseurs d'appoints d'Ariane 5 lors de leur séparation avec l'étage principal.

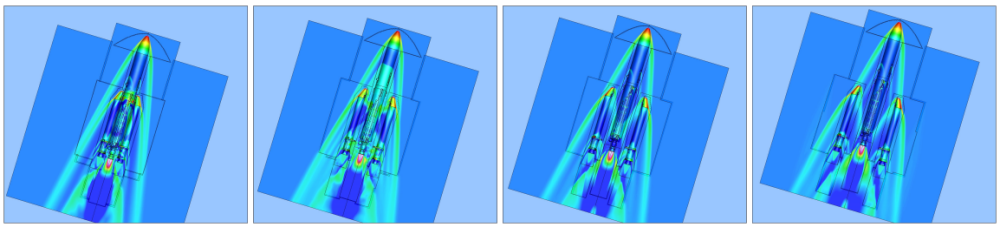


FIGURE 1 – Séparation des lanceurs d'Ariane 5 après l'allumage de la fusée, colorée en fonction de la pression.[13]

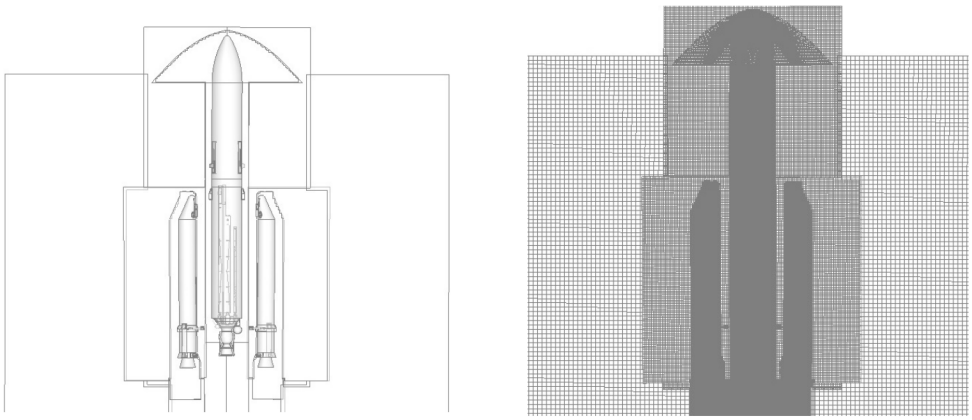
1 Le code de simulation FLUSEPA

FLUSEPA utilise les équations aux dérivées partielles de Navier-Stokes pour décrire le mouvement de fluide. Ces équations nécessitent d'être discrétisées en espace et en temps, c'est-à-dire seulement calculées pour un ensemble discret de points de l'espace, et uniquement à certains moments de l'échelle temporelle considérée.

1.1 Discrétisation spatiale

La représentation numérique des objets en mouvement et des fluides correspondants est rendue possible grâce à une discrétisation spatiale. Celle-ci se base sur une méthode dite des volumes finis. Concrètement, à partir d'une géométrie qui modélise les objets et flux comme sur la figure 2a, un découpage en petits volumes est réalisé au sein desquels les valeurs physiques que l'on calcule sont supposées uniformes. Cela constitue un compromis entre précision acceptable et temps de calcul global. Ce découpage, dont un exemple est présenté figure 2b, correspond à la création d'un maillage composé de cellules et de faces où les valeurs physiques telles que la pression ou la température sont calculées pour chaque cellule et les flux sont évalués entre les faces des cellules. Le calcul des flux nécessite donc les valeurs des cellules voisines et inversement, le calcul des valeurs nécessite ceux des flux.

En conséquence, la portion du code qui calcule la simulation numérique à chaque itération de calcul, appelée solveur aérodynamique, n'utilise à cet effet que des objets « cellules » et « faces ». Notons qu'une maille correspond à une cellule et ses faces.



(a) Géométrie (b) Maillages

FIGURE 2 – Représentation numérique des objets et flux.

1.2 Discrétisation temporelle

Dans de nombreuses simulations numériques, on souhaite obtenir une solution stationnaire et on utilise un schéma d'intégration temporel implicite. Un pas de temps global est fixé, par exemple trois secondes, et l'ensemble des mailles de la simulation est alors calculé à chaque multiple de trois secondes dans le référentiel temporel de la simulation. Dans ce cas, la méthode explicite n'est pas compétitive car elle n'est stable qu'avec de faibles pas de temps. Intuitivement, pour que la méthode explicite soit stable, les phénomènes ne doivent pas se déplacer de plus d'une maille par itération. Contrairement à une intégration explicite en temps, l'intégration implicite permet d'utiliser des pas de temps très élevés mais ne permet pas de suivre des phénomènes instationnaires, sauf si on utilise de faibles pas de temps. Cependant, la méthode implicite nécessite la résolution d'un système à chaque pas de temps et si celui-ci devient trop faible, elle est beaucoup trop coûteuse comparée à une méthode explicite.

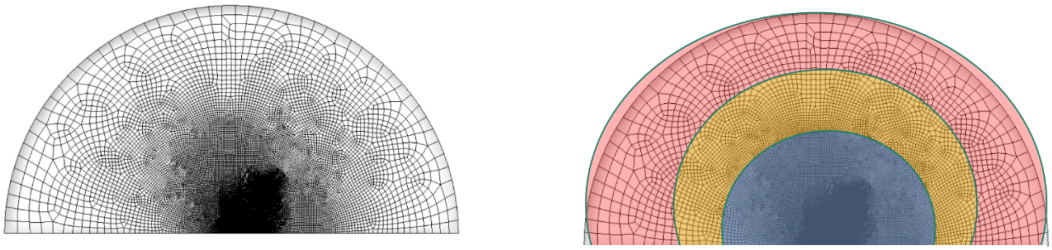
Dans le cas de FLUSEPA, les phénomènes simulés sont instationnaires, on veut calculer les évolutions du phénomène étudié au cours du temps. Actualiser la simulation nécéssite d'effectuer des calculs transitoires à des intervalles de temps bornés pour mettre à jour les valeurs physiques. Le solveur aérodynamique intègre donc un schéma d'intégration temporel adaptatif, basé sur une intégration en temps explicite.

1.2.1 Description de l'intégration temporelle adaptative

Lorsque les phénomènes à simuler sont instationnaires, il est nécessaire d'utiliser des maillages dont la résolution n'est pas uniforme. Les mailles ont alors des dimensions et fréquences de calculs très variables entre elles. La figure 3a présente un maillage caractéristique des maillages non-uniformes. Au centre de celui-ci, une concentration importante de mailles extrêmement fines est observée. Cette densité est dû au fait que les mailles sont plus faibles en dimension au cœur de l'action afin de conserver le plus de précision possible. Ces mailles très fines, situées par exemple au pied des réacteurs, évoluent très vite et nécéssitent une fréquence de calcul importante afin de ne pas avoir une approximation trop grossière (ou fausse) de la réalité. A contrario, les mailles deviennent

plus larges à mesure que l'on s'éloigne du phénomène. Ces dernières n'ont pas besoin d'être calculées avec le plus petit pas de temps possible puisque les grandeurs évoluent très lentement à ces endroits de l'espace.

Fixer un pas de temps global, forcément très faible, pour l'intégralité du maillage entraînerait une surcharge de calcul inutile. C'est là qu'intervient la *méthode d'intégration temporelle adaptative* qui permet de calculer les mailles à des fréquences différentes grâce à l'utilisation d'un pas de temps local propre à chaque maille.



(a) Vue de haut d'un maillage non-uniforme où les mailles n'ont pas la même dimension ni fréquence de calcul

(b) Maillage colorié en fonction d'une répartition théorique des mailles entre différentes classes de niveau temporel

FIGURE 3 – Classification des mailles sur un maillage non-uniforme.

1.2.2 Mise en oeuvre de l'intégration temporelle adaptative

La méthode d'intégration temporelle **explicite** réalise donc la seconde discrétisation de FLUSEPA grâce à la méthode de Heun. Celle-ci permet de résoudre des équations d'ordre 2 en se basant sur un schéma prédicteur-correcteur. Concrètement, la discrétisation temporelle consiste à déterminer à quelle fréquence chaque maille doit être recalculée.

Dans la simulation, une itération est définie comme la quantité de temps nécéssaire à ce que l'ensemble du maillage soit calculé. À la fin de chaque itération, toutes les mailles doivent avoir atteint le même temps physique. Pour permettre le calcul de mailles ayant des fréquences différentes, la méthode d'integration temporelle utilisée découpe les itérations en sous-itérations. Chaque itération est séparée en 2^θ sous-itérations où θ est le niveau temporel le plus élevé parmi les mailles et donc la fréquence la plus faible dans la simulation.

La première étape consiste à classer les mailles dans des groupes appelés *classes de niveau temporel*, en fonction de la fréquence à laquelle elles seront calculées durant la simulation. Pour chaque maille, cette distribution est calculée en fonction de sa fréquence de calcul minimale. Les mailles nécessitant la fréquence de calcul la plus faible possible, notée δ_t, appartiennent à la classe de niveau temporel $\tau = 0$. Chaque maille se voit attribuer un niveau temporel τ, traduit par l'appartenance à une classe. Les mailles de la classe de niveau temporel $\tau = 0$ vont devoir être recalculées à chaque sous-itération tandis que celles de la classe $\tau = 1$ seulement une sous-itération sur deux et celles de la classe $\tau = 2$, une sous-itération sur 3 et ainsi de suite. Notons que ce sont donc les mailles de niveaux temporels faibles qui vont requérir le plus de calculs.

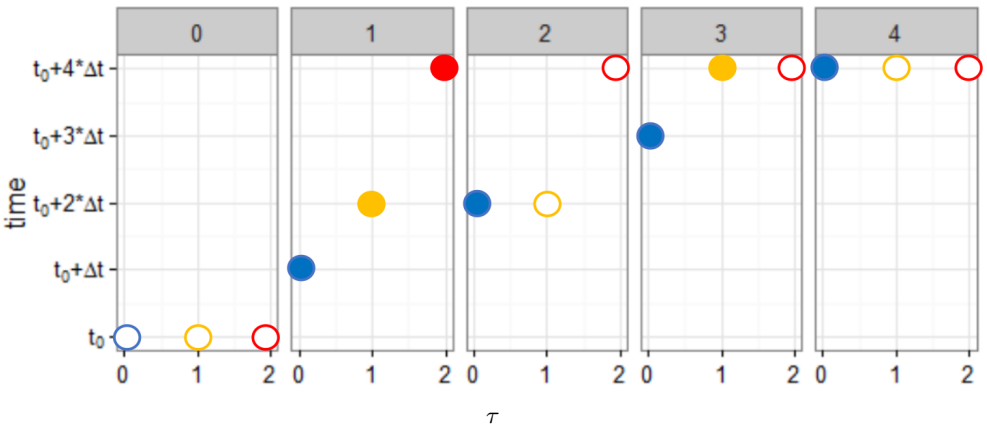


FIGURE 4 – Représentation de la méthode d'intégration temporelle adaptative où les classes temporelles sont calculées (cercle plein) ou non (cercle vide) à chaque sous-itération ($\theta = 2$, 4 sous-itérations). Les boîtes de 1 à 4 représentent les sous-itérations. La boîte 0 représente simplement l'état initial.

La figure 4 présente un fonctionnement schématique de l'intégration temporelle adaptative explicite décrite précédemment. Ici, le niveau temporel maximum nécéssaire au maillage est $\theta = 2$, l'itération est donc découpée en $2^2 = 4$ sous-itérations. Les mailles sont réparties en 3 niveaux temporels (0, 1 et 2) représentés en abcisse au sein de chaque sous-itération. Les cercles indiquent si les mailles appartenant à un niveau temporel τ sont calculées (cercle plein) ou non (cercle vide) pendant une sous-itération donnée. Ainsi, lors de la première sous-itération, l'ensemble des mailles sont calculées, comme l'indiquent les trois cercles pleins. Notons qu'à ce stade, le cercle rouge en haut indique que les mailles de niveau temporel 2 ont déjà atteint leur état final : elles n'auront plus besoin d'être calculées durant le reste de l'itération. Le cercle jaune, lui, est plein à la première sous-itération et à la troisième, signifiant que les mailles de niveau temporel 1 ne sont calculées qu'une fois sur deux. Enfin, le niveau temporel 0 affiche un cercle plein à chaque sous-itération : les mailles correspondantes nécessitent d'être recalculées à chaque sous-itération. Lors de la dernière sous-itération, toutes les mailles atteignent le même indice de temps.

Notons que la répartition des mailles en niveau temporel obéit à une contrainte supplémentaire issue de la physique : le niveau temporel de deux mailles voisines ne peut différer que de 1. Par ailleurs, l'ordre de calcul donné en figure 4 n'est pas anodin : par exemple, les mailles de niveau temporel 2, qui ne sont calculées qu'une seule fois au cours d'une itération complète, **doivent** l'être dès la première sous-itération. Ainsi, leur valeur est disponible pour leurs mailles voisines de niveau temporel 1, 2 ou 3.

On le voit, ce schéma temporel répartit les calculs de manière déséquilibrée entre les sous-itérations. Cela aura potentiellement un impact néfaste sur l'équilibrage de charge durant l'exécution puisque, comme nous le verrons, la répartition des mailles parmi les cœurs de la machine obéit à d'autres critères d'ordre topologique.

1.3 Modèle de programmation parallèle pour un code de simulation numérique

De la section précédente, on retient que FLUSEPA décrit les objets et flux à l'aide d'une représentation sous forme de maillage et détermine, grâce à une intégration temporelle, à quelle

fréquence les valeurs de ces mailles doivent être actualisées pour faire progresser la simulation.

Dans une version antérieure, le code FLUSEPA s'appuyait sur un modèle parallèle hybride classique (MPI+OpenMP) dans lequel le parallélisme intra-nœud était pris en charge par des *threads* OpenMP qui se synchronisaient à l'aide de barrières entre chaque sous-itération. Cette stratégie générait par conséquent un nombre important de sur-synchronisations en introduisant des barrières de synchronisation obligeant tous les processus à s'attendre mutuellement entre chaque sous-itération, ce qui n'est techniquement pas nécessaire pour tous les calculs.

1.3.1 Approche à base de tâches

Afin de respecter finement ces synchronisations et éviter une sur-synchronisation induite par des barrières (ou rendez-vous) collectives, une version à base de tâches a été développée durant la thèse de Jean-Marie Couteyen[11].

En effet, le travail généré par le traitement des mailles peut, de façon naturelle, s'exprimer sous forme d'un ensemble de tâches de calcul, dont certaines pourront s'exécuter en parallèle. Afin de respecter les contraintes de précédence entre les tâches de sous-itérations successives, un modèle de programmation à base de tâches avec dépendances a été mis en place dans FLUSEPA : les tâches sont reliées entre elles par des dépendances qui indiquent que certaines tâches ne peuvent débuter tant que d'autres ne sont pas terminées. Cette approche peut donc être représentée par un graphe orienté dont les sommets correspondent aux tâches et les arêtes aux dépendances.

Une approche basée sur des tâches permet de mettre en œuvre des synchronisations plus précises et donc d'augmenter l'asynchronisme global de l'exécution. Cela permet en outre une exécution plus souple des calculs entre les ressources disponibles : en fonction des dépendances exprimées dans le graphe, plusieurs ordres d'exécution de ces tâches sont valides.

1.3.2 Grandes étapes de traitement impliquées dans FLUSEPA

Nous donnons ici une vision globale de FLUSEPA en exposant les principales étapes de son exécution, depuis la lecture d'un maillage fourni entrée jusqu'à la génération d'un fichier contenant les valeurs calculées pour toutes les mailles à chaque fin d'itération. La figure 5 met en lumière les trois étapes majeures qui rythment son fonctionnement.

La première étape consiste à effectuer une décomposition de domaine. Le maillage donné en entrée est partitionné en plusieurs sous-maillages grâce à la bibliothèque Scotch. Ces sous-maillages sont assignés aux différents nœuds de la machine lors d'une sous-étape de *mapping* [22]. Puis, le graphe de tâches à exécuter à chaque itération est construit à partir de ce partitionnement, en prenant en compte de nombreux paramètres tels que les traitements à appliquer sur les cellules et les faces, le niveau temporel des mailles, etc. Les sommets de ce graphe correspondent aux tâches à exécuter et les arêtes aux dépendances entre ces tâches. Enfin, la dernière étape consiste à ordonnancer ces tâches sur les différentes unités de calcul à l'aide d'un support d'exécution dédié (StarPU[2] dans notre cas).

En pratique, les deux dernières étapes se déroulent simultanément, en *pipeline* : le graphe de tâches n'est jamais généré intégralement en une seule fois. Au contraire, au fur et à mesure que les tâches sont créées, elles sont soumises au support d'exécution sous-jacent qui les exécute dès que possible.

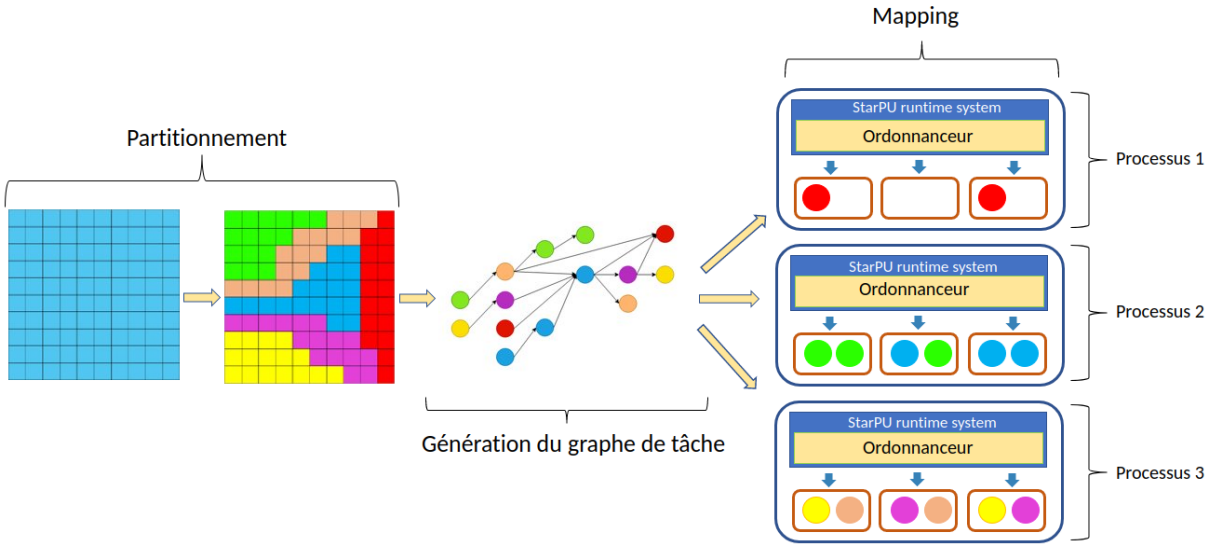


FIGURE 5 – Fonctionnement du code de simulation FLUSEPA.

1.4 Partitionnement du domaine

Première étape de la chaîne, la décomposition de domaine a pour finalité d'attribuer un sous-domaine à chaque nœud de la machine cible. Idéalement, cette décomposition doit être équilibrée, c'est-à-dire aboutir à une quantité de calcul similaire sur chaque nœud, et mener à des sous-domaines connexes avec un voisinnage de taille raisonnable, pour éviter de trop nombreuses communications.

1.4.1 État de l'art

Hormis pour des géométries assez simples où un partitionnement purement géométrique est possible, l'immense majorité des simulations numériques délèguent à des *partitionneurs de graphes* le soin de découper le domaine (3D ou 2D) en sous-domaines.

Certains partitionneurs découpent un graphe déduit du maillage où les sommets réprésentent les cellules du maillage et les arêtes les faces, en s'efforçant de minimiser une fonction de coût. Ce problème connu de partitionnement de graphe, qui s'applique à une variété de domaines que ce soit dans la recherche ou l'industrie, est NP-Difficile. Théoriquement, le problème consiste à diviser un graphe en un nombre de partitions équilibrées tout en minimisant le plus possible la présence d'arêtes entre ces partitions. Dans un contexte de mailles qui vont engendrer des calculs mais aussi des communications avec leurs voisines, ce problème s'exprime par une problématique d'équilibrage de charge de calcul entre ces partitions et où les communications et synchronisations entre unités de calcul doivent être minimisées en limitant la présence d'arêtes de faces entre les partitions.

Les outils de partitionnement utilisent des heuristiques afin d'obtenir la meilleure décomposition de domaine possible en un temps raisonnable. Ces outils implémentent majoritairement un algorithme multi-niveau [21], fournissant des partitions de très bonne qualité. Cet algorithme se décompose en trois étapes visibles sur la figure 6.

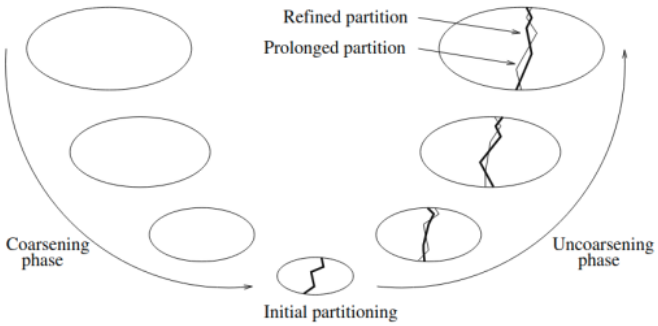


FIGURE 6 – Représentation des différentes phases de l'algorithme de partitionnement multi-niveau : compression, partitionnement et expansion [21].

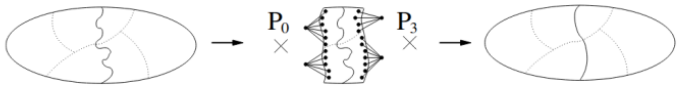


FIGURE 7 – Dernière phase de l'algorithme où les partitions sont raffinées localement afin d'améliorer le partitionnement initial [21].

La première étape correspond à la contraction du graphe où celui-ci est réduit progressivement en fusionnant sommets et arêtes jusqu'à atteindre un certain seuil de nombre de sommets. Cette étape permet une réduction du problème initial en une variante plus grossière où un partitionnement peut être appliqué. Ce partitionnement initial cherche donc à obtenir une décomposition la plus optimale possible en appliquant une heuristique. La décomposition obtenue est ensuite projetée sur ce graphe réduit. Ce dernier est alors décompressé successivement jusqu'à obtenir le graphe initial. Á chaque étape d'expansion, le partitionnement est raffiné localement. La figure 7 expose ce processus qui consiste à changer de partition des sommets situés en frontière si et seulement si cela améliore la partition globale. Le résultat obtenu sur le plus petit graphe est donc propagé jusqu'au graphe d'origine.

De nombreaux outils implémentent cet algorithme. On peut citer notamment Metis [18], hMetis [16], KaHIP [25] (Karlsruhe High Quality Partitioning), PatoH [26] (Partitioning Tools for Hypergraphs) et bien sûr la bibliothèque Scotch [23] qui est développée au Laboratoire Bordelais de Recherche en Informatique (LABRI). Chacun de ces partitionneurs multi-niveaux, bien qu'implémentant le même algorithme, offrent des variantes et des fonctionalités qui les rendent plus adaptés à certaines applications. hMetis et PatoH sont particulièrement bien adaptés au partitionnement d'hypergraphes par exemple. De plus, bien que ces outils s'exécutent souvent de manière séquentielle, certains d'entre eux possèdent une variante parallèle distribuée, utile quand les graphes dépassent plusieurs dizaines de millions de sommet : c'est le cas de Metis et de Scotch, qui se déclinent respectivement en Par-Metis [17] et en PT-Scotch [9].

FLUSEPA s'appuie sur la variante séquentielle de Scotch. Polyvalente, elle propose aussi des fonctions de projection de graphes sur des architectures distribuées, qui permettent par exemple de

pouvoir répartir un grand nombre de sous-domaines sur un petit nombre de machines (surjection).

Comme la plupart des bibliothèques de partitionnement, Scotch est *mono-critère* : un seul paramètre est utilisé pour estimer la qualité d'un partitionnement. Déterminé par l'utilisateur, ce paramètre se caractéristise par un poids attribué à chaque sommet du graphe, indiquant notamment le coût de traitement de la cellule qu'il représente. En fonction de ces sommets pondérés, Scotch va produire une décomposition où les différentes partitions se répartissent équitablement ces poids.

1.4.2 Intégration dans le code de simulation numérique et pondération des sommets

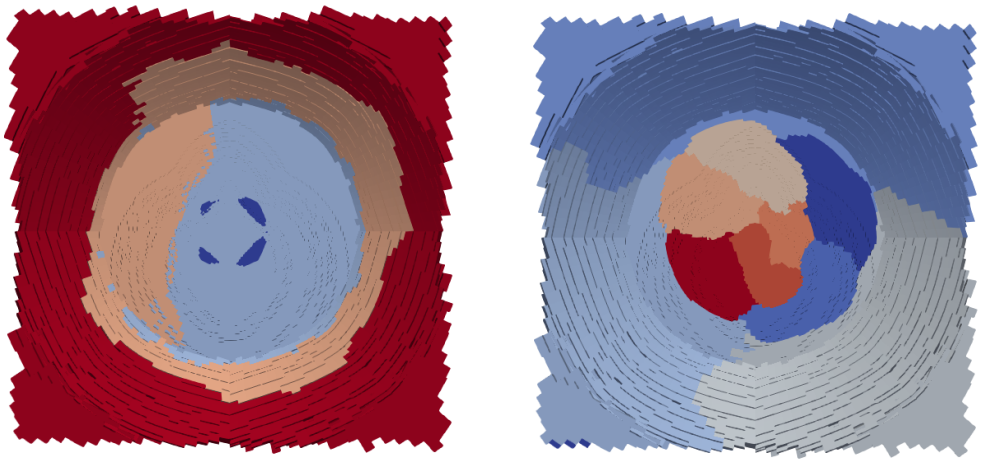
Comme vu précédemment, FLUSEPA effectue une discrétisation spatiale qui génère des mailles représentant une portion physique du phénomène à simuler. Une maille est constituée d'une cellule heaxèdre et de ses faces qui marquent les bordures avec les cellules voisines.

Le code de simulation s'appuie sur la géométrie du maillage pour générer le graphe de tâches qui doit s'exécuter sur les cœurs. Il est donc possible de créer les tâches de calcul en se basant directement sur les faces et les cellules où par exemple chaque tâche traite seulement une seule cellule du maillage. Cependant, la quantité de travail par cellule et/ou par face est trop faible pour couvrir le coût de création d'une tâche ainsi que les synchronisations induites. Il est donc nécéssaire de former des groupes de mailles qui constitueront l'unité de base que les tâches pourront manipuler. Le code de simulation pourra alors générer des tâches plus conséquentes en terme de travail. Pour ce faire, la bibliothèque Scotch prend donc en entrée le maillage original, l'interprète sous forme de graphe pour former des partitions de taille choisie (i.e. les groupes de mailles), puis fournit en sortie le maillage partitionné.

Lorsqu'on ne transmet aucune pondération particulière à la bibliothèque Scotch, cela revient à dire que toutes les cellules ont un coût opératoire analogue entre elles (poids = 1). L'outil produit donc un partitionnement équilibré en terme de nombre de cellules. Toutefois, comme discuté en section 1.2.2 (figure 4), la particularité de l'intégration temporelle adaptative est d'introduire des cellules de niveaux temporels différents, qui conduiront à des temps de calcul disparates. Plus le niveau temporel est faible, plus la cellule a un coût opératoire élevé.

Afin de prendre en compte ces caractéristiques dans l'étape de partitionnement, nous définissons le poids d'un sommet du graphe (c'est-à-dire le coût opératoire d'une cellule) ainsi : $C = 2^{\theta-\tau}$ avec θ le niveau temporel maximum rencontré et τ le niveau temporel de la cellule. Grâce cette pondération des sommets, Scotch construit désormais des partitions de coût opératoire équilibré.

Notons que Scotch continue de s'appuyer sur la localité topologique pour minimiser la présence d'arêtes entre partitions. Ainsi, même si la charge de travail théorique est désormais répartie équitablement, il est possible qu'un domaine récupère quelques cellules coûteuses et qu'un autre récupère de nombreuses cellules ne demandant presque pas de travail.



(a) Mailles coloriées en fonction du niveau temporel

(b) Mailles coloriées en fonction du domaine d'appartenance

FIGURE 8 – Partitionnement d'un maillage en dix domaines avec la bibliothèque Scotch en fonction de la répartition des niveaux temporels.

À titre d'illustration, la figure 8a présente un extrait de maillage où les cellules sont coloriées en fonction de leur niveau temporel. Le plus faible, en bleu foncé, est situé au centre. Plus on s'en éloigne, plus le niveau temporel diminue. La figure 8b présente le partitionnement effectué par Scotch en fonction de ces niveaux temporels où les mailles sont coloriées en fonction de leur domaine d'appartenance. Des domaines très grands sont donc observés en bordure du maillage où sont situées les cellules peu coûteuses. La dimension de ces partitions diminue en se rapprochant du centre puisqu'elles contiennent des cellules plus coûteuses. En effet, pour compenser les coûts, les domaines peuvent être de dimension très variable. Nous verrons plus tard l'incidence de cette caractéristique sur l'équilibrage de charge.

1.5 Génération du graphe de tâche

Le maillage fraîchement partitionné, le code de simulation numérique va, pour chaque domaine, générer un ensemble de tâches spécifiques. Comme dit précédemment, le partitionnement influe fortement sur la génération du graphe de tâches puisque celles-ci ont la charge d'actualiser les valeurs stockées dans les cellules et les faces. Le parallélisme extrait de cette approche par tâche est donc essentiellement spatial.

FLUSEPA distingue, au sein de ces domaines, plusieures composantes [2] différentes pour décrire le maillage. Un domaine peut être composé de cellules et faces *internes* mais aussi de cellules et faces *externes*, formant à elles quatre les composantes définies dans FLUSEPA. Une composante est externe si elle est voisine avec d'autres composantes qui appartiennent à des domaines différents. Á l'inverse, une composante est interne si elle n'est voisine qu'avec des cellules et des faces de composantes appartenant au même domaine. Ces quatre composantes sont visibles sur la figure 9.

2. Á noter que dans le cadre de ce document, seules certaines composantes sont mentionnées.

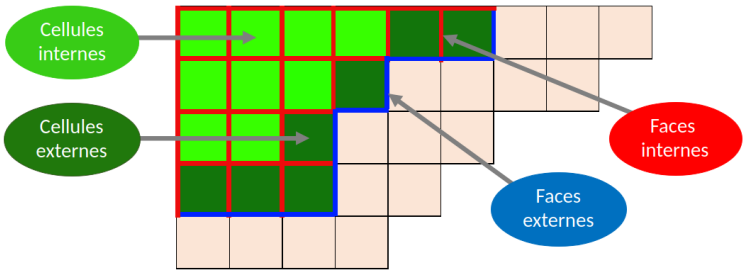
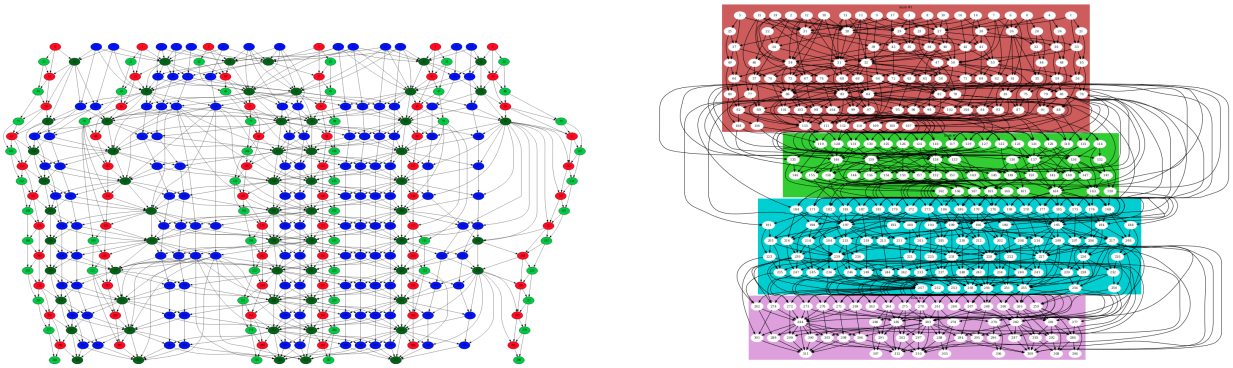


FIGURE 9 – Distinction des différentes composantes d'un maillage 2D schématisé.

FLUSEPA définit quatre types de tâches de calcul, chacun ayant pour résultat la mise à jour des valeurs d'un type de composante. Produire des tâches spécifiques pour calculer chacune des composantes permet d'introduire davantage d'asynchronisme au sein du graphe de tâches. En s'abstrayant d'un ensemble de paramètres secondaires qui influent sur la forme du graphe final, il est possible de décrire schématiquement le fonctionnement des fonctions de génération des tâches implémentées par FLUSEPA. Ces fonctions de génération sont exécutées pour chaque domaine du maillage. En itérant sur les composantes d'un domaine, la boucle de génération produit :

— 1 tâche qui traite l'ensemble de ces cellules internes,
— 1 tâche qui traite l'ensemble de ces cellules externes,
— 1 tâche qui traite l'ensemble de ces faces internes,
— 1 tâche qui traite l'ensemble de ces faces externes.



(a) Colorié en fonction de la composante traitée par la tâche

(b) Colorié en fonction de la sous-itération de la tâche

FIGURE 10 – Graphe de tâche obtenu à partir d'un maillage 2D avec $\theta = 2$.

La figure 10a présente le plus petit graphe que nous avons généré : il comporte 313 tâches au total. Ce faible nombre de tâches est dû à l'utilisation d'un maillage 2D produit à la main d'à peine une centaine de cellules. En comparaison, le maillage fourni par *Airbus* présente plus de huit millions de cellules et, suivant les paramètres, mène à la génération de graphes de l'ordre de deux millions de tâches. Notre petit exemple est donc utile pour appréhender la forme du graphe, repérer les tâches qui vont « libérer » le plus de parallèlisme (c'est-à-dire possédant de nombreuses arêtes sortantes), etc. Sur la figure 10a, on distingue nos 4 types de tâches dont les couleurs correspondent à celles

de la figure 9. On remarque que les tâches calculant les cellules dépendent des tâches calculant les faces, et inversement. Nous reviendrons sur ce point dans les sections suivantes. Surtout, on remarque que le graphe révèle une certaine irrégularité, avec peu de motifs récurrents d'un *étage* à l'autre. Cela provient de la méthode d'intégration temporelle adaptative, comme nous allons en discuter maintenant.

1.5.1 Influence de la méthode d'intégration temporelle adaptative

Outre le partitionnement, l'intégration temporelle adaptative influe aussi directement sur la génération du graphe de tâches. Comme mentionné précédemment, cette méthode range les cellules et les faces dans des classes de niveau temporel, reflétant directement la fréquence de calcul nécéssaire pour chaque composante. Les domaines issus du partitionnement sont typiquement composés de cellules et de faces de niveaux temporels variables. Lors de la génération des tâches, pour chaque sous-itération, seules les composantes ayant besoin d'être recalculées vont provoquer l'insertion de tâches dans le graphe.

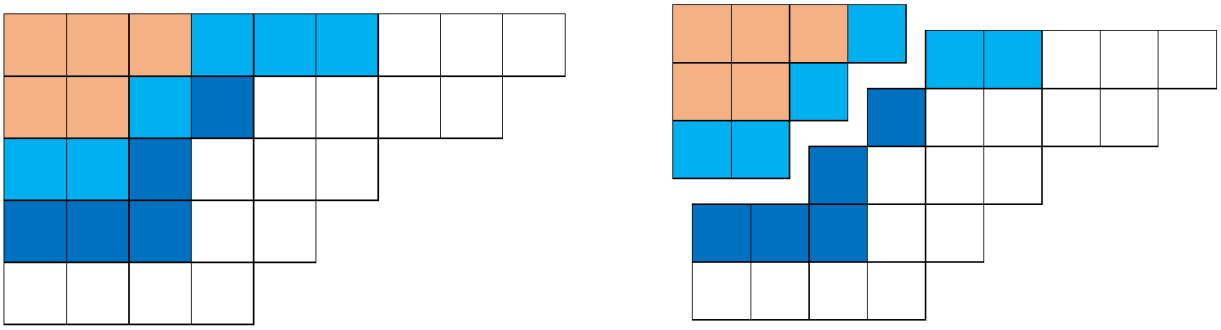


FIGURE 11 – Maillage 2D où les mailles sont coloriées en fonction de leurs niveaux temporels.

Le maillage 2D de la figure 11 présente trois niveaux temporels différents. Bleu foncé correspond au niveau temporel $\tau = 0$, bleu clair à $\tau = 1$ et enfin saumon à $\tau = 2$. Prenons le cas de la génération de tâche traitant les cellules pour un domaine. En s'appuyant sur la figure 4, les fonctions de génération itérant sur ce domaine vont générer pour la première sous-itération (qui rappelons-le traite tous les niveaux temporels) :

— 1 tâche qui calcule l'ensemble des cellules externes de niveau 0
— 1 tâche qui calcule l'ensemble des cellules externes de niveau 1
— 1 tâche qui calcule l'ensemble des cellules internes de niveau 1
— 1 tâche qui calcule l'ensemble des cellules internes de niveau 2

En revanche, la seconde sous-itération où les niveaux temporels 1 et 2 ne doivent pas être calculés, génère seulement :

— 1 tâche qui calcule l'ensemble des cellules externes de niveau 0

Une première observation que l'on peut faire est que certaines sous-itérations vont produire beaucoup plus de tâches et donc beaucoup plus de travail que d'autres, ce qui se confirme sur la figure 10b. Cette dernière correspond au graphe de tâche de la figure 10a, généré à partir du petit maillage 2D, mais où des blocs de couleurs entourent les tâches générées lors de chaque sous-itération. Les quatre sous-itérations ont, comme niveau temporel traité le plus élévé (aucune maille de niveau

supérieur n'est calculée), respectivement 3, 0, 2, 0. On constate aisèment que la quantité de tâches incluses dans un bloc est représentative du niveau temporel maximum traité dans la sous-itération.

La géométrie du maillage, c'est-à-dire sa décomposition en domaine et l'intégration temporelle adaptative influent donc fortement sur le façonnage de ce graphe de tâches.

1.6 Ordonnancement et distribution des tâches sur les unités de calcul

Les supercalculateurs contemporains possèdent tous une architecture distribuée : ils sont formés de nœuds – qui sont des ordinateurs à mémoire commune – interconnectés par un réseau rapide, comme illustré en figure 12. Cette architecture est souvent appelée « grappe de PC » (*PC Cluster* en anglais). Chaque nœud possède plusieurs cœurs de calculs qui se partagent [3] une mémoire vive, et parfois ils sont équipés d'accélérateurs de calculs tels que des cartes graphiques (GPU).

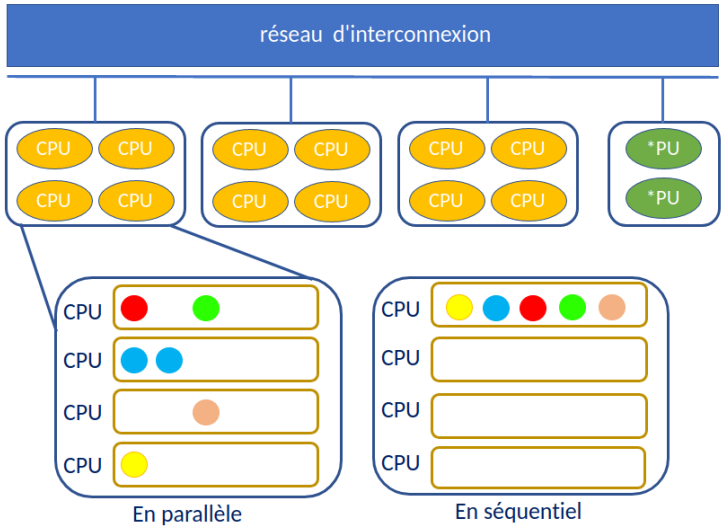


FIGURE 12 – Schématisation d'une mémoire distribuée.

Ordonnancer des tâches au sein d'un nœud peut s'effectuer de manière efficace car toutes les opérations de synchronisation (e.g. pour gérer les dépendances, ou l'accès à des sections critiques du code) et d'échange de données peuvent être implémentées au travers de la mémoire, voire directement entre les cœurs eux-mêmes. De nombreux supports d'exécution permettent aisément de générer et d'ordonnancer efficacement des graphes de tâches au sein d'un nœud. Le code FLUSEPA utilise le support d'exécution StarPU développé à l'Inria Bordeaux Sud-Ouest. Concrètement, lors de la création d'une tâche, l'application indique les zones mémoire auxquelles la tâche va accéder ainsi que le mode d'accès (lecture/écriture). StarPU en déduit automatiquement les dépendances entre les tâches de manière à ce que la cohérence des données soit la même que pour une exécution séquentielle des tâches (STF Order, *sequential task flow*).

En revanche, les choses se compliquent lorsqu'il s'agit d'ordonnancer un graphe de tâches à l'échelle de la machine, car tous les échanges de données et les synchronisations entre les nœuds

3. Les mémoires étant souvent hiérarchiques, les accès des cœurs aux différents bancs mémoire ne sont pas uniformes, mais cet aspect n'entre pas dans la portée de cette étude.

doivent traverser le réseau. L'approche la plus répandue pour implanter ces communications inter-nœuds consiste à utiliser une bibliothèque d'envoi/réception de messages telle que MPI [19] (*Message Passing Interface*). L'idée est d'utiliser un processus (au sens Unix) par nœud, chacun s'appuyant sur une instance de support d'exécution capable d'ordonnancer localement des tâches. L'application a évidemment la lourde charge de partitionner le graphe de tâches sur l'ensemble des nœuds de la machine, et d'implanter les mouvements de données et les synchronisations inter-nœuds à l'aide d'échanges de messages.

Dans notre cas, le graphe de tâches n'est pas véritablement partitionné : ce sont les sous-domaines (i.e. ensembles de mailles) qui sont répartis entre les nœuds, les tâches résultantes sont créées au sein de chaque nœud pour chaque composante dont il est responsable. Bien sûr, les composantes externes aux domaines possèdent des voisines distantes, ce qui se traduit par des dépendances entre tâches s'exécutant au sein de nœuds différents.

2 FLUSIM : un émulateur d'exécution

FLUSEPA est un code de production de plus de 160.000 lignes. Dans le meilleur des cas, une seconde physique peut-être simulée en deux jours et une simulation complète comme un décollage de fusée peut atteindre une durée de sept jours, sans compter le temps de compilation. Ce code de simulation FLUSEPA ne peut donc pas être directement lancé sur une machine de travail classique mais doit être exécuté sur un cluster de nœuds distants. D'autres personnes peuvent accèder à ce supercalculateur pour profiter de cette puissance de calcul disponible. Les ressources nécéssaires aux calculs, c'est-à-dire les nœuds, doivent donc être réservées au préalable afin que l'exécution de FLUSEPA ne soit pas pertubée par d'autres calculs extérieurs.

Pour un stage n'excédant pas une période de six mois, observer l'influence des nombreux paramètres et tester de nouvelles implémentations constitueraient une entreprise complexe et chronophage. Aussi, *ArianeGroup* a mis au point un émulateur d'exécution du code de simulation FLUSEPA développé spécifiquement pour contourner ce problème. Cet émulateur, nommé FLUSIM, est écrit en Python et peut s'exécuter sur une machine personnelle. Il reproduit le comportement du code de simulation en évitant de calculer les valeurs physiques stockées dans le maillage. L'émulateur se contente simplement d'estimer la durée des calculs, ce qui permet en pratique de ramener le temps d'exécution de plusieurs jours à quelques heures.

2.1 Fonctionnement intrinsèque de FLUSIM

La première chose à noter sur cet émulateur est qu'il reproduit un environnement d'exécution idéal où certaines réalités physiques sont ignorées. C'est le cas des coûts de communication entre les nœuds par exemple, qui sont complètement négligés. Il en va de même pour le coût de création des tâches qui, même s'il est en général faible, vaut ici zéro.

En revanche, une caractéristique très intéressante de cet outil est que les exécutions sont reproductibles. Si les paramètres sont inchangés d'une exécution à l'autre, l'exécution produite sera identique à la précédente. Le maillage partitionné, le graphe de tâche produit et l'exécution de ces dernières seront en tout point identique.

Les objectifs d'*ArianeGroup* en implémentant FLUSIM sont multiples. D'une part, cela permet de tester rapidement de nouvelles stratégies et implémentations. D'autre part, cet émulateur vise à isoler les étapes principales du processus de simulation (partitionnement, création du graphe et

ordonnancement) afin de mieux identifier les problèmes de période d'inactivité présents sur les cœurs lors de l'exécution du code. C'est d'ailleurs ce qui a motivé ce stage.

La figure 13 présente les différentes étapes du simulateur FLUSIM. Cette section décrit en détails les différences avec le code original. Une première observation est que le nombre de processus n'est pas ici défini par l'architecture du cluster de nœuds mais directement choisi par l'utilisateur et passé en paramètre. La première étape de partitionnement avec la bibliothèque Scotch reste identique. Une décomposition de domaine avec un critère d'équilibrage des coûts est effectuée puis est passée en entrée des fonctions de génération des tâches. En revanche, la seconde étape diffère. Les tâches ne sont pas générées immédiatement après la décomposition. Cela s'explique par le fait que les outils MPI et StarPU ne sont pas directement appelés dans l'émulateur qui reproduit simplement le comportement de ces derniers.

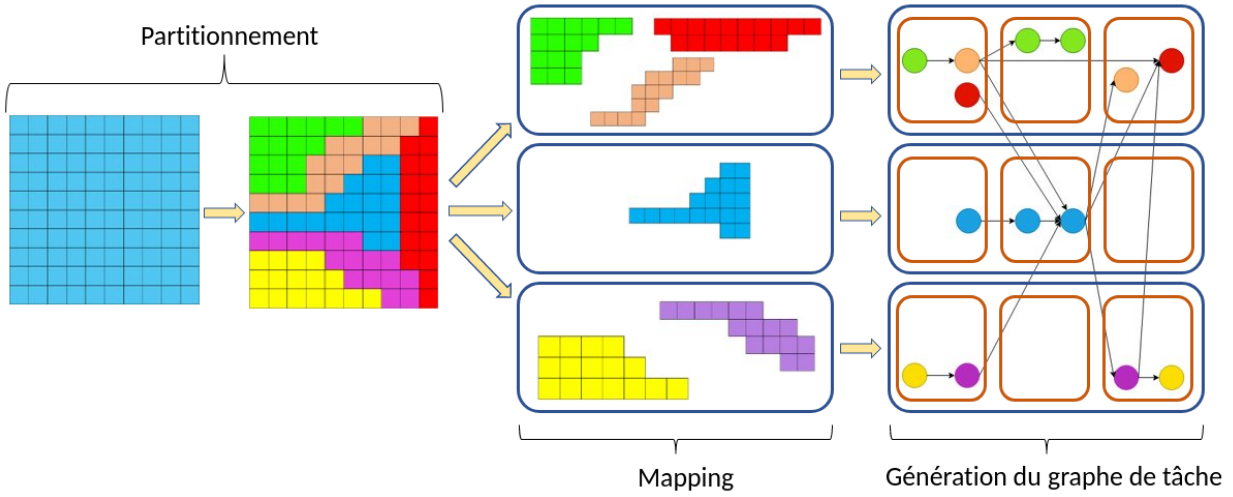


FIGURE 13 – Fonctionnement du code de simulation FLUSIM.

La seconde étape consiste à distribuer les partitions obtenues entre les différents processus. Lorsque la répartition des mailles entre domaines est connue après le partitionnement, FLUSIM va déterminer au préalable quel domaine s'exécute sur quel processus en invoquant l'outil de *mapping* de Scotch de manière similaire à FLUSEPA. En revanche, l'architecture n'est pas prise en compte puisque c'est un émulateur. Cependant, les domaines vont être distribués de manière à ce que chaque processus se voit attribuer la même charge de travail en passant un critère de distribution (comme pour la première étape de partitionnement). Toutes les tâches générées par un domaine vont s'exécuter sur le processus auquel leur domaine d'appartenance a été attribué.

L'étape de génération du graphe de tâches est similaire. En fonction de l'intégration temporelle et de la géométrie de la décomposition, un graphe de tâches est généré. Dans ce graphe, on retrouve nos quatre types de tâches pour traiter les quatre composantes qui décrivent le maillage. Comme StarPU n'est pas directement utilisé, FLUSIM doit imiter son comportement. C'est-à-dire générer lui-même les dépendances et ordonnancer les tâches. Une fois que les tâches ont été définies, FLUSIM, ayant stocké les accès mémoire pour chaque type de tâche, calcule toutes les dépendances. Cette implémentation est rendue possible par le fait que ces tâches sont empilées (et donc stockées) au sein d'une structure dans leur ordre de création et les valeurs nécéssaires à leurs calculs ont théoriquement été calculées au préalable. Ce qui signifie que les dépendances se génèrent en regardant le dernier

accès mémoire à la structure de la composante nécéssaire.

Algorithm 1: Dag generation function in Flusim

```
for each sub-iteration k do
    nsit ← maximum-temporal-level-reached-at-k;
    while loops ≠ ∅ do
        Foreach-F(predictor, nsit);
        Foreach-C(predictor, nsit);
        loops ← loops − 1;
    end while
    while nsit ≠ ∅ do
        while loops ≠ ∅ do
            Foreach-F(corrector, nsit);
            Foreach-C(corrector, nsit);
            nsit ← nsit − 1;
        end while
    end while
end for
```

FIGURE 14 – Algorithme d'insertion des tâches pour le solveur aérodynamique.

Les fonctions de génération de tâches implémentées par FLUSIM et visibles sur la figure 14 sous le nom de FOREACH_F et FOREACH_C déterminent, pour chaque tâche, sa durée d'exécution théorique. La durée d'une tâche est déterminée par la quantité de la composante qu'elle calcule d'un niveau temporel donné au sein d'un domaine. Par exemple, la tâche qui traite l'ensemble des cellules internes de niveau temporel 0 dans un domaine va compter le nombre de ces cellules internes et estimer sa durée avec un coefficient.

Une fois le graphe de tâche généré complètement, FLUSIM simule l'algorithme d'ordonnancement utilisé par défaut dans StarPU (décrit plus en détail par la suite).

2.1.1 Visualisation de l'exécution simulée

Pour comprendre d'où proviennent d'éventuels problèmes de performance, la durée totale d'une exécution est évidemment insuffisante. Aussi, FLUSIM fournit un outil de post-traitement, écrit en R, permettant de visualiser notamment des traces d'exécution.

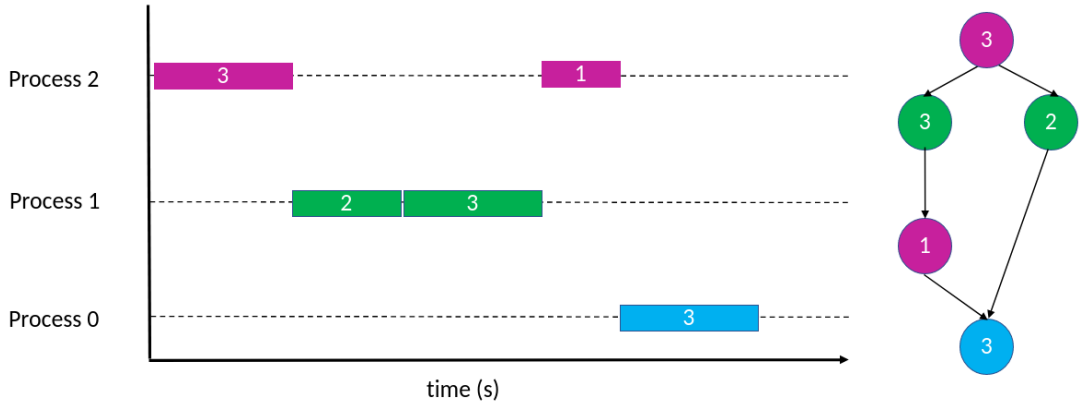


FIGURE 15 – Schématisation d'une trace d'exécution obtenue en sortie de l'émulateur FLUSIM.

Une trace d'exécution enregistre, pour chaque tâche, ses dates de début et de fin ainsi que le cœur sur lequel elle a été exécutée. La figure 15 présente la visualisation (sous forme d'un diagramme de Gantt) d'une trace d'exécution « jouet ». Le temps est représenté en abscisse, tandis que les processus figurent en ordonnée. Les rectangles de couleur représentent l'empreinte temporelle des tâches. Dans cet exemple, trois domaines ont générés des tâches, chacun distribué sur un nœud différent (ou processus). Les dépendances entre tâches sont toutes respectées.

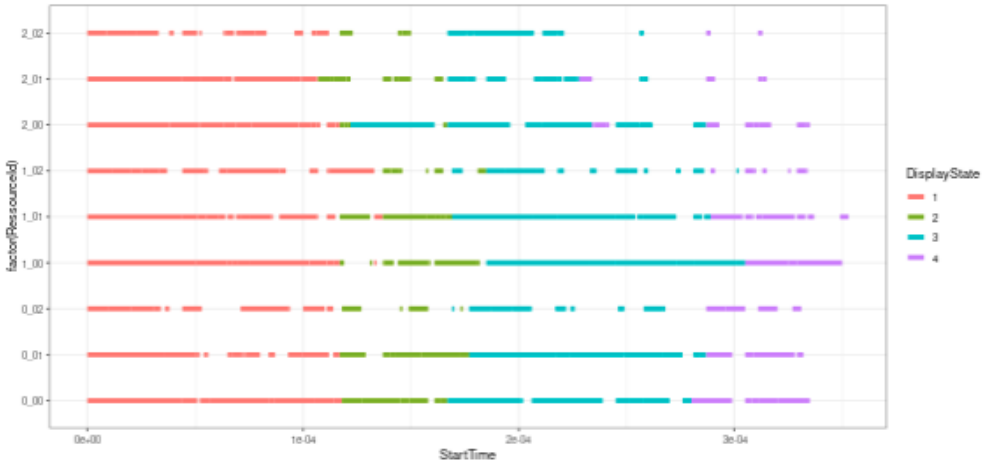


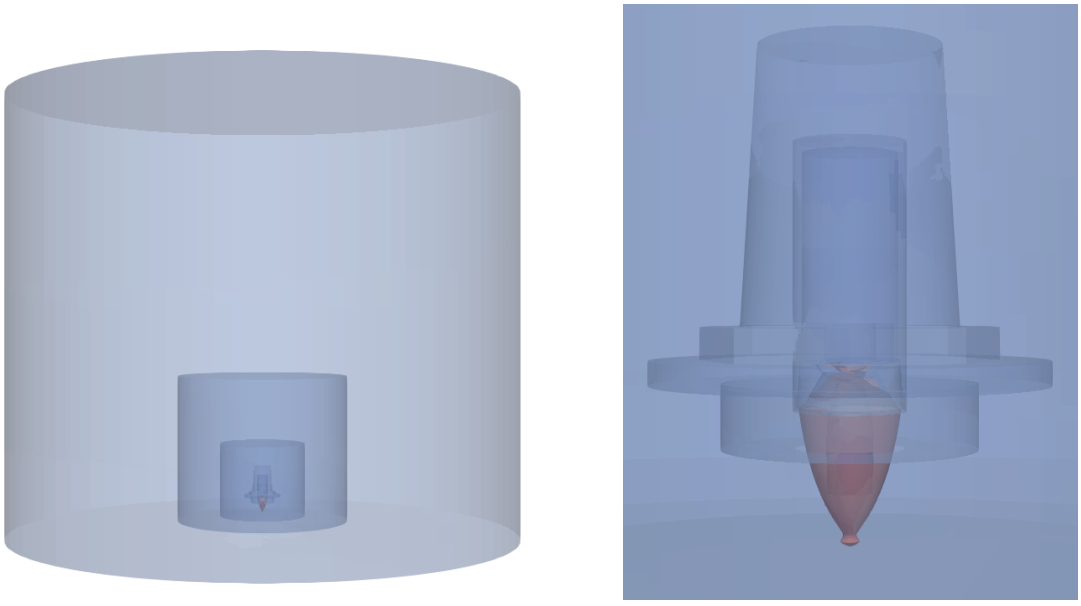
FIGURE 16 – Trace obtenue en sortie du petit simulateur de maillage 2D où les tâches sont coloriées en fonction de leur sous-itération.

La figure 16 correspond à la trace d'exécution du graphe de la figure 10a. Les tâches sont ici coloriées en fonction de la sous-itération où elles ont été générées. La représentation de chaque couleur correspondant à celle de la figure 10b. Á noter que les rectangles de couleur correspondent à des tâches compressées. Dans une telle trace issue de l'exécution d'un gros graphe, il est difficile de distinguer individuellement les tâches sans zoomer fortement avec l'outil. Toutes les expériences réalisées par la suite sont effectuées au sein de l'environnement FLUSIM.

Troisième partie

Contributions

Dans le cadre de ce stage, le maillage de référence fourni par *Airbus* est présenté figure 17. Le cylindre (fig. 17a) est composé de 8 millions de cellules et 26 millions de faces, contenant une pièce interne particulière visible sur la figure 17b. Il est décrit dans un fichier d'extension *.cgns* et peut-être visualisé à l'aide du logiciel libre de visualisation de données Paraview[1].



(a) Visualisation par transparence de la composition du maillage

(b) Zoom sur la pièce centrale intérieure

FIGURE 17 – Aperçu global du maillage cylindrique.

3 Outils de visualisation et scripts automatisés

Le code de simulation FLUSIM prend en entrée un maillage et produit en sortie une trace d'exécution. Avec un nombre important d'éléments paramétrisables, comprendre le fonctionnement du simulateur et l'influence de chacune de ses étapes et variables constituent une tâche hardue. La première étape de ce stage s'est donc revélée être le développement d'outils annexes permettant de répondre à ces problématiques mais aussi de faciliter la recherche de la cause des périodes d'inactivité présentes dans l'exécution.

3.1 Visualisation de graphe de tâche

Les travaux de M. Jean-Marie Couteyen-Carpaye [12] [13] [11], à l'origine de la parallélisation à base de tâche du code original, présentent quelques exemples de graphes obtenus à partir de ce type d'application. Cependant, ces graphes proviennent de FLUSEPA à partir de maillages assez différents du nôtre. Obtenir un aperçu du graphe de tâche généré spécifiquement par FLUSIM

avec notre maillage cylindrique a permis d'extraire des informations détaillées de l'exécution. D'une part, sur le fonctionnement du processus intrinsèque de génération du graphe et de l'influence des divers paramètres sur cette génération (en particulier l'intégration temporelle et la géométrie du maillage). D'autre part, sur l'organisation générale du graphe puisqu'une forme particulière peut être un indicateur d'une exécution plus efficace.

Dans ce but, différents logiciels et bibliothèques ont été testés durant ce stage. Le choix de l'outil final s'est basé sur un compromis entre visualisation de large graphe et souplesse d'affichage d'informations sur ces tâches. En effet, des logiciels tels que Tulip permettent de visualiser spécifiquement des graphes de tâche très larges. Cependant, cet outil est moins adapté pour l'affichage rapide du graphe selon un certain critère. Colorier les tâches qui se trouvent dans le chemin critique ou explorer spécifiquement une partie du graphe pour visualiser les dépendances par exemple reste peu intuitif. La librairie PyVis est l'outil qui a été utilisé durant les premières semaines du stage. Il s'agit d'un module Python qui permet de facilement s'intégrer au code de FLUSIM où les structures qui gèrent les tâches et les dépendances sont elles-aussi en Python. Avec un rendu intéractif, PyVis permet de facilement explorer le graphe en détails, et peut mettre en valeur certains critères. Bien que l'entièreté du graphe pouvait être affichée, le placement hiérarchique des tâches n'était pas intuitif.

Le choix final s'est porté sur le logiciel GraphViz. Une fonction, implémentée et appelée directement dans le simulateur, récupère les informations nécéssaires après l'exécution et génère par la suite un fichier *.dot*. Avec l'extension du logiciel dans l'IDE Visual Studio Code, le *.dot* peut être affiché sous forme de graphe. L'ensemble des graphes présentés dans ce rapport est généré grâce à cet outil, même ceux de la section précédente. Bien que seule une partie du graphe peut être affichée, l'implémentation des fonctions de génération de *.dot* exploitable est très intuitive. Ce module Python permet donc de produire aisèment des représentations de graphes aérées tout en soulignant de manière intelligible un aspect particulier.

En fonction des interrogations, intuitions et implémentations de ce stage, plusieures fonctions ont été implémentées, permettant chacune de souligner des éléments distinctifs dans le graphe. Les tâches peuvent donc être organisées dans l'affichage de manière à distinguer pour chacune d'entre elles le type de composante de domaine qu'elles calculent, leur appartenance ou non au chemin critique (plus long chemin séquentiel), leur niveau de priorité et la distribution globale de cette dernière sur l'ensemble du graphe, la sous-itération à laquelle elles ont été générées et enfin le processus sur lequel elles ont été distribuées.

3.2 Trace d'exécution

Comme mentionné dans les sections précédentes, FLUSIM fournit un outil de post-traitement des informations receuillies pendant l'exécution du simulateur. Cet outil, écrit en langage R, permet de visualiser l'exécution des tâches dans le temps sur les différents processus. Il est aussi possible de zoomer en détails sur certains extraits de la trace, ce qui permet de pouvoir distinguer plus facilement ces tâches très compressées sur l'affichage. En sortie du simulateur, un fichier *.csv* est généré avec de nombreuses informations sur le déroulement de l'exécution. Ces informations se traduisent par les caractéristiques de chaque tâche notammment le temps d'exécution, le temps de départ, le numéro du cœur et du processus de la tâche... Ce fichier est ensuite lu par l'outil de post-traitement pour générer une trace d'exécution.

Visualiser le déroulement d'une exécution d'un graphe de moins d'une centaine de tâches est aisé puisque les tâches ne sont pas encore totalement compressées sur la trace. En revanche, lorsque

le graphe possède 500.000 à 2 millions de tâches, cela devient plus complexe.

3.2.1 Métriques

Afin de compenser cette visibilité réduite, l'outil a été étendu sur plusieurs aspects. Le premier consiste en l'ajout de métrique directement visible lors de l'affichage de la trace. Au fur et à mesure du stage, plusieures d'entre elles ont été implémentées.

L'implémentation d'une métrique consiste à générer un fichier *.csv* en fin d'exécution du simulateur. Ce fichier stocke les informations qui seront lues et affichées en plus par l'outil sur la trace. Afin de simplifier l'interface de post-traitement, les informations sont écrites dans le *.csv* qui stocke les informations sur l'exécution des tâches, au lieu de générer un fichier annexe. La fonction qui génère ce fichier dans le simulateur FLUSIM est donc modifiée afin de pouvoir afficher sur la trace des métriques indiquant la fin du chemin critique et celle de l'exécution, pratique lorsque certaines tâches isolées sont peu ou pas visibles à cause de la compression de l'affichage. Une métrique indiquant la charge de travail moyenne peut aussi être affichée. Cette charge correspond au temps total de travail de l'ensemble des tâches divisé par le nombre de ressources disponibles. C'est un indice de la qualité de la répartition du travail entre les différentes ressources. Cette dernière métrique est accompagnée de deux autres marquant l'écart type de cette charge. Plus l'écart type est grand, plus le déséquilibre de charge entre les ressources est important. Ces métriques venant d'être décrites sont représentées par des lignes verticales sur la trace.

Chaque processus contient le même nombre de cœur. FLUSIM distribue une certaine charge de travail à chaque processus selon une stratégie donnée. Au sein de ces processus, le travail est redistribué de manière arbitraire entre les cœurs. Sur chaque ligne horizontale de la trace qui correspond à un cœur, l'affichage d'un cercle peut indiquer la charge de travail attribuée au processus du cœur divisée par le nombre de cœur présent au sein du processus. Cela permet d'isoler la distribution du travail entre processus et celle entre cœurs afin d'identifier plus facilement la source potientielle du déséquilibre de charge.

Enfin, la dernière métrique qu'il soit possible d'observer sur la trace est la quantité de période d'inactivité. Cette dernière correspond aux temps d'attente entre deux tâches cumulés pour chaque cœur qui participe à l'exécution. Parfois les traces sont très compressées et il peut être difficile d'évaluer si certaines variables ont permis de modifier l'exécution. Ces quantités de temps passées à attendre sont donc représentées par des tâches grisâtres démarrant toutes au même instant après la métrique verticale qui marque la fin de l'exécution.

3.2.2 Comparaison de trace

Comparer de nouvelles implémentations par rapport aux précédentes nécéssite de passer les deux fichiers d'exécution l'un à la suite de l'autre dans l'outil de post-traitement et de faire des captures d'écran pour ensuite pouvoir les comparer entre elles. Cela peut se réveler peu pratique d'autant plus qu'elles n'ont pas forcément la même échelle en abscisse puisque cette dernière s'adapte en fonction du temps final de l'exécution. Une nouvelle version de l'outil a donc été réalisée, capable de prendre en entrée deux fichiers *.csv* générés à la fin des exécutions respectives et de comparer directement les deux traces correspondantes sur une même échelle.

3.3 Simulateur de maillage 2D et scripts automatisés

3.3.1 De huit millions de cellules à une centaine

Bien que les outils de visualisation précédents aident fortement à la compréhension de l'environnement fourni, il serait utile de pouvoir tester les différents paramètres et nouvelles implémentations sur des graphes de l'ordre d'une centaine de tâche. En effet, confirmer que 2 millions de tâches se sont vues attribuer un valeur correcte par exemple risque de prendre un certain temps.

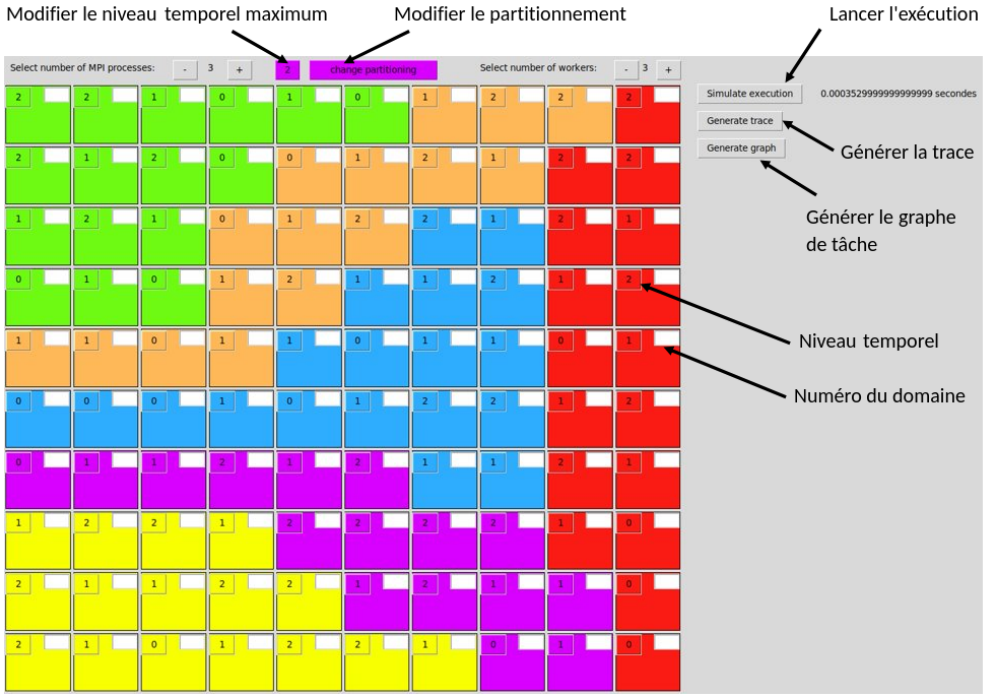


FIGURE 18 – Aperçu de l'outil de visualisation de maillage 2D.

Le nombre de tâche peut être réduit en partie avec le règlage de certains paramètres mais l'ordre de grandeur reste assez proche de celui de départ. Dans notre cas, le graphe est généré à partir d'un maillage 3D de plus de 8 millions de cellules et 26 millions de faces. Il est donc inévitable d'atteindre un certain ordre de grandeur en terme de nombre de tâches produites. Un moyen efficace de réduire drastiquement le nombre de tâche est de partir d'un maillage très petit où idéalement l'exécution peut être réalisée à la main. Cela permet d'une part, de bien comprendre pas à pas le fonctionnement et d'autre part de vérifier aisèment le bon fonctionnement des implémentations.

Un outil graphique a été développé à l'aide de la bibliothèque Python Tkinter et dont la figure 18 en affiche un aperçu. Ce dernier présente un maillage 2D sous la forme d'une grille 10×10. En plus de contenir seulement une centaine de cellule, un maillage en 2D avec des cellules de forme carrée permet de limiter la génération de tâches à cause d'un plus faible nombre de faces.

Il est possible de définir son propre partitionnement intéractivement tout comme le nombre de processus et de cœurs par processus ainsi que le niveau temporel maximum de l'exécution. En effet, la grille peut avoir 4 distributions de niveaux temporels différentes : une avec des cellules de niveaux 0, 1, 2 et 3 si le niveau temporel maximum est $\tau = 3$; une avec des cellules de niveaux 0, 1, 2 si $\tau = 2$

et ainsi de suite. Cet outil respecte les contraintes de la simulation où une cellule d'un niveau donné τ ne peut avoir que des cellules voisines de niveau τ, $\tau - 1$ ou $\tau + 1$. Réduire le niveau maximum permet de générer sensiblement moins de tâche.

Depuis l'interface, il est possible d'appeler directement l'outil de post-traitement et le programme python de visualisation de graphe afin de générer la trace d'exécution et le graphe de tâche. Pour cela, les environnements respectifs à ces deux visualisations ont du être modifiés en ce sens. L'outil de post-traitement est étendu de manière à pouvoir générer et afficher la trace de manière automatisée dans une fenêtre séparée en appuyant simplement sur un bouton de l'interface présentée figure 18. L'outil de visualisation du graphe est simplement étendu en appelant une fonction dans le programme qui lance directement la production d'un *.dot* et la visualisation du graphe correspondant dans une fenêtre séparée.

La difficulté de l'implémentation de ces extensions réside dans la possibilité de lancer toutes ces fenêtres sans bloquer l'outil principal du maillage 2D. Les connexions de ces fenêtres annexes peuvent donc être coupées et une nouvelle exécution peut être lancée sans fermer et recharger l'outil central.

Cet outil tout-en-un constitue donc un préambule aux nouvelles implémentations puisqu'il permet de générer des exécutions sur un maillage de taille modérée produisant au maximum des graphes de l'ordre de 300 à 600 tâches. Le résultat de différentes configurations peut donc être observé au sein même de l'outil en produisant et affichant traces et graphiques sans recourir à des environnements et commandes externes.

3.3.2 Graphiques et scripts automatisés

Des scripts python ont été implémentés afin de délimiter les diverses implémentations et faciliter les exécutions en fixant certains des nombreux paramètres. Enfin, les différents graphiques présents dans ce rapport sont réalisés grâce à des programmes implémentés qui utilisent notamment les bibliothèques Python pandas et matplotlib.

4 Stratégies d'ordonnancements de tâche

Paralléliser le code de simulation numérique avec une approche à base de tâches permet de fournir une exécution des calculs plus souple et donc un parallélisme plus efficace. Le graphe de tâches, selon sa configuration, peut être parcouru de plusieures manières différentes. Certains parcours augmentent le degré de parallélisme plus que d'autres. Deux tâches prêtes sans aucune dépendance entre elles peuvent être exécutées dans n'importe quel ordre. En revanche, exécuter la seconde d'abord peut débloquer plus rapidement des calculs critiques et influencer sensiblement le temps d'exécution. L'application a donc intérêt à privilégier les parcours de graphes efficaces en faisant appel à des stratégies d'ordonnancement de tâches.

L'implémentation de ces stratégies constitue le premier aspect où il est possible d'intervenir afin d'influer sur le temps d'exécution. Ces futures politiques d'ordonnancement, afin qu'elles aient un vrai impact sur le résultat final, doivent appuyer leurs implémentations sur le fonctionnement de l'ordonnanceur présent dans l'application.

4.1 L'ordonnancement de tâche au sein du simulateur FLUSIM

Le rôle d'un ordonnanceur, comme son nom l'indique, est de répartir les tâches d'un graphe entre plusieurs cœurs. Des stratégies d'ordonnancement peuvent être utilisées afin d'influer sur le parcours du graphe en priorisant l'exécution de certaines tâches. FLUSIM utilise l'ordonnanceur du runtime StarPU dont le fonctionnement est présenté figure 19. Cet ordonnanceur est composé d'un ensemble de conteneur. Lorsqu'une tâche devient prête, c'est-à-dire que toutes les tâches dont elle dépend ont été calculées, le support d'exécution utilise l'opération *push* afin de placer la tâche dans un conteneur disponible. Dès qu'un cœur se trouve inactif, il appelle la fonction *pop* qui récupère une tâche prête dans un de ces conteneurs. Les stratégies d'ordonnancement consistent à attribuer à chaque tâche des indices qui indiquent leur niveau de priorité. Les tâches sont donc placées en fonction de leur indice où un conteneur correspond à une priorité. Les cœurs inspectent chaque conteneur par ordre décroissant jusqu'à trouver une tâche disponible.

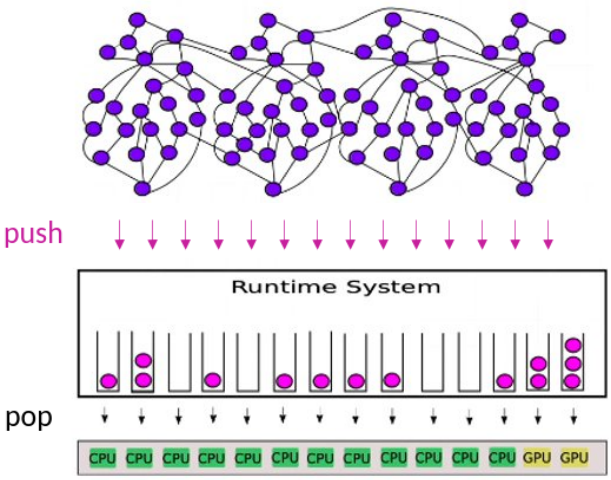


FIGURE 19 – Fonctionnement de l'ordonnanceur StarPU[10].

FLUSIM implémente un ordonnanceur en s'appuyant sur le même principe. L'ensemble des conteneurs s'exprime sous la forme d'un dictionnaire Python, appelé *ready_tasks*. Les clés de ce dictionnaire sont identifiées comme les priorités et les valeurs de ces clés sont des listes où les tâches prêtres sont stockées. La toute première étape de l'ordonnanceur est d'appeler la stratégie d'ordonnancement qui attribue une priorité à chacune des tâches définies. Les priorités sont représentées sous forme d'entier et ajoutées parmi les caractéristiques qui constituent les tâches (chaque tâche est représentée par un dictionnaire). Par la suite, les tâches sans successeur du graphe, constituant les tâches de départ de l'exécution, sont transferées dans la structure *ready_tasks* et l'exécution peut commencer.

Dès que la valeur d'un compteur de dépendances d'une tâche est à 0 (i.e toutes les dépendances de la tâche sont résolues), celle-ci est mise dans la structure *ready_tasks* et placée en fonction de sa priorité. Cela permet aux cœurs inactifs de récupèrer les tâches les plus prioritaires en premier en parcourant la structure dans l'ordre décroissant des valeurs. Ces tâches sont ajoutées et retirées de la structure en appelant les fonctions *push* et *pop*. En conséquence, le code de simulation ne peut exécuter que les tâches qui sont présentes dans la structure *ready_tasks*. Á chaque moment de l'exécution, un certain nombre de tâches est disponible à l'exécution dans cette structure. Ces tâches présentent des priorités variables entre elles : celles ayant les plus élevées sont retirées en premier de la structure.

4.1.1 Stratégie utilisée

Bien qu'il soit possible de lancer l'exécution sans aucune priorité, FLUSIM en propose une qui essaye de compenser l'irrégularité du graphe de tâches induite par l'influence de la méthode d'intégration temporelle. Ce déséquilibre se traduit par un graphe dont la largeur peut varier très fortement lors de son parcours. La figure 20 présente un graphe produit par le code original FLUSEPA. Celui-ci devient de moins en moins large au fur et à mesure que l'on s'approche des dernières tâches.

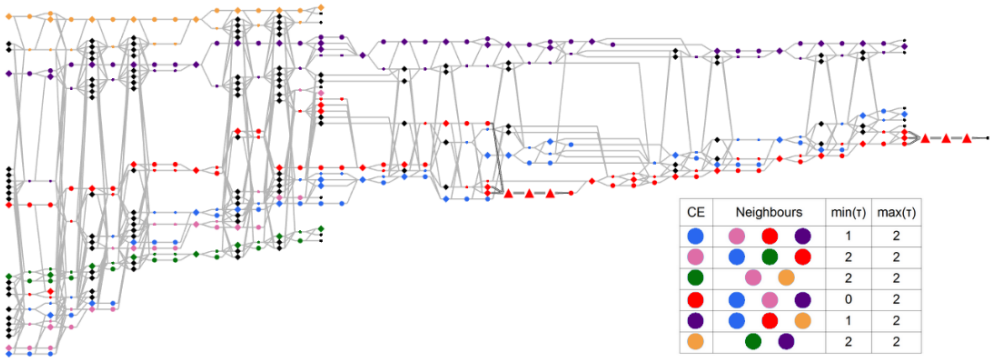


FIGURE 20 – Graphe de tâche obtenu avec FLUSEPA[11] en partitionnant en 6 domaines et $\theta = 2$.

Cela se traduit par des situations potentielles de famine entre cœurs puisqu'il est possible qu'il n'y ait plus assez de tâches prêtes pour nourrir l'ensemble des ressources disponibles à certains moments de l'exécution. Cette situation pénalise fortement l'exécution et peut donc être la source des périodes d'inactivité visibles dans les traces d'exécution.

La politique d'ordonnancement implantée dans le simulateur priorise les tâches sur les branches les plus longues du graphe. La manière dont cette priorité est écrite s'imprime de la manière dont StarPU implémente ces priorités (en particulier celle nommée *prio*). Celles implémentées durant ce stage n'ont pas la même structure puisqu'elles s'appuient directement sur le fonctionnement interne du processus de simulation. Un point important est que les priorités ne se propagent pas d'une tâche à l'autre. Une tâche ayant la plus haute priorité peut être voisine d'une tâche ayant une priorité proche de 0.

Le fonctionnement de la priorité implémentée se décompose comme suit. Les différentes priorités vont être distribuées en fonction du domaine à partir duquel la tâche a été générée. Les domaines possèdant le plus grand nombre de cellules de niveau temporel faible, donc les plus coûteuses ont une priorité maximale arbitraire (par exemple 1000). Les priorités des autres domaines sont calculées en fonction de leur distance par rapport à ceux marqués comme étant les plus prioritaires. Plus la distance est faible, plus la priorité est élevée. Se baser sur la quantité du plus faible niveau temporel au sein d'un domaine pour en extraire une priorité et la propager permet de traiter le plus rapidement possible les tâches extraites des domaines qui vont générer le plus de calculs pendant l'exécution. Ces tâches vont être dépendantes entre elles (au sein d'un même domaine) à cause des accès mémoires. Elles vont donc générer les chaînes les plus longues du graphe.

Théoriquement, les chaînes les plus longues sont celles qui vont ralentir l'exécution puisque ce sont elles qui vont mettre le plus de temps à être calculées. Le but de cette priorité est donc de fournir moins de travail dès le début mais de mieux le répartir au fil de l'exécution. Les cœurs vont traiter les chaînes importantes dès le début, ce qui peut débloquer plus de tâches pour la suite de l'exécution.

4.2 ASAP et ALAP : les deux côtés d'une même face

Cette section marque le début des contributions sur la première approche de ce stage, à savoir influer sur l'ordonnancement des tâches afin de réduire les périodes d'inactivité observées et ainsi obtenir une exécution plus efficace.

Dans un premier temps, deux stratégies d'ordonnancement ont été considérées. Celles-ci se basent sur l'article *A Makespan Lower Bound for the Scheduling of the Tiled Cholesky Factorization based on ALAP Schedule*[6]. L'idée principale de ces deux politiques est d'exécuter les tâches hors chemin critique au plus tôt (ASAP) ou au plus tard (ALAP) pour tenter d'équilibrer les tâches dans le temps en amenant davantage de tâches « prêtes » à des moments où le degré de parallélisme est trop faible. Un peu comme dans un jeu de tetris où il faut caser au mieux les pièces afin de perdre le moins d'espace possible entre celles-ci. La priorité d'une tâche est ensuite déterminée par son placement dans l'exécution. Elles diffèrent donc de la stratégie déjà implémentée par le fait que la priorité est donnée ici à une tâche en fonction de sa position dans une exécution optimale déterminée par certains critères. La stratégie *prio* s'appuie sur des caractéristiques propres à la géométrie du domaine.

4.2.1 Contributions communes

Dans l'article, les graphes de tâche ont un seul nœud de départ et un seul de fin. En revanche, les graphes de simulation de mécanique des fluides peuvent présenter plusieurs nœuds sans prédécesseur ou sans successeur. En adaptant les stratégies présentées dans l'article directement, il est possible de tomber sur des maximas locaux et donc d'avoir des priorités fausses. La première étape

d'implémentation est donc d'insérer deux tâches *fantômes* dans le graphe. Une qui relie les nœuds de départ sans prédécesseur et une qui relie les nœuds finaux sans successeur. Leurs durées sont mises à zéro afin de ne pas influencer le temps d'exécution final.

Si une priorité est attribuée à une tâche en fonction de son propre temps d'exécution et d'autres critères définis par la stratégie d'ordonnancement utilisée, toutes les tâches vont avoir une priorité différente. En effet, les temps d'exécution sont représentés par des floats et ont un ordre de grandeur de 10^{-6} au minimum. La structure *ready_ tasks* stocke les tâches en fonction de leur priorité. La taille de la structure correspond donc au nombre de priorités différentes présentes dans le graphe de tâches. Disons qu'une exécution est composée de plus de 2 millions de tâches, toutes avec une priorité différente. La structure *ready_ tasks* est parcourue en ordre décroissant, donc de la plus haute priorité à la plus faible. En fonction de la stratégie (notamment ASAP), il est possible que plus l'exécution progresse vers les tâches de fin, plus les priorités aient des valeurs faibles. En conséquence, à chaque fois qu'un cœur va se trouver inactif, le dictionnaire de plus de 2 millions de clé va devoir être parcouru jusqu'à la fin pour aller chercher les tâches de plus faible priorité et donc situées le plus "loin" dans la structure. Les exécutions du simulateur FLUSIM deviennent aussi longues que celles du code d'origine FLUSEPA. Cette problématique est résolue en définissant, dans l'implémentation de nos politiques, un intervalle de priorité « raisonnable » dans lequel les valeurs calculées pour chaque tâche sont projetées et converties en format entier.

4.2.2 ASAP : As soon as possible

Dans l'article, l'idée est que si l'on dispose d'un nombre infini de cœurs, les tâches doivent être théoriquement exécutées à l'instant même où elles deviennent prêtes et le temps d'exécution correspond donc au chemin critique. Ce dernier est défini, dans notre cas, comme étant la séquence de tâche la plus longue, en terme de temps d'exécution, qui doit être exécutée en séquentiel. Les tâches sont traitées les unes à la suite des autres et ne peuvent pas être exécutées simultanèment. Le chemin critique est donc considéré comme une borne inférieure : le temps d'exécution ne peut pas descendre en dessous. En présence d'une limitation de ressources, l'idée est de donner une priorité à une tâche en fonction de sa distance à la dernière tâche du graphe. Les tâches les plus prioritaires sont donc celles qui sont le plus éloignées de la fin de l'exécution en terme de temps et correspondent aux tâches situées sur les branches les plus longues du graphe. Ces tâches doivent donc être exécutées le plus tôt possible.

Dans l'article, le chemin critique d'une tâche est considéré comme le chemin le plus long de cette tâche à la dernière tâche du graphe. En revanche, dans le code de FLUSIM, la fonction *compute_ critical_ path* calcule, pour chaque tâche, le plus long chemin entre la tâche et la première du graphe. La stratégie d'ordonnancement ASAP est donc implémentée de la manière suivante. Une nouvelle fonction est définie qui calcule cette fois le chemin critique de chaque tâche d'après la définition de l'article, c'est-à-dire la distance de la tâche donnée par rapport à la dernière tâche du graphe. Pour cela, cette fonction parcourt le graphe en partant de ce dernier nœud. La valeur attribuée à une tâche correspond à la somme de sa durée d'exécution prédite par le simulateur et du chemin critique le plus élevé trouvé parmi ses successeurs (dans le sens original du graphe). Une fois que le chemin critique de chaque tâche est calculé, ces valeurs sont converties pour être placées dans un intervalle de priorités. Ce dernier est défini ici en fonction du plus long chemin critique calculé dans le graphe et de la priorité maximale qui peut être attribuée. Cette valeur est mise à 1000. Sur la base de divers tests, il s'est avéré qu'à partir d'un certain seuil, augmenter cet intervalle n'apporte plus à l'exécution (à part ralentir le calcul des priorités).

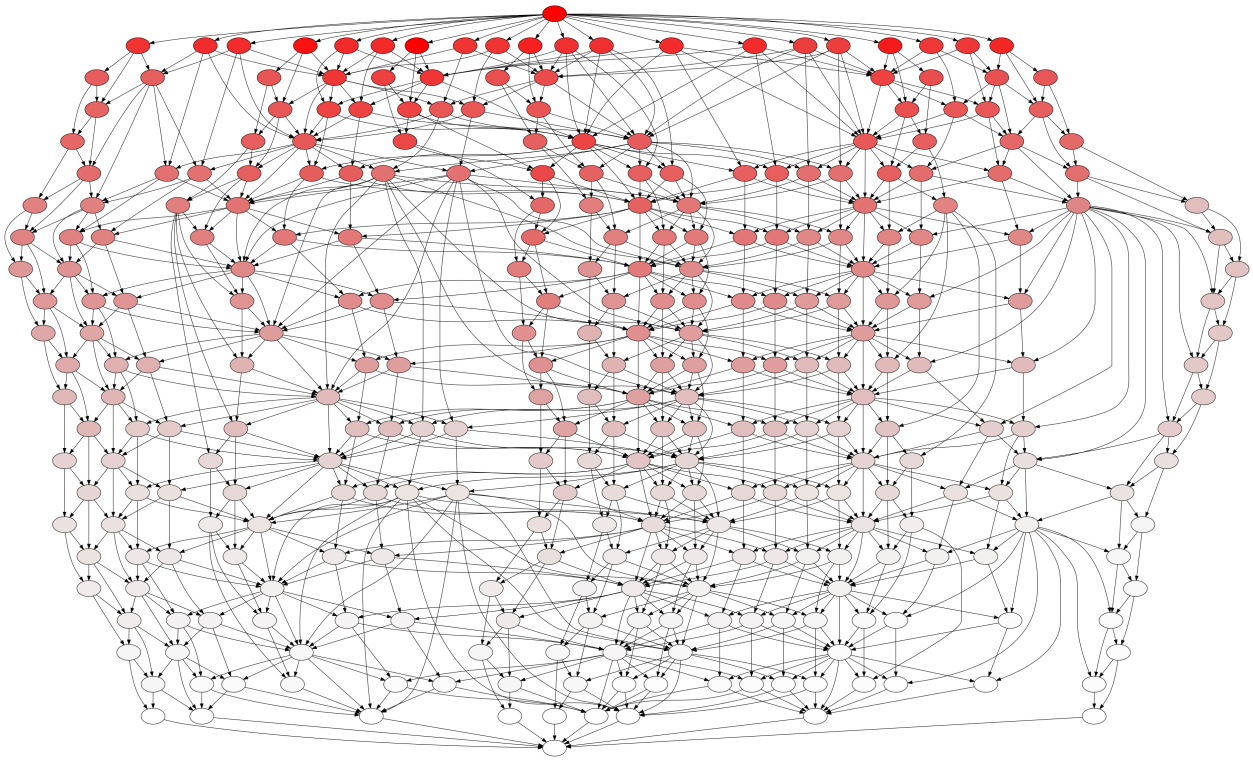


FIGURE 21 – Graphe de tâche du maillage 2D exécuté avec la priorité ASAP et colorié en fonction de la priorité de la tâche.

Il est possible de reconnaître, sur la figure 21, le graphe de tâche généré par le maillage 2D de 100 cellules et présenté dans les sections précédentes. Une première observation est la présence des deux tâches fantômes reliant tous les nœuds de début et de fin du graphe. Les tâches sont coloriées dans un dégradé de rouge en fonction de la priorité qui leur a été attribuée. Puisque le graphe est parcouru en partant de la fin, la valeur des priorités des tâches augmentent au fur et à mesure (comme les couleurs) puisque les sommes de chemins critiques s'accumulent. Cette visualisation confirme aussi que les politiques d'ordonnancement ne modifient pas le graphe mais influencent seulement son exécution sur les cœurs.

Attention cependant ! Cette représentation peut laisser entendre qu'en descendant vers la fin du graphe, les tâches n'aient plus de priorités. ASAP calcule les priorités sur l'ensemble du graphe mais toutes les tâches ne sont pas prêtes en même temps. La figure 22 affiche ce même graphe où les tâches ont exactement la même priorité que celle de la première figure. Cependant les tâches sont coloriées en fonction de leur priorité par rapport à celles des tâches qui se situent sur le même étage. En effet, on suppose que les tâches situées sur une même ligne horizontale risquent de se retrouver en concurrence dans la structure *ready_ tasks*. Chaque étage possède donc une variété de priorité et ce sur l'entièreté du graphe.

Les figures 23a, 23b et 23c donne une idée plus précise du fonctionnement de la priorité. Un graphe de test se voit ajouter des tâches fantômes (en losange gris) pour éviter les maximas locaux puis la priorité de chaque tâche est calculée. La figure 23b souligne le chemin critique calculé par ASAP d'une tâche et explique la valeur de celle-ci à 4 et non pas 3 puisque c'est sur le successeur ayant le chemin critique le plus élevé que se base le calcul de la tâche donnée. Enfin, la figure 23c

correspond à la trace d'exécution de ce graphe sur deux unités de calcul.

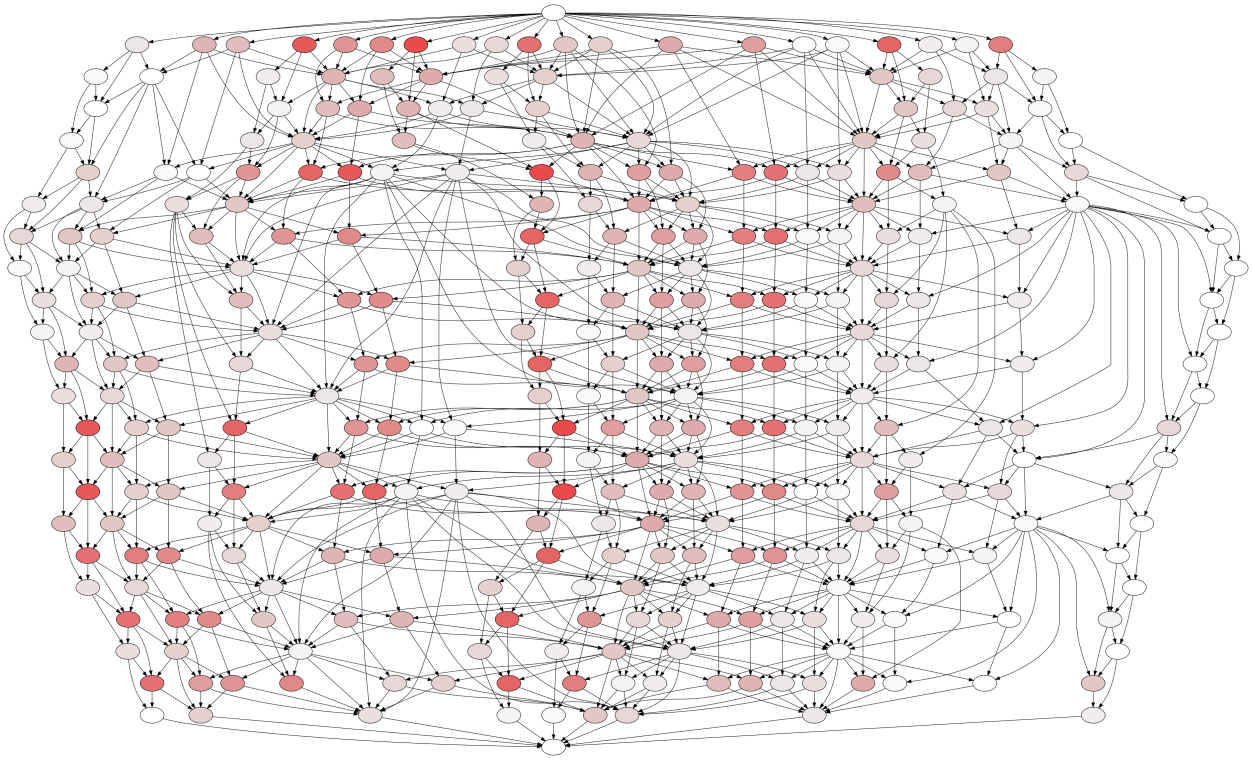


FIGURE 22 – Graphe de tâche du maillage 2D exécuté avec la priorité ASAP et colorié en fonction du rang de la priorité de la tâche sur l'étage du graphe où elle se trouve.

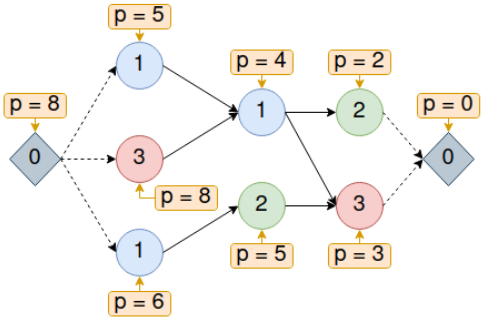
4.2.3 ALAP : As late as possible

Une autre approche, qualifiée sous le nom de *As Late As Possible*, est présentée dans l'article. Celle-ci ordonnance les tâches en considérant non pas le point de départ de l'exécution mais la fin, au contraire des approches habituelles. Comme un tetris inversé, où les dernières pièces sont placées en premier sur la limite définie par le jeu. Dans l'article, l'idée est de renverser le graphe (les successeurs deviennent des prédécesseurs et inversement) et d'ordonnancer les tâches dès qu'elles sont prêtes dans le cas où l'on dispose d'un nombre infini de cœurs.

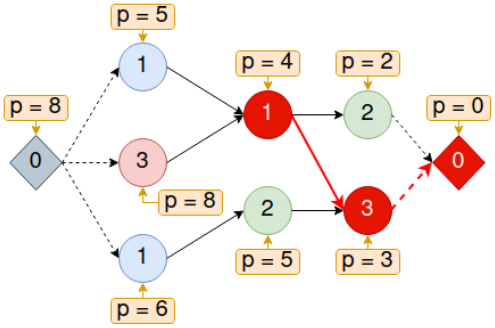
En pratique, la stratégie ALAP consiste à appliquer la stratégie ASAP sur le graphe inversé. L'implémentation se déroule comme suit. Dans un premier temps, le graphe original est renversé comme précédemment. Au lieu de lancer le graphe original avec les valeurs des priorités extraites du graphe inversé comme la stratégie ASAP, c'est ce dernier qui va être directement passé en paramètre de l'exécution. Celle-ci va appeler la stratégie d'ordonnancement ASAP implémentée précedemment sur ce nouveau graphe. En sortie, on obtient donc notre *.csv* qui stocke toutes les informations liées à l'exécution des tâches. Les colonnes de ce fichier qui correspondent au temps de départ et de fin sont modifiées de manière à ce que les tâches soient placées en partant de la fin de l'exécution. Cela permet d'obtenir la trace finale d'exécution avec le graphe original sur lequel la priorité ALAP a été appliquée.

Le fonctionnement d'ALAP peut être observé plus en détails sur les figures 23d, 23e et 23f. De

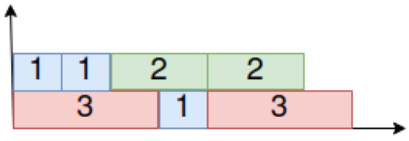
manière similaire à ASAP, le graphe test se voit ajouter des tâches fantômes. La priorité ASAP est donc appliquée sur le graphe inversé et ce dernier est passé en paramètre de l'exécution. De celle-ci on obtient la trace du haut de la figure 23f qui est ensuite transformée pour obtenir l'exécution théorique du graphe de départ avec la priorité ALAP. La figure 23e indique le chemin critique de la même tâche que pour l'exemple d'ASAP. De manière similaire, la priorité est de 4 puisque la durée de la tâche est de 1 et le chemin critique le plus long parmi ses successeurs est de 3.



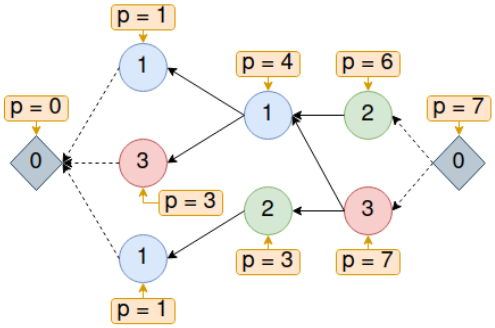
(a) Graphe avec priorité ASAP



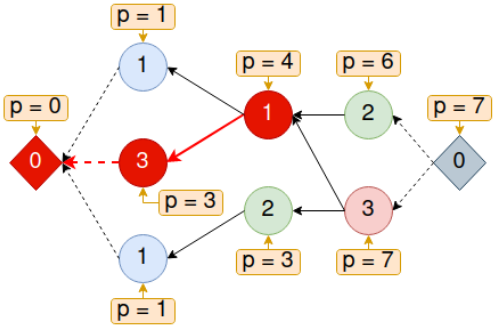
(b) Chemin critique d'une tâche avec ASAP



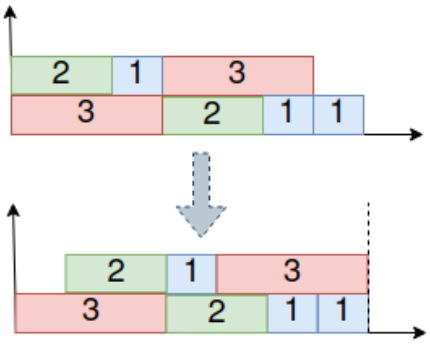
(c) Trace de l'exécution du graphe avec ASAP



(d) Graphe avec priorité ALAP



(e) Chemin critique d'une tâche avec ALAP



(f) Trace de l'exécution du graphe avec ALAP

FIGURE 23 – Fonctionnement schématisé des priorités ASAP (à gauche) et ALAP (à droite).

4.2.4 Expériences et résultats

Maitenant que ces deux nouvelles politiques d'ordonnancement implémentées ont été décrites, leur efficacité et bon fonctionnement doivent être déterminés par rapport à une exécution originale avec la priorité déjà fournie par le simulateur FLUSIM et décrite plus haut. Dans un premier temps, les priorités sont analysées sur un cas minime puis par la suite sont appliquées à une configuration formant un compromis idéal entre limite du simulateur et situation sur architectures réelles. Certains paramètres qui ne sont pas mentionnés dans ce rapport sont fixés de manière à se rapprocher le plus d'une exécution type du code de simulation FLUSEPA.

Les figures 24 et 25 correspondent à une configuration identique où le maillage est partitionné en 128 domaines et réparti entre 16 processus de 4 cœurs chacun (64 cœurs au total). 75.545 tâches sont générées, +2 pour les tâches fantômes. Chaque figure présente une comparaison entre une exécution avec la politique de base de FLUSIM (trace du bas) et une exécution avec la nouvelle priorité (trace du haut). Les tâches sont coloriées en fonction de la sous-itération où elles ont été générées. On retrouve le comportement décrit dans les sections précédentes où la première sous-itération génère le plus de tâches et les suivantes plus ou moins en fonction du niveau temporel maximum. Les grandes périodes d'inactivité aussi mentionnées tout au long de ce rapport sont bien observables. Trois métriques principales sont affichées. En rouge, la charge totale de travail divisée par le nombre de ressources disponibles, entourée de ses deux métriques de l'écart-type. Très rapprochées sur les deux traces, ces métriques indiquent que l'équilibre de charge entre les domaines est de qualité. La métrique bleue indique le chemin critique, temps idéal de l'exécution et ici assez éloigné de la métrique de la fin de l'exécution. Les traces d'exécutions les plus rapides affichent une métrique supplémentaire après celle qui marque la fin de leur exécution. Celle-ci correspond à la fin de la seconde exécution et met en valeur l'intervalle de temps qui a été gagné. Une première observation est que le chemin critique, dans les deux figures, est identique entre les deux traces comparées. Cela s'explique par le fait que le graphe généré est identique, seul l'ordre des calculs se modifie.

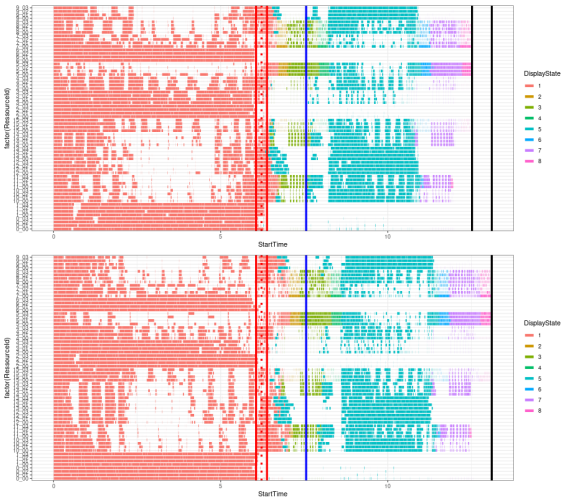


FIGURE 24 – Trace d'exécution avec la stratégie ASAP +4.5128%.

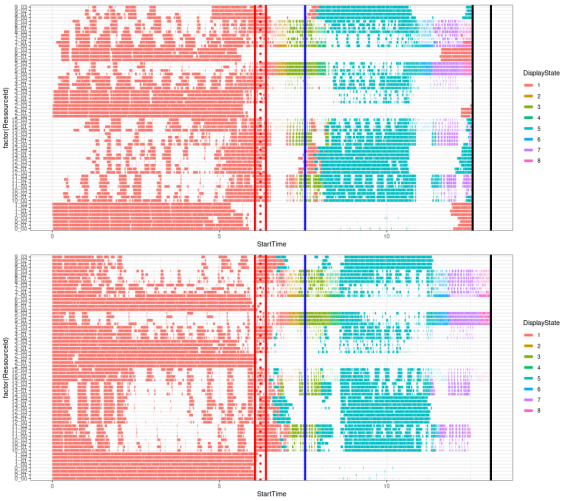


FIGURE 25 – Trace d'exécution avec la stratégie ALAP +4.1750%.

La priorité ASAP appliquée à ce cas minime gagne 4.5128% par rapport à la priorité de base. Ce gain est observable sur la trace du haut de la figure 24, comme celui de 4.1750% obtenu grâce

à la priorité ALAP sur la figure 25. Un point intéressant de relever est que sur la trace d'exécution d'ALAP, certaines tâches de la toute première sous-itération sont exécutées vers la fin. Leurs dépendances et cette priorité *As Late As Possible* ont permis ce placement de tâche.

Ces résultats sont donc prometteurs. Cependant, avant de les appliquer sur une configuration plus importante, il peut être intéressant de regarder la distribution des valeurs de ces priorités. Les figures 26 et 27 confirment que les deux stratégies implémentées couvrent bien l'ensemble des valeurs comprises entre 0 et 1000 de l'intervalle de priorités et ne distribuent pas seulement une ou deux valeurs. En effet, avoir un trop grand nombre de tâches ayant une priorité identique entre elles risque de rendre la priorité caduque. Fort heureusement, ce n'est pas le cas ici, la distribution est assez homogène.

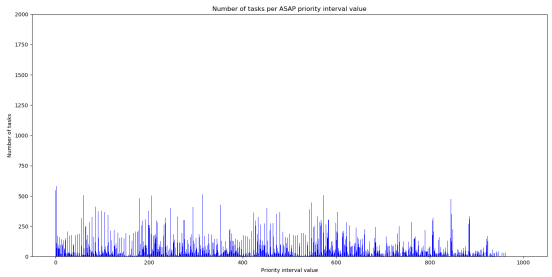


FIGURE 26 – Distribution des valeurs de la priorité ASAP sur l'ensemble des tâches.

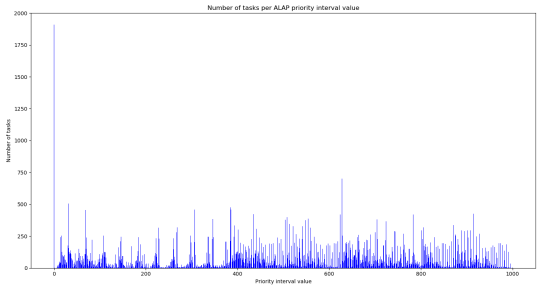


FIGURE 27 – Distribution des valeurs de la priorité ALAP sur l'ensemble des tâches.

Les figures 28 et 29 présentent les gains obtenus grâce aux deux nouvelles politiques appliquées à une configuration plus importante. Le maillage est découpé en 1024 partitions distribuées entre 128 processus ayant 4 cœurs chacun (512 cœurs au total). 556.740 tâches sont générées pour l'exécution de base mais +2 pour les nouvelles priorités à cause des tâches fantômes. ASAP obtient un gain de 6.03% et ALAP 3.35%. ALAP est toujours meilleure que l'exécution de base mais dépasse rarement ASAP en terme de gains. Une explication peut être que la stratégie ALAP lance l'exécution directement sur le graphe original inversé. Cependant, le processus de simulation de FLUSIM orbite autour du graphe généré. Modifier le graphe après la génération et le passer en paramètre de l'exécution peut donc impacter certains paramètres d'exécution réglés sur le graphe original. Une autre possibilité est qu'effectuer les tâches le plus tard possible réduit les chances de débloquer des tâches critiques et donc d'obtenir un parcours du graphe optimal.

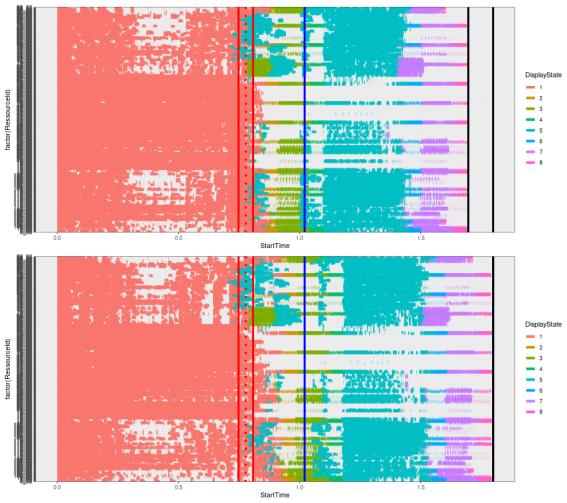


FIGURE 28 – Trace d'exécution avec la stratégie ASAP 6.03%.

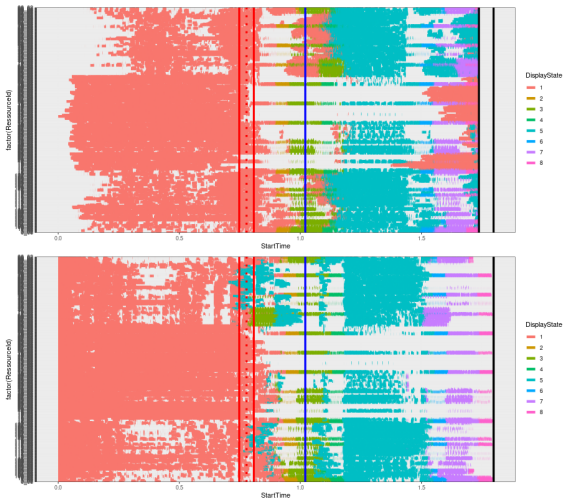


FIGURE 29 – Trace d'exécution avec la stratégie ALAP 3.35%.

4.3 Priorité aux faces externes distribuées

Avant d'implémenter une stratégie d'ordonnancement qui s'appuie sur les discrétisations spatiales et temporelles de FLUSIM, le ou les points de contention du graphe doivent être identifiés afin de déterminer les tâches à prioriser.

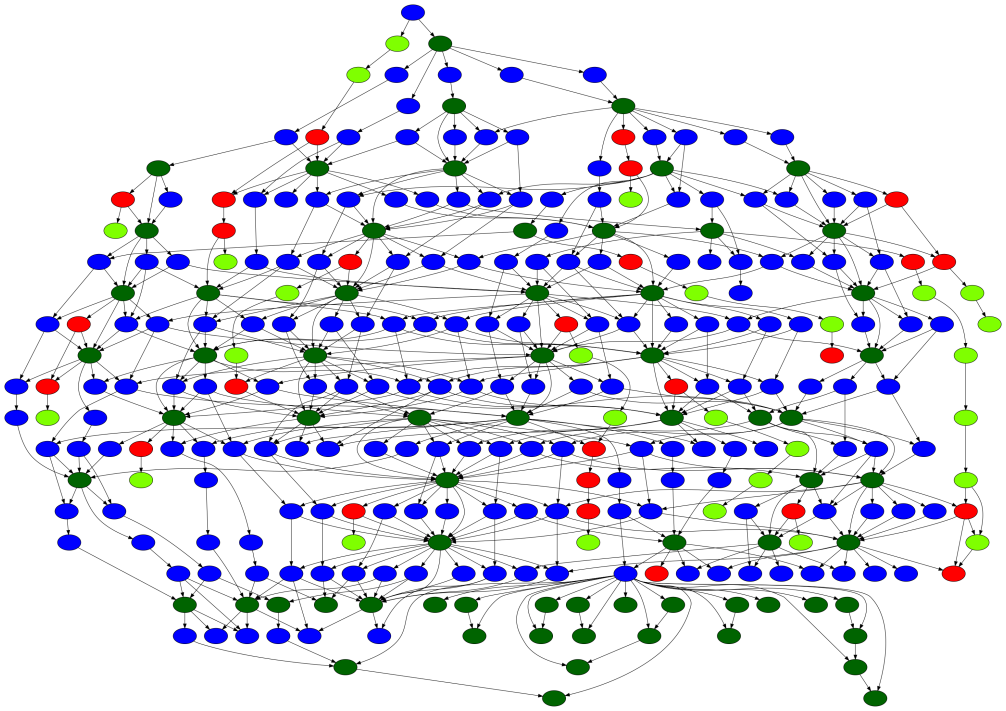
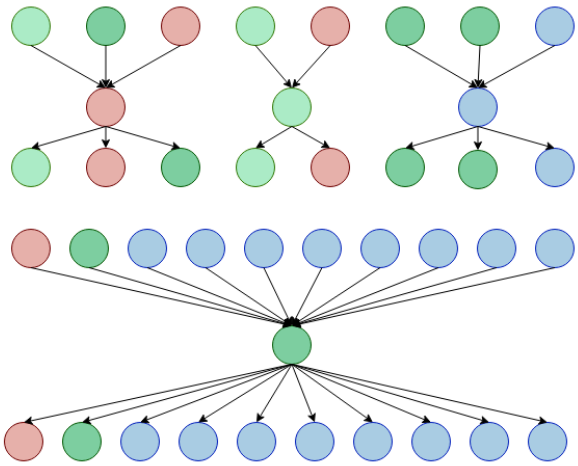


FIGURE 30 – Extrait d'un graphe généré à partir du maillage original où les tâches sont coloriées en fonction de la composante qu'elles calculent.

Les points de contention correspondent à des tâches qui possèdent beaucoup de dépendances, donc beaucoup d'arêtes, que ce soit en terme de prédécesseur ou successeur. D'après le graphe représentatif de la figure 30, ces points de contention correspondent aux tâches vertes foncées qui traitent les cellules de bords. Les dépendances de ces tâches viennent majoritairement des tâches bleues foncées qui traitent les faces de bords. Celles-ci n'ont pas un nombre excessif d'accès mémoire par rapport aux autres composantes mais sont présentes en quantité importante dans les accès mémoires des cellules de bord. Ces dernières, en plus de deux accès mémoires fixes, nécéssitent l'ensemble des faces de bord du domaine où elles se situent mais aussi celles des domaines voisins. La figure 31a présente les accès mémoires définis au sein de FLUSIM et donc les dépendances pour les différents types de tâche. Les cellules de bords nécéssitent le nombre d'accès mémoires le plus important dont la majorité sont aux faces de bords. Pour un domaine ayant quatre voisins, la figure 31b donne une idée des dépendances générées pour chaque tâche spécifique.

Task type	Memory Access
fii	Ci_i(R), Cb_i(R), Fii(RW)
fij	Cb_i(R), Cb_j(R), Fij(RW)
c+b	Fii(R), Cb_i(RW), foreach neigh of i as vFvi(R); Fiv(R)
c+i	Ci_i(RW), Fii(R)

(a) Accès mémoire en fonction du type de tâche dont les dépendances de celles-ci découlent directement



(b) Schématisation des dépendances des quatre types de tâche pour un domaine ayant 4 voisins

FIGURE 31 – Accès mémoire définis au sein de FLUSIM et les dépendances en résultant pour les différents types de tâche.

Parmi les tâches qui traitent ces faces de bords, toutes ne vont pas être priorisées, permettant ainsi d'éviter que trop de tâches aient une priorité proche et donc rendre la stratégie caduque. La figure 33 affiche, une fois encore, le petit graphe généré par le maillage 2D où les tâches présentes dans le même rectangle grisâtre sont calculées sur le même processus. Cette visualisation distingue deux couleurs différentes, jaune et verte, représentant toutes deux des tâches traitant les faces externes. Cependant, les tâches jaunes situées en bordure de rectangle, correspondent à des tâches qui traitent les faces externes dont les domaines de bordure sont attribués à des processus différents. Leurs dépendances, mises en valeur sur la figure 32, ne sont donc pas locales. Des messages devront être envoyés ou reçus entre processus pour satisfaire ces dépendances. Dans la perspective d'appliquer ces stratégies sur le vrai code de production, où le coût des communications est pris en compte, effectuer ces tâches en priorité permettrait de mieux couvrir les temps de transfert. De plus, les traiter le plus rapidement possible permet dans notre cas de débloquer plus rapidement des points

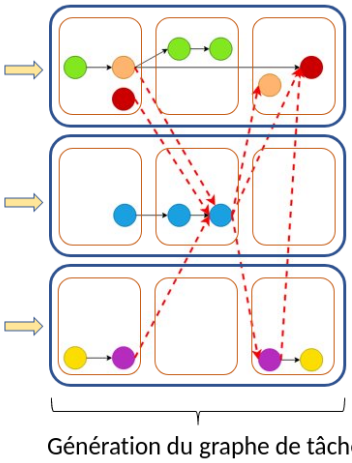


FIGURE 32 – Schématisation de la dernière étape du processus de fonctionnement de FLUSIM où les dépendances des tâches à prioriser sont mises en valeur.

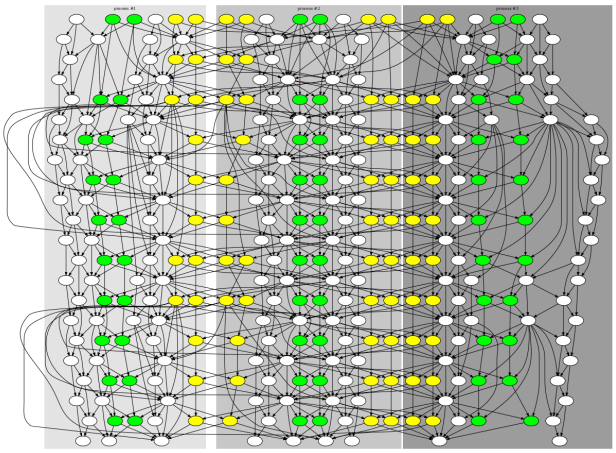


FIGURE 33 – Graphe de tâche du maillage 2D exécuté sur trois processus. Les tâches des faces de bords sont coloriées en fonction de leur caractère hautement prioritaire ou non.

de contention situés sur d'autres processus. Ce sont donc ces tâches spécifiques qui sont ciblées dans la nouvelle politique d'ordonnancement.

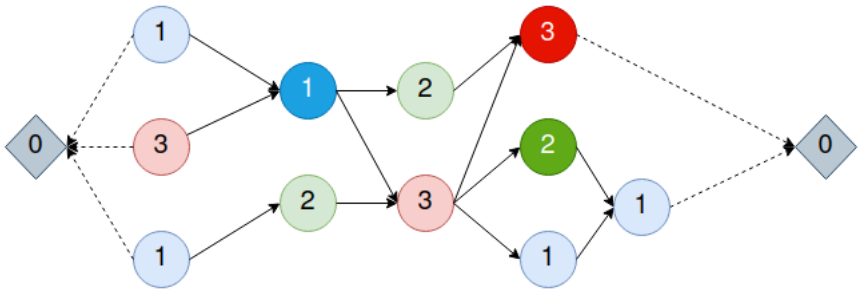
Ces tâches traitant les faces externes dont les domaines sont distribués sur des processus différents sont définies comme tâche hautement prioritaire. Afin de les débloquer au plus vite, chaque tâche du reste du graphe est priorisée en fonction de sa distance éventuelle à la prochaine tâche hautement prioritaire rencontrée dans le graphe. Á noter qu'un chemin plus prioritaire en écrase un autre, mais les tâches spécifiques conversent leur priorité maximale.

L'implémentation de cette nouvelle politique se compose comme suit. La fonction qui calcule les valeurs des priorités des tâches s'inspire de l'implémentation de la fonction *compute_critical_path* de FLUSIM. Comme vu dans l'implémentation des priorités précédentes, cette fonction calcule le plus long chemin critique de chaque tâche à partir de sa distance à la première tâche du graphe. En revanche, ici le chemin critique d'une tâche va être calculé en fonction de la distance à la prochaine tâche hautement prioritaire rencontrée.

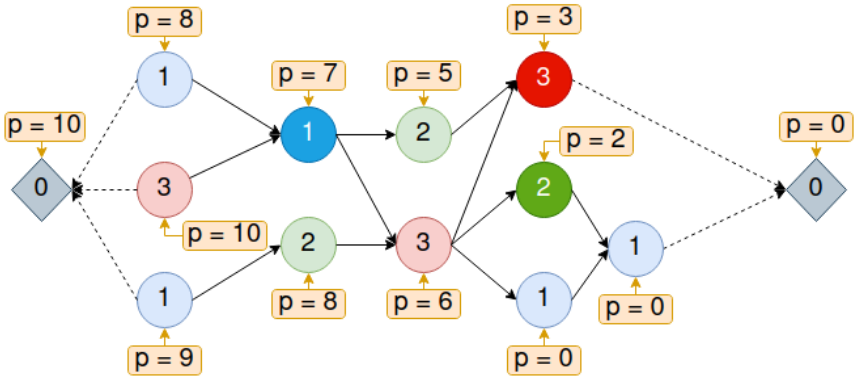
Le graphe est donc parcouru en partant des nœuds de fin. Dans un premier temps, chaque tâche qui traite une face externe dont les domaines sont sur des processus différents se voit donner la priorité maximale, sinon une priorité de 0. Par la suite, chaque nœud récupère la valeur du chemin critique la plus élevée parmi ses successeurs. Si les successeurs ne sont pas à 0, cela signifie que la tâche actuelle mène à une tâche hautement prioritaire et doit donc être priorisée en fonction. Á l'inverse, si tous les successeurs sont à zéro et que la tâche elle-même n'est pas une tâche spécifique alors il n'est pas nécéssaire de la prioriser. Cela permet d'éviter d'avoir un trop grand nombre de tâche priorisée et de perdre l'effet de la stratégie. Une fois la fonction appelée, l'ensemble de ces valeurs est converti, comme pour les priorités précédentes, sur un intervalle défini au préalable.

La figure 34 expose shématiquement le processus de cette nouvelle stratégie. Un nouveau graphe de test se voit attribuer des tâches fantômes et les valeurs des nœuds correspondent à une durée théorique. La figure 34b présente l'étape d'attribution des priorités aux nœuds en fonction de

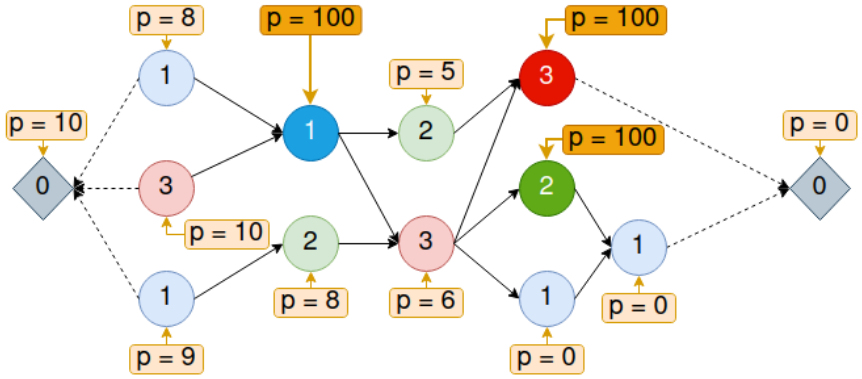
leur distance la plus importante qui mène à une tâche hautement prioritaire. Enfin, ces dernières se voient donner une priorité maximale, comme sur la figure 34c, afin de garantir qu'elles soient immédiatemement exécutées dès qu'elles deviennent prêtes.



(a) Example de graphe où les tâches dont la priorité doit être maximale sont mises en valeur



(b) Attribution des priorités en fonction de leur chemin à la prochaine tâche de priorité maximale



(c) Attribution des priorités maximales aux tâches spécifiques

FIGURE 34 – Processus de fonctionnement de la priorité aux faces externes distribuées implémentée.

4.3.1 Expériences et résultats

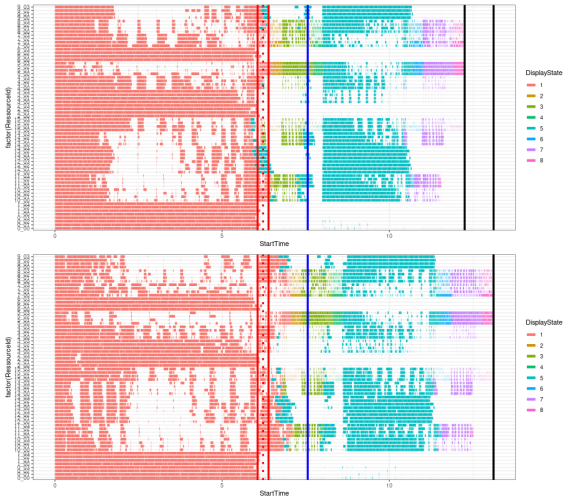


FIGURE 35 – Trace d'exécution avec la stratégie aux faces externes distribuées +6.5598%.

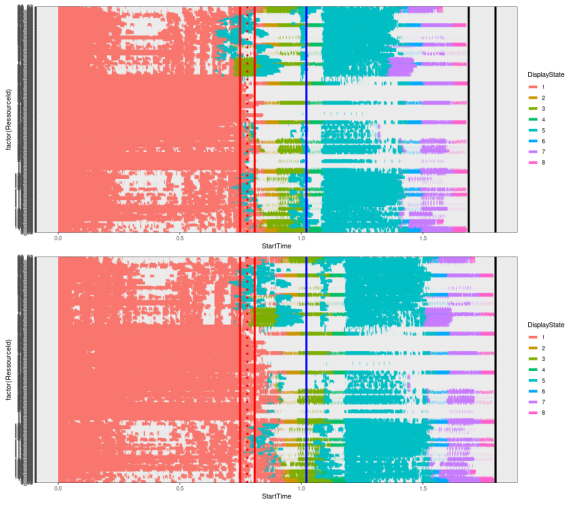


FIGURE 36 – Trace d'exécution avec la stratégie aux faces externes distribuées +6.12%.

De manière similaire aux autres priorités implémentées, cette nouvelle politique d'ordonnancement est comparée à celle de base proposée dans FLUSIM. La priorité est analysée sur les deux mêmes configurations que precèdemment. Une amélioration de +6.5598% est obtenue en appliquant cette priorité qui vise les points de contention sur la plus petite configuration figure 35 (i.e. 128 partitions sur 64 cœurs divisés en 16 processus de 4 cœurs). Il est possible de constater que l'ordre d'exécution des tâches a bien été impacté par cette stratégie. En effet, les tâches de la première sous-itération en particulier, présentent un motif différent entre les deux traces.

La distribution des valeurs de la priorité sur l'ensemble des tâches est aussi assez homogène. La figure 37 indique un nombre de tâches conséquent ayant une priorité de 0, indiquant que les tâches n'ont bien pas été toutes priorisées. Á noter que les tâches hautement prioritaires représentent 47% des tâches totales (dont la valeur n'est pas incluse dans la figure 37).

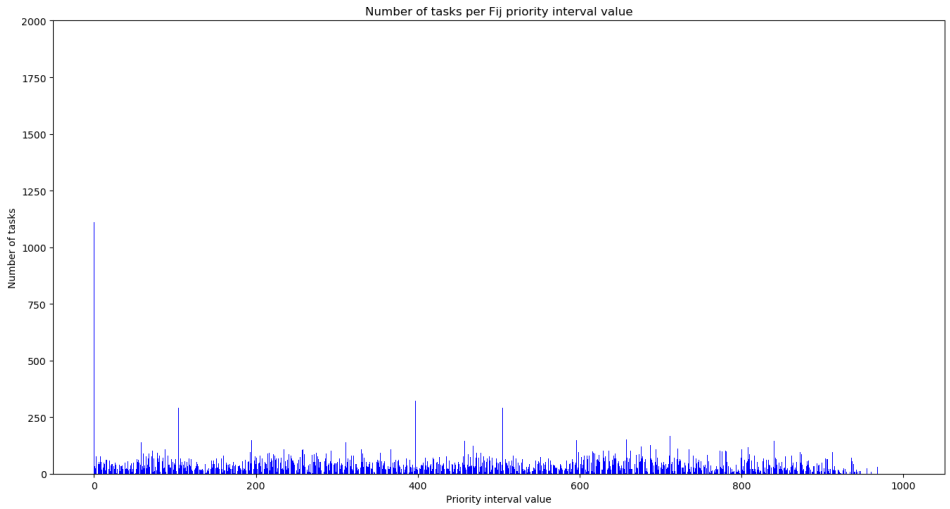


FIGURE 37 – Distribution des valeurs de la priorité aux faces externes distribuées sur l'ensemble des tâches.

Appliquer cette priorité qui semble prometteuse sur une configuration générant 556.742 tâches réparties entre 128 processus de 4 cœurs chacun (512 cœurs) permet d'obtenir un gain de 6.12%. La stratégie d'ordonnancement qui priorise les tâches en s'appuyant sur les points de contention du graphe semble fournir les meilleures exécutions. La différence peut s'expliquer par le fait que seule cette stratégie d'ordonnancement se base directement sur les caractéristiques du graphe de tâches. En conséquence, c'est cette priorité qui est intégrée à toutes les expériences dans la suite de ce rapport.

5 Décomposition de domaine

Bien qu'influencer l'ordonnancement des tâches sur les différents processus fournit des résultats intéressants, les traces d'exécutions présentent toujours des larges périodes d'inactivité. Ce premier levier permet donc d'influencer des aspects de l'exécution qui ne sont pas décisifs dans l'apparition de ces périodes. Afin de résoudre la plupart des problématiques existantes autour de l'exécution de graphe de tâche sur architectures distribuées, les approches consistent majoritairement à implémenter des stratégies d'ordonnancement pour placer aux mieux ces tâches dans l'exécution. La particularité ici est que les graphes de tâche peuvent être modelés d'une certaine manière en influant directement sur leur processus de génération. En effet, au lieu de prendre en entrée un graphe déjà fixé comme la plupart des applications, FLUSIM récupère un maillage et génère le graphe par la suite. Ce graphe est issu à la fois de l'intégration temporelle adaptative et de la décomposition de domaine. Modifier l'intégration temporelle n'est pas à l'ordre du jour, puisqu'il s'agit de la méthode que nous souhaitons paralléliser efficacement. Par contre, il est possible d'agir sur la décomposition de domaine.

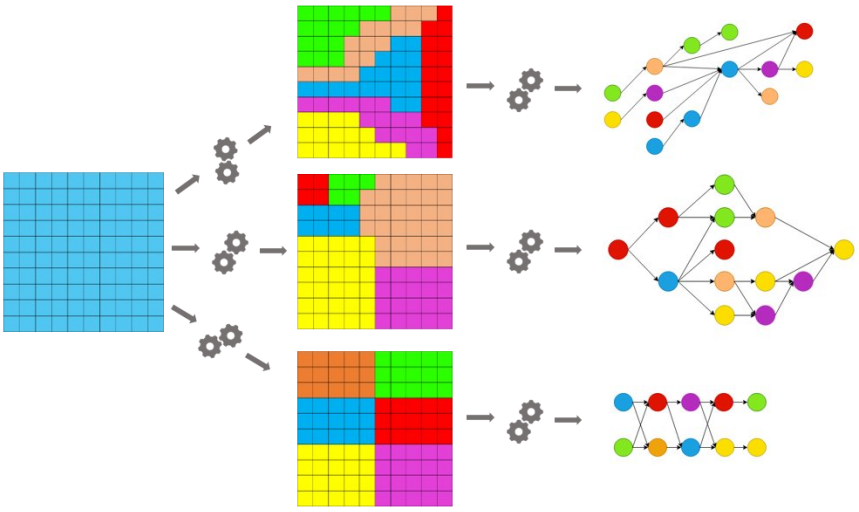


FIGURE 38 – Modifier la décomposition de domaine génère un graphe de tâche totalement différent.

5.1 Décomposition de domaine avec la bibliothèque Scotch

Avant d'aborder les différentes implémentations, rappelons un peu comment s'organise la décomposition de domaine et son influence sur la graphe. FLUSIM prend en entrée un maillage composé de cellules et de faces et appelle la bibliothèque Scotch afin de le partitionner. Cela permet de regrouper les cellules et faces en un ensemble de domaines, ensuite distribué entre les processus.

Le partitionnement initial de Scotch (celui appliqué sur le graphe réduit dans l'algorithme multi-niveau) s'appuie sur un seul paramètre pour découper le maillage. Dans notre cas, le paramètre fait en sorte que chaque partition du maillage ait un même coût opératoire. Ce coût est calculé en fonction du niveau temporel de la cellule. Plus la cellule a un niveau temporel faible, plus elle est considérée comme coûteuse. En effet, c'est elle qui a la fréquence de calcul la plus haute et qui, en conséquence, génère le plus de calculs durant l'exécution. Lorsque le maillage de 8 millions de cellules et 26 millions de faces est partitionné, obtenir des domaines équilibrés en coûts mais très

déséquilibrés en terme de dimension est une situation courante. Cela est dû au fait que, d'une part, les niveaux temporels des cellules évoluent progressivement au sein du maillage. L'ensemble des cellules $\tau = 3$ compose essentiellement la bordure du maillage et il faut traverser les cellules de $\tau = 2$ puis celles de $\tau = 1$ pour trouver celles de $\tau = 0$. Cette caractèristique est observable sur la figure 39 où la couleur des mailles informe sur leur niveau temporel respectif τ. Les cellules où $\tau = 3$ forment une couche épaisse sur l'extérieur du maillage, englobant la pièce interne centrale. Cette même pièce se caractérise comme le cœur du phénomène, concentrant l'ensemble des niveaux temporels définis. D'autre part, Scotch se repose sur la topologie géométrique du maillage où les cellules d'une même partition ne peuvent pas former plusieurs foyers mais un seul continu. La conséquence de ces deux caractèristiques est observable sur la figure 40.

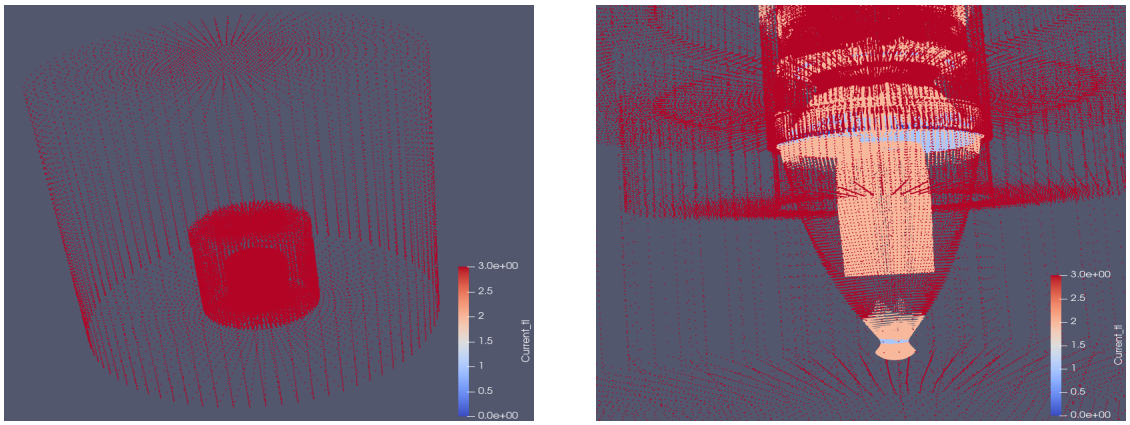


FIGURE 39 – Maillage colorié en fonction du niveau temporel de chaque maille.

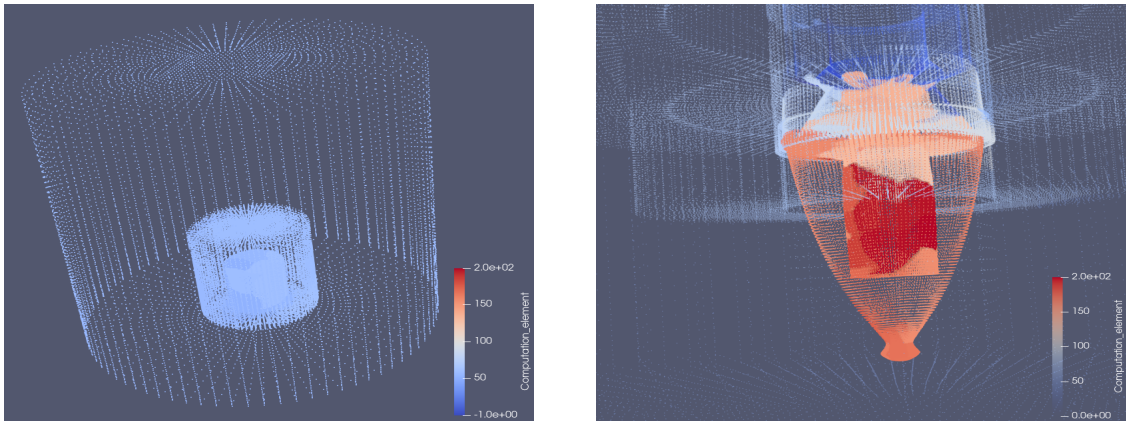


FIGURE 40 – Maillage colorié en fonction du domaine d'appartenance après un partitionnement réalisé avec Scotch équilibrant les coûts opératoires entre domaines.

Scotch compense les coûts par des domaines de taille très disparate entre eux. Ce partitionnement peut donc présenter une toute petite partition contenant très peu de cellules extrêmements coûteuses et une autre contenant des millions de cellules ayant un coût très faible. Les cellules de niveau temporel 3 sont englobées dans un même grand domaine d'après la figure 40. La vue de la pièce centrale présente au contraire une variété de couleur sur une surface réduite du phénomène. Cela indique la présence de domaine de plus faible dimension et donc en concentration plus importante puisque la même surface doit être couverte. La figure 41 se focalise sur la pièce centrale après un partitionnement Scotch. La dimension des domaines diminue progressivement à mesure que l'on se rapproche du centre puisque les cellules deviennent de plus en plus coûteuses.

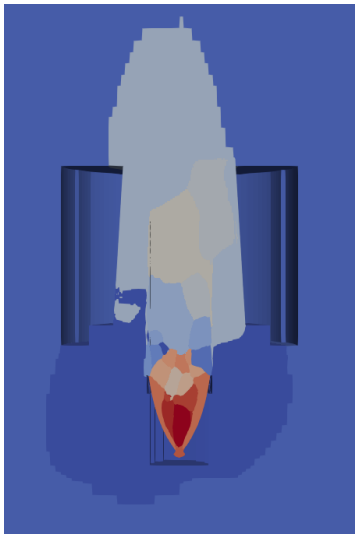


FIGURE 41 – Pièce interne du maillage partitionnée et coloriée en fonction du domaine d'appartenance de chaque maille.

5.2 Domaines équilibrés et périodes d'inactivité

Modifier le partitionnement, c'est modifier la génération du graphe de tâche. D'après les sections précédentes, la bibliothèque Scotch réalise un partitionnement bien équilibré des coûts opératoires en s'appuyant sur son unique critère. Pourtant, les traces d'exécution revèlent des périodes d'inactivité.

Intéressons-nous plus en détails aux traces d'exécution fournies en sortie du simulateur. Une observation assez flagrante est que certaines sous-itérations produisent beaucoup plus de travail que d'autres. Comme précisé dans les sections précédentes, le code de simulation détermine des niveaux temporels maximum propres à chaque sous-itération puisque toutes les mailles n'ont pas la même fréquence de calcul. Seules les tâches dont le niveau temporel n'excède pas le maximum local de la sous-itération sont générées. La figure 42 indique ces niveaux temporels à ne pas dépasser pour chaque sous-itération au sein d'une exécution classique avec 2^3 sous-itérations. L'ensemble des expériences de ce rapport adopte ce schéma.

	1	2	3	4	5	6	7	8
max	3	0	1	0	2	0	1	0

FIGURE 42 – Niveau temporel maximum permis pour chaque sous-itération.

Les périodes d'inactivité observées viennent peut-être de cette disparité de travail entre sous-itérations, provoquant des famines soudaines sur les processus. Passer de la première sous-itération qui traite l'ensemble des cellules du maillage à la suivante qui traite seulement les cellules $\tau = 0$ peut insérer dans la structure *ready_ tasks* un nombre de tâche inférieur au nombre de ressources disponibles. Modifier ces niveaux temporels maximum et distribuer les calculs du maillage équitablement entre les sous-itérations n'est pas possible. Cela revient à casser l'implémentation de la

méthode temporelle adaptative ainsi que l'ordre des calculs entre sous-itérations, les rendant faux.

Malgré ces éléments fixes, il est possible que le partitionnement actuel amplifie ce déséquilibre. En effet, la nuance du partitionnement de Scotch est que certains domaines ne vont contenir que des cellules d'un niveau temporel donné. En conséquence, les fonctions *foreach* ne vont générer aucune tâche de ces partitions lors de certaines sous-itérations. Un domaine ne contenant que des cellules où $\tau = 2$ ne va générer aucune tâche pour la sous-itération 4 de la figure 42 par exemple. Certains processus peuvent se voir attribuer des domaines qui ne génèrent plus de calcul pour une sous-itération donnée, créant une situation de famine à un instant t. Et ce même si les domaines sont répartis entre les processus de manière à équilibrer parfaitement la charge de travail entre ces derniers. Cependant, l'aspect le plus pénalisant vient directement du processus de génération des tâches. Prenons un grand domaine contenant l'ensemble des cellules $\tau = 3$ du maillage. Afin de calculer les cellules internes par exemple, 1 seule tâche très coûteuse en terme de temps d'exécution va être générée pour les calculer toutes. Ce partitionnement produit donc des chaînes de tâches plus compliquées à ordonnancer, le graphe est moins souple.

Le critère qui semble jouer une part importante est la répartition des niveaux temporels entre domaines. Afin de pallier simultanément ces aspects pénalisants, une idée est d'obtenir des domaines équilibrés en terme de coût opératoire, mais surtout en terme de répartition des niveaux temporels. C'est-à-dire que chaque domaine contient le même nombre de cellule de niveau temporel 0, 1, 2 et 3.

5.2.1 Effet théorique d'un équilibrage de coût opératoire et de niveau temporel

L'ajout d'un seul autre critère dans le partitionnement initial de l'approche multi-niveau peut influer sur un certain nombre d'aspect de l'exécution. En particulier, dans le cas de la répartition des niveaux temporels. Remplacer l'équilibrage des coûts par celui des niveaux temporels en premier critère n'est pas possible. La forme d'entrée du critère principal ne permet pas de préciser à Scotch : tous les domaines doivent avoir la même quantité de cellules $\tau = 3$, $\tau = 2$, $\tau = 1$ et $\tau = 0$.

La figure 43 présente en fonction du partitionnement d'un maillage, le motif de tâches générées. Seules les tâches traitant les cellules internes sont considérées. Le partitionneur de gauche imite le comportement de Scotch avec un critère d'équilibrage des coûts. Celui de droite correspond à un partitionneur multi-critère équilibrant coût opératoire et répartition des niveaux temporels. En ordonnée se trouvent les cinq premières sous-itérations (sur huit).

La première observation pouvant être faite est que faire en sorte que chaque domaine possède une quantité relativement identique de l'ensemble des niveaux temporels existants dans le maillage permet de générer des tâches plus fines en quantité plus importante. En effet, le processus de génération de tâche distingue les niveaux temporels, générant une tâche pour traiter chacun d'entre eux au sein d'un domaine. Ces tâches contiennent moins d'éléments puisque la quantité de cellule du domaine est divisée entre les valeurs de τ. Ces tâches plus fines en temps d'exécution et plus nombreuses permettent, d'une part, un parcours de graphe plus flexible et donc un éventuel meilleur ordonnancement. D'autre part, la quantité plus importante de tâche permet de mieux nourrir les cœurs au sein des processus. Plusieurs cœurs se voient chacun attribuer une petite tâche au lieu d'un seul cœur avec un tâche très coûteuse. Pour une sous-itération donnée, une même ligne du tableau donne une même quantité de travail mais exprimée différemment. La répartition du travail entre processus, déjà équilibrée par Scotch, serait mieux distribuée aux cours de l'exécution.

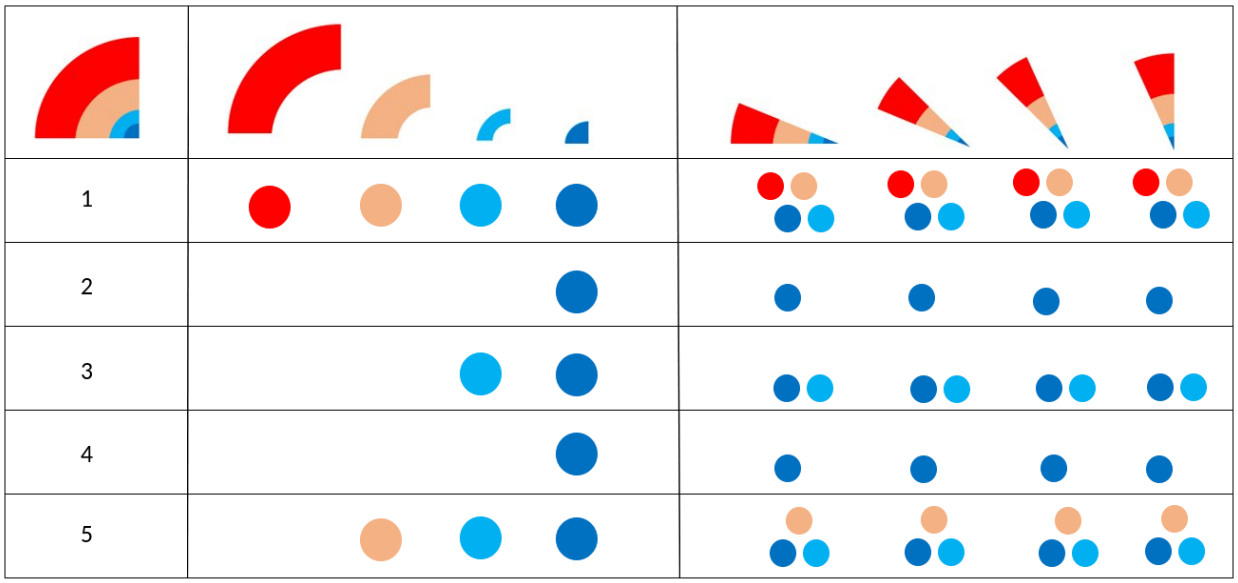


FIGURE 43 – Différence de tâches générées pour chaque domaine entre un partitionnement mono-critère qui imite le comportement de Scotch et un partitionnement supposé multi-critère pour les cinq premières sous-itérations (3, 0, 1, 2) du schéma temporel.

La seconde observation est que chaque domaine va générer le même motif de tâche à chaque sous-itération. Sur la ligne de l'itération 3, les domaines rouges et jaunes ne génèrent aucune tâche et seuls les deux autres vont travailler pendant cette sous-itération. Selon la configuration initiale et la distribution des processus, il peut y avoir un risque de rendre des paquets de cœurs inactifs. Et ce, même si chaque processus a une charge de calcul identique. En revanche, le partitionnement de droite possède des domaines qui génèrent les mêmes tâches entre eux à chaque sous-itération. Une fois ces domaines distribués entre les processus, cet équilibre de tâche peut limiter les effets de l'intégration temporelle adaptative. Il est donc possible que ce partitionnement limite les périodes de famine de travail et les effets du déséquilibre de charge entre sous-itérations, réduisant les périodes d'inactivité de la trace d'exécution.

La décomposition de domaine doit présenter un équilibrage des coûts opératoires et des différents niveaux temporels entre partitions. Deux critères sont donc à fournir en entrée du partitionneur. Or passer le second critère à Scotch pose problème. En effet, la bibliothèque est mono-critère et cette unique place est déjà occupée par un équilibrage des coûts opératoires entre partitions.

5.3 État de l'art

Au sein de la sphère du partitionnement multi-critère, deux branches principales peuvent être distinguées. La première correspond à des librairies et logiciels directement créés dans ce but et pouvant être intégrés dans des codes de production. La librairie Zoltan[14] développée par le laboratoire national Sandia en constitue un exemple. Celle-ci implémente un mode multi-critère en passant par des algorithmes de partitionnement géométrique. Ces derniers s'appuient directement sur les coordonnées physiques des données du maillage pour le découper. Ils sont connus pour être rapide en terme d'exécution mais fournissent des partitions de qualité modérément bonne. La manière dont est implémentée l'extension multi-critère de cette librairie ne permet pas de l'appliquer à nos types

de maillage. Dans notre cas, cette méthode risque de casser la connexité du maillage.

La seconde branche contient les partitionneurs mono-critères qui ont été étendus de manière à obtenir un comportement multi-critère, en modifiant notamment les différentes étapes de l'approche multi-niveau (contraction, partitionnement initial et expansion). Ce sont les outils les plus courants dans cette sphère. La plupart des travaux correspondent à l'implémentation d'heuristiques qui cherchent à intégrer des critères de partitionnement supplémentaires dans l'approche multi-niveau. Deux aspects sont à retenir de ces travaux. D'une part, ces critères supplémentaires peuvent être considérés malgré tout comme secondaires. En effet, dans les implémentations décrites, le graphe est toujours partitionné avec le premier critère principal. Les autres critères servent à raffiner localement les partitions obtenues. D'autre part, ces critères secondaires sont partiellement fixés par leurs heuristiques. En effet, ces dernières sont exclusivement dédiées à minimiser les coûts de communications entre processus puisque ces questions de partitionnement multi-critère prennent souvent place dans les problématiques liées aux simulations numériques s'exécutant en mémoire distribuée. Ces coûts sont donc à prendre en compte dans le partitionnement afin d'obtenir des partitions limitant l'apparition de ces communications. La marge d'action de l'utilisateur par rapport à ces critères supplémentaires consiste à déterminer la tolérance permise par ses critères secondaires. Cela correspond, dans la phase de raffinement par exemple, à regarder si la valeur d'une solution obtenue par le mouvement d'un nœud d'une partition à l'autre rentre dans le seuil de tolérance. Si ce n'est pas le cas, la solution risque de maximiser les communications par exemple et le mouvement n'est pas effectué.

Les travaux de Rémi Barat présentent en particulier un prototype en langage python appelé PIERE [4] (Partitionnement Initial Équilibré et Raffinement Équilibré) qui implémente ce partitionnement multi-critère essayant de minimiser les communications. Certains de ces travaux présentent une modification directe du partitionnement initial avec plusieurs critères et sont en cours d'intégration dans la prochaine version de Scotch. Les travaux d'Astrid Casadei [7] modifient aussi les différentes étapes de l'algorithme multi-niveau. En revanche, ces derniers sont plus orientés vers une heuristique qui équilibre le partitionnement en fonction du parallélisme disponible. C'est-à-dire obtenir un partitionnement qui tient compte que le travail va être distribué entre processus MPI (premier niveau de parallélisme), puis une seconde fois entre les cœurs de calcul au sein des processus (second niveau de parallélisme).

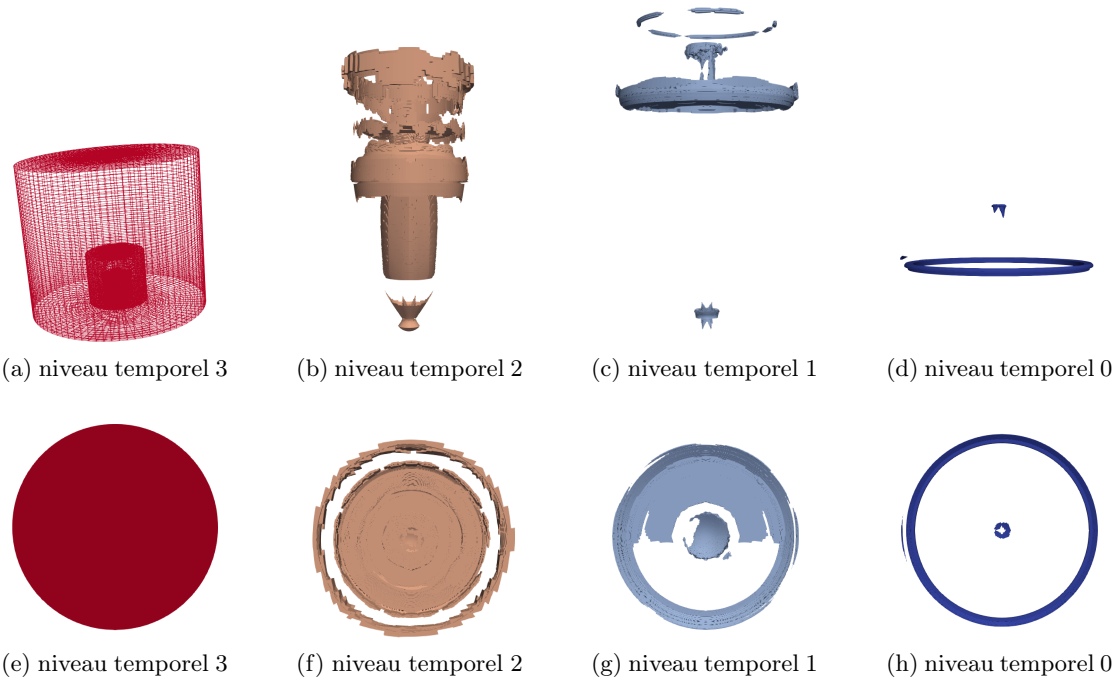
5.4 Partitionneur multi-critère implicite

Il est possible de conclure de l'état de l'art que, d'une part, le partitionneur géométrique multi-critère qui aurait pu être intégré dans FLUSIM ne s'adapte pas aux maillages spécifiques d'entrée. D'autre part, les heuristiques permettant d'étendre un outil mono- en multi-critère et décrites dans les travaux de recherche à ce sujet, raffinent une partition déjà existante réalisée avec le seul critère principal. De plus, ces heuristiques ciblent spécifiquement pour la plupart la réduction des communications entre processus, étant très coûteuses dans une exécution en mémoire distribuée.

Une solution idéale serait de modifier l'étape du partitionnement initial de l'algorithme multi-niveau implémenté dans la bibliothèque Scotch. De cette manière, la découpe s'appuie sur plusieurs critères principaux, en particulier sur les deux qui nous intéressent à savoir le coût opératoire et le niveau temporel des cellules. Pour un stage d'un peu moins de six mois, cela constitue une entreprise risquée surtout qu'il n'existe aucune garantie que cela résolve le problème des périodes d'inactivité.

5.4.1 Géométrie et symétrie

Les visualisations du maillage obtenues avec l'outil Paraview, dont quelques-unes sont intégrées dans le rapport, ont permis de constater que le maillage fourni par *Airbus* possède une répartition relativement symétrique des niveaux temporels. La figure 44 expose l'organisation globale de ces niveaux temporels au sein du maillage. Ce dernier est composé de quatre niveaux temporels où $\tau = 3$ est usuellement représenté par la couleur rouge, $\tau = 2$ par la couleur saumon, $\tau = 1$ par du bleu clair et enfin $\tau = 0$ par du bleu foncé. Les figures 44a, 44b, 44c et 44d correspondent à un aperçu de chacun de ses niveaux, extraits du maillage afin d'observer en détails leur configuration dans l'espace. En terme d'échelle, un niveau temporel d'une colonne de la figure englobe tous les niveaux qui se trouvent à sa droite. Les figures 44e, 44f, 44g et 44h correspondent à une vue du haut du maillage. Il est possible d'observer que seules les mailles de $\tau = 1$ présentent une forme asymétrique sous cet angle.



(a) niveau temporel 3 (b) niveau temporel 2 (c) niveau temporel 1 (d) niveau temporel 0

(e) niveau temporel 3 (f) niveau temporel 2 (g) niveau temporel 1 (h) niveau temporel 0

FIGURE 44 – Configuration spatiale de chaque niveau temporel extrait du maillage original.

Le maillage fourni par *Airbus* présente donc une symétrie assez forte en terme de niveau temporel vue de haut. Cette symétrie nous permet d'implémenter un partitionneur multi-critère implicite. Celui-ci s'appuie directement sur les coordonnées physiques du maillage pour répartir les cellules et faces entre plusieures partitions. En fonction des coordonnées de la cellule, cette dernière est directement placée dans un domaine spécifique.

Découpé à la manière de part de gâteau, le maillage peut, de part sa symétrie de niveau temporel sur l'axe vertical, fournir des domaines équilibrés à la fois en terme de coût opératoire et de niveau temporel. Ce partitionneur géométrique, en s'appuyant seulement sur la position des mailles dans l'espace, permettrait de satisfaire les deux critères d'équilibrage.

5.4.2 Implémentation d'un partitionneur géométrique

La réelle complexité de cette implémentation est la manière dont l'ensemble des informations d'un maillage 3D complexe est traduit sous forme de structure Python afin de pouvoir être interprété par le code de simulation. FLUSIM s'appuie sur une structure de donnée centrale nommée *computation_ elements*, dont la taille correspond à la quantité de cellule présente dans le maillage. Chaque indice de cette structure correspond à une cellule où le numéro du domaine auquel elle appartient est stocké. Cette structure, passée en paramètre de la fonction de partitionnement, est actualisée par Scotch. Elle contient donc les nouvelles partitions.

Le maillage est représenté sous forme de structures imbriquées. La difficulté est de récupérer à partir de la position d'une cellule dans *computation_ elements*, les coordonnées polaires X, Y, Z de celle-ci. Au sein des structures de stockage du maillage, une cellule correspond à une continuité de huit *cases* qui stocke les nœuds de la maille hexaèdre. En revanche, une cellule correspond à une *case* dans *computation_ elements*. Plusieurs aspects sont à prendre en compte. D'une part, toutes les cellules dont le domaine est de valeur -1 dans *computation_ elements* ne doivent pas être prises en compte dans le découpage. D'autre part, le maillage est constitué de deux *zones* différentes. Les cellules de ces zones doivent être accèdées et parcourues séparemment. Ces dernières n'ont pas la même dimension et accèdent parfois à une même structure de donnée, que ce soit directement des champs d'une structure Python ou bien au sein d'une liste stockée par ces structures, à partir d'indices différents.

Tous ces éléments nécéssitent de faire attention à ce que les bonnes données soient accèdées afin de bien répartir les cellules entre domaines et d'éviter les erreurs de segmentations, c'est-à-dire d'accèder à des régions de la mémoire qui ne sont pas réservées à nos structures. Dans les premières implémentations de la fonction, les frontières entre domaines étaient très irrégulières et certaines cellules d'un même domaine formaient plusieurs foyers au sein du maillage comme constaté figure 45. Ces soucis étaient majoritairement dus à un mauvais parcours et accès des données du maillage.

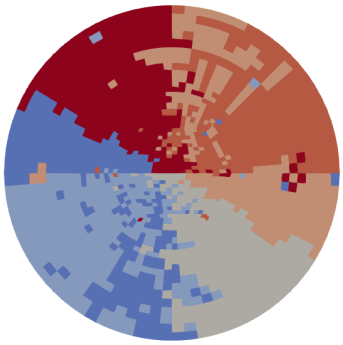


FIGURE 45 – Vue de haut du maillage 3D partitionné en six domaines où les données ne sont pas bien accèdées et les domaines mal attribués.

Ce nouveau partitionneur prend en entrée la structure *computation_ elements* déjà initialisée et actualise les indices afin d'attribuer les cellules aux domaines voulus, de manière similaire à la fonction de partitionnement utilisant Scotch. Á noter que l'appartenance des faces aux différents domaines est déterminée dans la suite de l'exécution en fonction du partitionnement de leurs cellules correspondantes. Ces faces ne sont pas donc considérées dans le partitionneur géométrique. Le principe de ce dernier est de décomposer le cylindre en un nombre de tranche verticale comme exposé sur la figure 46. Ces tranches constituent les domaines du maillage.

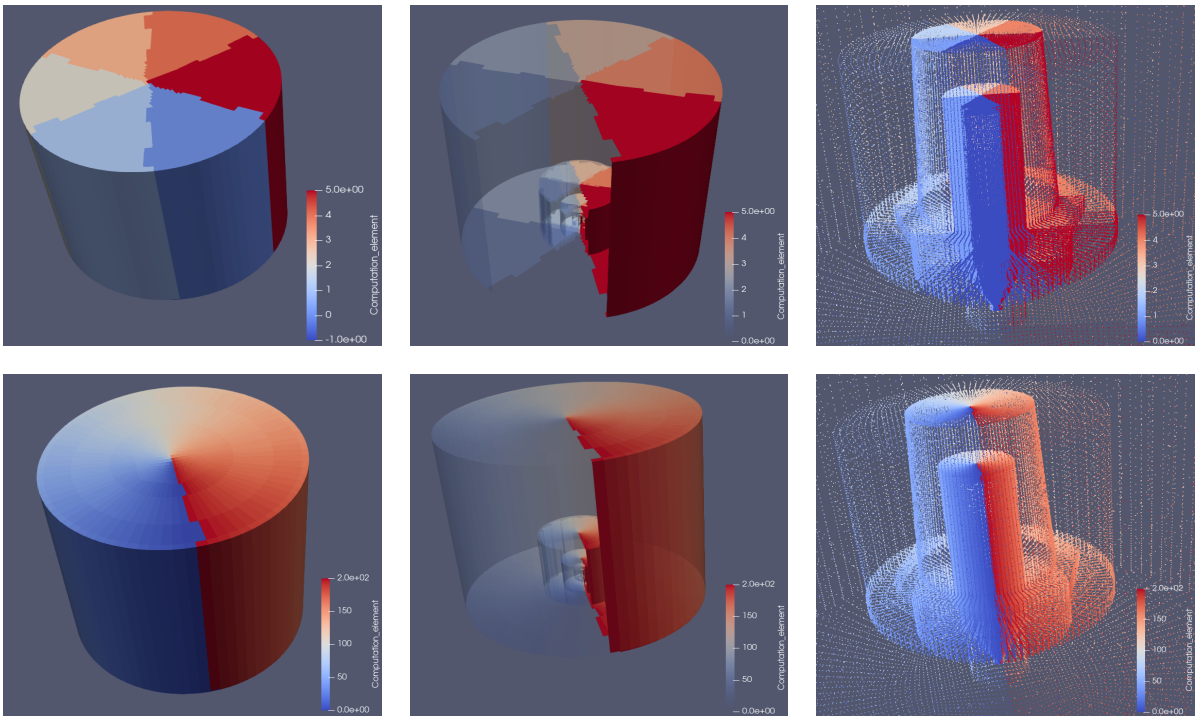


FIGURE 46 – Partitionnement géométrique du maillage en 6 et 200 domaines où les mailles sont coloriées en fonction de leur domaine d'appartenance.

La figure 47 présente un extrait de la fonction implémentée. La boucle englobante itère sur l'ensemble des cellules du maillage (donc sur plus de huit millions d'éléments). Les deux *zones* sont parcourues séparemment au sein de cette boucle principale. Á noter que l'extrait de code ne propose que celui de la première zone. Dans les deux cas, les cellules dont les domaines ont pour valeur -1 dans la structure *computation_ elements* ne sont pas traitées et resteront avec cette valeur. Elles ne seront pas prises en compte pendant l'exécution. De plus, la condition ligne 3 vérifie aussi l'appartenance de la cellule à la zone abordée dans le bloc.

```
1 for (int i = 0; i < dd->num_cells; ++i)
2 {
3     if (dd->computation_elements[i] != -1 && i <= dd->zones[0].size[1])
4     {
5         float Z = dd->zones[0].coord_z[dd->zones[0].connectivity[i * 8]-1];
6         float Y = dd->zones[0].coord_y[dd->zones[0].connectivity[i * 8]-1];
7         dd->computation_elements[i] = get_slice(Z, Y, number_of_domains);
8     }
9 }
```

FIGURE 47 – Extrait de code qui attribue un domaine à chaque cellule de la première zone du maillage en fonction de ses coordonnées polaires.

Les lignes critiques 5 et 6 récupérent les valeurs des coordonnées polaires en fonction de l'indice de la cellule. Les mailles sont héxaèdres et sont donc constituées de huit nœuds. Chaque zone du maillage possède les champs *coord_ x/y/z* qui stockent l'ensemble des positions polaires des nœuds de la zone. Parmi ces *8 millions de cellules* × *8 nœuds*, il faut pouvoir, à partir de la position d'une maille dans *computation_ elements*, déterminer l'indice à passer dans les champs *coord_ x/y/z* afin d'accèder aux coordonnées du premier nœud qui la compose. Pour cela, le champ *connectivity* est utilisé. Ce champ, dont un est défini pour chaque zone, stocke les nœuds associés aux mailles. Les huit premiers éléments correspondent à la première maille, les huit suivants à la seconde et ainsi de suite. En passant l'indice de la maille dans *computation_ elements* à ce champ, on obtient l'indice du premier nœud de la maille. Cet indice est passé en paramètre des champs *coord_ x/y/z* pour ainsi obtenir les coordonnées d'une maille dans l'espace.

La figure 48 indique le code correspondant aux lignes 5 et 6 de la figure 47 afin de pouvoir, cette fois, attribuer les mailles de la seconde zone dans des partitions. Il est aisé de constater que les accès mémoires diffèrent.

```
1 float Z = dd->zones[1].coord_z[dd->zones[1].connectivity
2                                                 [(i -  dd->zones[0].size[1]) * 8]-1];
3 float Y = dd->zones[1].coord_y[dd->zones[1].connectivity
4                                                 [(i -  dd->zones[0].size[1]) * 8]-1];
```

FIGURE 48 – Code qui diffère du premier par le fait que ce sont les cellules de la seconde zone qui sont parcourues dans la fonction de partitionnement.

Déterminer l'appartenance de la cellule à une tranche, ligne 7 de la figure 47, nécéssite de regarder le cylindre sous une vue en deux dimensions de manière à avoir un cercle. Par la suite, les coordonnées Y et Z de la cellule servent dans le calcul de son angle dans le cercle par rapport au centre. En fonction de cet angle et du nombre de domaines voulus par l'utilisateur, une tranche constituant son domaine d'appartenance lui est attribuée.

Cette nouvelle implémentation fournit en sortie des décompositions assez régulières comme on peut le voir figure 46. Puisque le maillage s'y prête, on obtient des domaines supposés équilibrés en terme de coût opératoire et de niveau temporel puisque ces partitions découpées selon l'axe vertical sont relativement identiques en terme de distribution de niveaux temporels et de nombre de cellules.

5.4.3 Expériences et résultats

Dans un premier temps, ce nouveau partitionnement est comparé à celui de Scotch sur une petite configuration de 16 processus de 4 cœurs chacun (64 cœurs au total). Le nombre de domaines passé en paramètre est 128. Toutes les exécutions sont réalisées avec la priorité aux faces externes distribuées implémentée dans la section précédente.

La figure 49 montre que l'on obtient un gain de +45.8215% grâce à l'utilisation du partitionneur géométrique. Cette comparaison de trace fournit plusieures informations sur l'exécution et le comportement de cette nouvelle implémentation. Une première observation est que la fin du chemin critique (en bleu) diffère puisque le graphe des deux exécutions n'est plus le même. Le graphe résultant de la décomposition avec Scotch est constitué de 75.547 tâches tandis que celui du nouveau en contient 217.857. Ce changement d'ordre de grandeur est dû au fait que chaque domaine possède une quantité similaire de cellule des différents niveaux temporels présents dans le maillage.

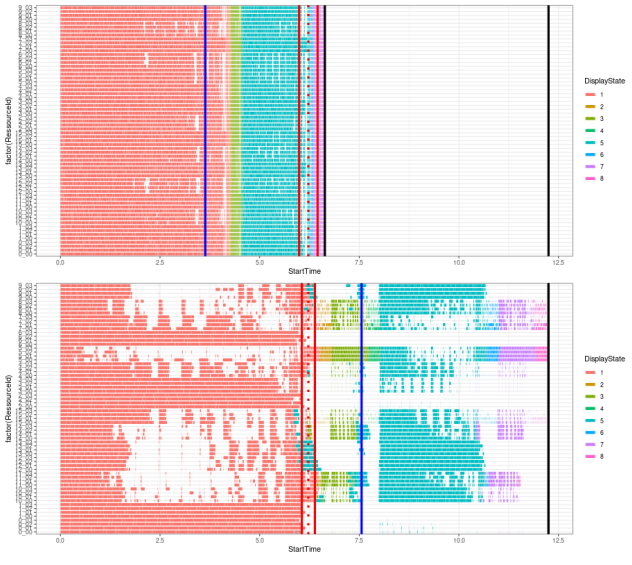


FIGURE 49 – Traces d'exécution avec le maillage partitionné par Scotch (en bas) et par le nouveau partitionneur géométrique (en haut) +45.8215%.

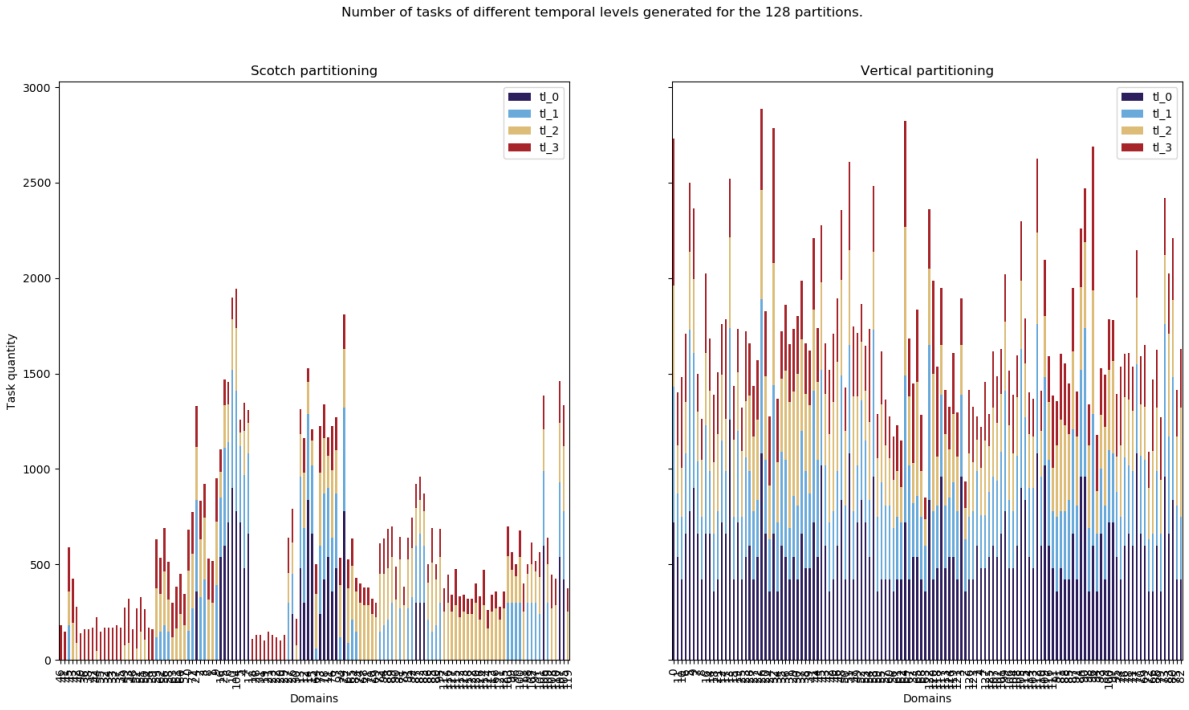


FIGURE 50 – Quantité de tâche des différents niveaux temporels générée par domaine avec un partitionnement réalisé par Scotch (à gauche) et par le multi-critère implémenté (à droite).

La figure 50 indique dans le cas des deux partitionnements comparés, la quantité de tâches de

niveau temporel 0, 1, 2 et 3 que chaque domaine a généré pendant l'exécution. La première remarque est que celle utilisant Scotch produit bien moins de tâches dont la répartition est disparate entre les domaines. Il est possible d'observer que les fonctions *foreach* génèrent des tâches d'un seul niveau temporel pour certains domaines de Scotch tandis que pour d'autres de l'ensemble des niveaux présents dans le maillage. Au contraire, à partir des partitions obtenues avec le nouveau partitionneur implicite, les *foreach* semblent générer un nombre de tâches où la disparité est moins flagrante. Chaque domaine possède une quantité de tâches de chaque niveau temporel assez équivalente.

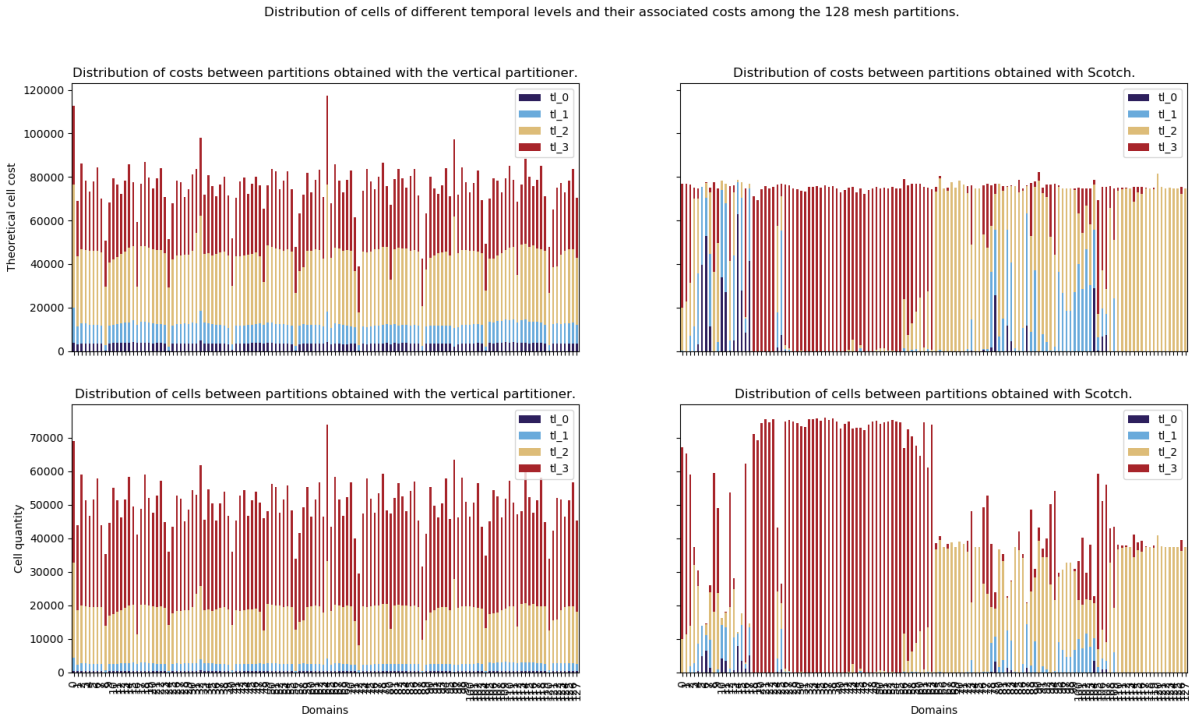


FIGURE 51 – Distribution des cellules de différents niveaux temporels et leurs coûts associés entre les 128 partitions du maillage.

La figure 51 présente la distribution des cellules de différents niveaux temporels du maillage entre les domaines et leur coût opératoire théorique correspondant. Les graphiques de droite extraits des données du partitionnement de Scotch confirment bien le comportement décrit précédemment. Les coûts entre domaines sont presque parfaitement répartis puisque c'est ce critère exact qui est passé en paramètre. En revanche, la quantité de cellule entre domaines est assez déséquilibrée et certains d'entre eux ne contiennent qu'un type de niveau temporel. Les graphiques de gauche correspondant au nouveau partitionneur indiquent que les cellules sont moins bien réparties en terme de coût opératoire. Cependant, la distribution de chaque niveau temporel entre domaines que ce soit en terme de quantité de cellule ou de coût opératoire est bien équilibrée. Le partitionneur crée pourtant des pics réguliers sur les graphiques. Ces derniers proviennent du fait que vers le centre où les tranches deviennent très fines, les cellules peuvent basculer d'une partition à l'autre à cause des arrondis lors des calculs. Ces cellules étant très coûteuses, cela influence fortement l'exécution.

La figure 52 indique l'obtention d'un gain de +32.9064% avec la nouvelle implémentation sur une configuration plus importante de 32 processus ayant 4 cœurs chacun. Le maillage est partitionné en 512 domaines et 1.237.502 tâches sont générées contre seulement 284.612 pour l'exécution utilisant la bibliothèque Scotch. Malgré le temps d'exécution réduit, il est toujours possible d'observer des périodes d'inactivité sur la meilleure trace. Les périodes d'inactivité ne semblent pas provenir d'un mauvais équilibrage de charge entre processus puisque l'écart-type est très réduit.

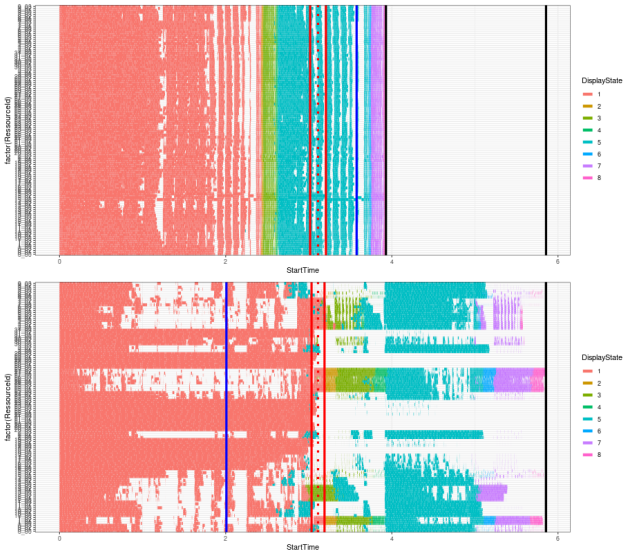


FIGURE 52 – Traces d'exécution avec le maillage partitionné par Scotch (en bas) et par le nouveau partitionneur géométrique (en haut) +32.9064%.

Le nouveau partitionnement multi-critère géométrique fournit donc une exécution plus efficace en distribuant les tâches de manière plus équilibrée au cours de l'exécution. Le parcours du graphe est plus flexible avec des tâches plus fines en quantité plus importante. Cela permet de limiter l'apparition de famine de travail entre les cœurs et de limiter l'influence de l'intégration temporelle dans le graphe de tâche. L'équilibrage des niveaux temporels semble donc plus décisif dans l'exécution finale au profit d'un équilibrage global des coûts opératoires.

6 Granularité

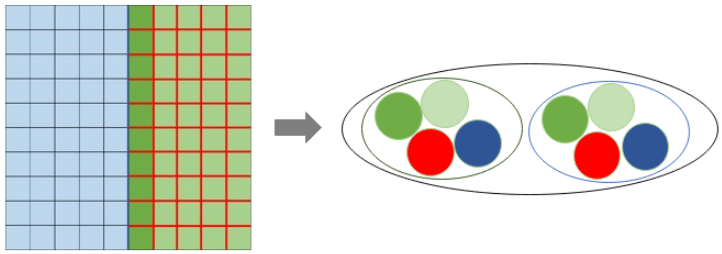
La section précédente a permis de déterminer qu'un partitionnement multi-critère adapté apporte des résultats prometteurs. Cependant, la trace d'exécution présente toujours des périodes d'inactivité, certes réduites, mais assez régulières au sein de chaque sous-itération. Le partitionnement géométrique produit des tâches plus fines en terme d'exécution, et donc en plus grande quantité. En effet, quelle que soit la configuration des tâches, la charge de travail cumulée à la fin de l'itération reste toujours la même. Cette approche fournit plusieurs apports à l'exécution. Le premier est d'obtenir un parcours du graphe ainsi qu'une distribution de tâche plus flexibles. Le second est de pouvoir nourrir plus facilement les ressources au cours de l'exécution. Ces deux éléments permettent de limiter les situations de famine qui peuvent occurer entre cœurs. En effet, une large tâche ne peut nourrir qu'un cœur pendant un certain temps et crée une chaîne plus longue dans le graphe, tandis que ce même travail, divisé en plusieures tâches fines bien réparties, peut en nourrir plusieurs et ainsi réduire le temps d'exécution.

Une déduction de nos expériences précédentes, peu intuitive de prime abord, est qu'en augmentant le nombre de tâches, le temps d'exécution final a été réduit. L'idée est donc de jouer avec cette notion de granularité des tâches, c'est-à-dire leur quantité de travail respective (i.e. leur temps d'exécution). La décomposition de domaine intervient directement sur ce paramètre et donc sur la quantité générée de tâches. La suite des implémentations de cette section se concentre donc toujours sur la première étape du processus de fonctionnement de FLUSIM qui est la décomposition de domaine. Maintenant que les critères d'équilibrage sont fixés, à savoir le coût opératoire des cellules et la répartition des niveaux temporels entre domaines, il faut trouver un autre levier qui modifie ce nouveau paramètre.

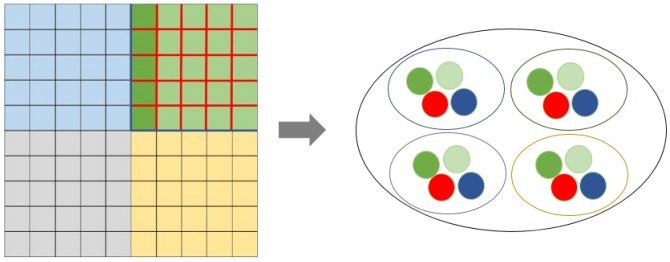
6.1 Nombre de domaines et nombres de tâches

La clé vient de l'implémentation du processus de génération de tâches. Ce dernier crée pour chaque domaine et à l'aide d'une fonction *foreach* une tâche qui va calculer l'ensemble de la composante d'un niveau temporel donné. Par exemple, pour traiter l'ensemble des cellules internes $\tau = 1$, une tâche qui engloble tous les calculs de ces cellules va être générée. Il est possible de déduire que plus il y a de partitions présentes au sein d'un maillage, plus la quantité de tâches est importante. La figure 53 expose cette caractéristique. La figure 53a présente un partitionnement en deux domaines d'un maillage 2D. Les fonctions *foreach* génèrent schématiquement quatre tâches pour chacun de ses deux domaines, une pour chaque composante. La tâche traitant les cellules internes du domaine bleu englobe les 40 qui le composent (si l'on suppose une valeur de τ uniforme). En revanche, un découpage en 4 domaines du même maillage génère 4×4 tâches. Ces dernières sont plus fines en terme de charge de travail. La tâche qui traite les cellules internes du domaine bleu en englobe seulement 20. Le coût de la tâche est donc plus faible. En conséquence, la granularité des tâches diminue avec l'augmentation du nombre de domaine, qui en revanche augmente la quantité de celles-ci.

L'idée abordée dans cette section est d'augmenter le nombre de domaines passé en paramètre de la fonction de partitionnement. La thèse de M. Jean-Marie Couteyen-Carpaye[11], qui est à l'origine de la *taskification* du code de production FLUSEPA, précise que l'introduction de la notion de domaine provient de la granularité trop fine apportée par le calcul d'une seule cellule ou d'une seule face. En effet, les regrouper permet de produire des tâches couvrant le coût de leur création et ordonnancement avec le runtime StarPU. Cela permet aussi de couvrir le coût introduit par les



(a) Partitionnement en 2 domaines



(b) Partitionnement en 4 domaines

FIGURE 53 – Quantité générée de tâches en fonction du nombre de partitions au sein du maillage.

communications entre processus. En effet, la présence de transfert dans l'exécution où la taille des messages envoyés ne permet pas de recouvrir les surcoûts engendrés pénalise fortement l'exécution. En revanche, afin d'isoler les étapes de processus du fonctionnement de simulation (partitionnement, création du graphe et ordonnancement), ces coûts de communication sont ignorés dans le simulateur FLUSIM. Il peut donc être intéressant d'essayer une approche extrême en enlevant cette notion de domaine afin de déterminer si ce paramètre de granularité est à l'origine des périodes d'oisiveté observées sur les cœurs.

6.2 Limitations du simulateur FLUSIM

L'objectif de cette section est donc d'étudier un partitionnement où une cellule correspond à un domaine. Le maillage original contient huit millions de cellule. Dans les expériences précédentes, des maillages possèdant moins de 1000 partitions ont généré des graphes de 500.000 à plus d'un million de tâches. Un partitionnement en huit millions de domaine n'est pas supporté par le simulateur FLUSIM : trop de tâches sont produites. Une solution consiste à appliquer ce partitionnement sur un maillage avec moins de cellules, permettant d'utiliser cette nouvelle implémentation sans générer une quantité démesurée de tâches.

Bien que nous ayons à disposition le petit simulateur de maillage 2D, tester sur un maillage réel 3D est préférable dans la mesure où les résultats sont plus représentatifs. Pour cela, une coupe caractéristique est extraite du maillage original afin d'obtenir un petit maillage 3D. Celui-ci est échantillonné afin de contenir l'ensemble des niveaux temporels et d'imiter la configuration de celui de départ avec un centre de mailles très coûteuses qui le sont de moins en moins en s'éloignant du cœur du phénomène, typique des maillages de mécanique des fluides.

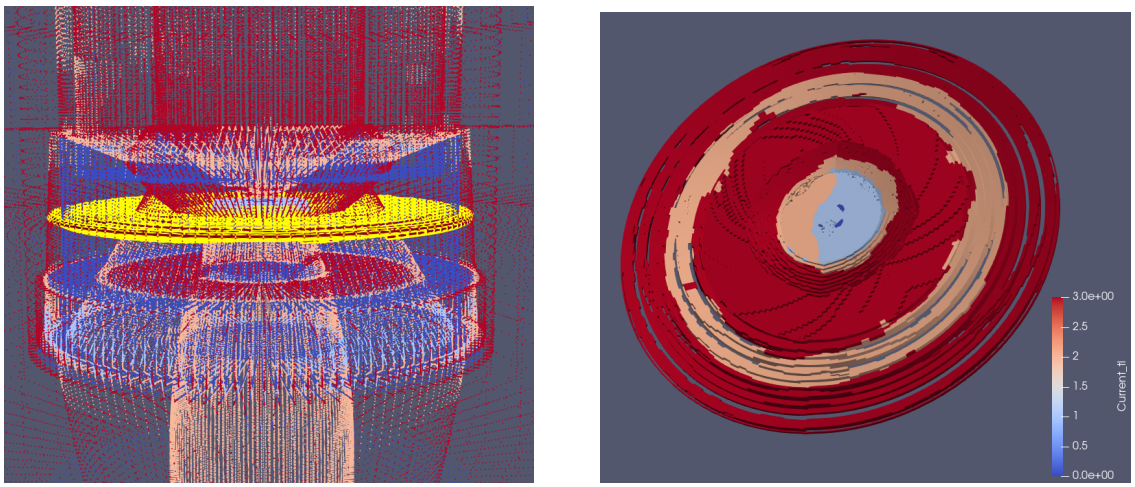


FIGURE 54 – Nouveau maillage extrait de l'original fourni par *Airbus*. Les mailles sont coloriées en fonction de leur niveau temporel.

En se basant sur la composition du maillage d'*Airbus*, la coupe s'effectue au sein de la pièce centrale contenue dans le cylindre. Pièce qui contient un échantillon de chacun des différents niveaux temporels existants dans le maillage. De plus, c'est au sein de cette pièce que se trouve l'ensemble des cellules $\tau = 2$, $\tau = 1$ et $\tau = 0$. La pièce possède donc une densité de maille très importante par rapport au reste du maillage.

6.3 Implémentation d'un partitionnement 1 cellule = 1 domaine

Une première approche pour fabriquer un nouveau maillage à partir de l'original a été d'utiliser l'outil Paraview. Celui-ci prend le fichier *.cgns* qui stocke les informations relatives au maillage et affiche sa représentation 3D. Il permet aussi de manipuler ces données. En particulier, son interface propose de sélectionner des mailles en fonction de certains critères et de les extraire du maillage original. Ce type de manipulation permet d'obtenir des représentations telles que celles de la figure 44. La sélection obtenue peut ensuite être écrite dans un nouveau fichier *.cgns*. Cependant, cette solution n'a pas aboutie. Le format de fichier *.cgns* (CFD General Notation System) est une extension spécialement conçue pour le stockage de donnée à destination de code de simulation numérique appliqué à la mécanique des fluides. Ces fichiers possèdent des champs spécifiques au maillage utilisé. En conséquence, les codes de simulation se basent sur la configuration de ce fichier pour leur implémentation. Par exemple, la structure du *.cgns* fourni par *Airbus* correspond aux structures de données Python implémentées. Á partir de l'extraction des données, l'outil Paraview a produit un fichier *.cgns* générique qui n'a pas pu être lu par FLUSIM, les champs de stockage étant différents.

La solution est donc d'intervenir plus tard dans le processus du simulateur, en particulier dans la fonction de partitionnement de FLUSIM. La structure de donnée *computation_ elements* indique la partition de chaque cellule. Celles dont la partition indique une valeur de -1 ne sont pas prises en compte ni par le futur partitionnement, ni par le processus de génération du graphe de tâches et ni pour l'exécution. Ces cellules n'existent donc pas pour le simulateur FLUSIM. La fonction de partitionnement géométrique implémentée ainsi que celle qui appelle la bibliothèque Scotch ne modifient pas la valeur de ces mailles. Subséquemment, mettre tous les indices des mailles que l'on

ne souhaite pas inclure à -1 dans la structure *computation_ elements* résulte en l'extraction d'un nouveau maillage. Une nouvelle fonction de partitionnement est implémentée que l'utilisateur peut appeler au choix entre la fonction utilisant le partitionneur Scotch et la fonction du partitionneur géométrique multi-critère à l'aide d'un script automatisé.

```
1  int num_domains = 0;
2  for (int i = 0; i < dd->num_cells; ++i)
3  {
4      if (dd->computation_elements[i] != -1 && i <= dd->zones[0].size[1])
5      {
6          float X = dd->zones[0].coord_x[dd->zones[0].connectivity[i * 8]-1];
7          float Y = dd->zones[0].coord_y[dd->zones[0].connectivity[i * 8]-1];
8          float Z = dd->zones[0].coord_z[dd->zones[0].connectivity[i * 8]-1];
9
10         if ((X > -4.22 && X <= -4.2) && (Y >= -2 && Y <= 2) && (Z >= -2.5 && Z <= 2.5))
11         {
12             dd->computation_elements[i] = num_domains;
13             num_domains++;
14         } else
15             dd->computation_elements[i] = -1;
16     } else
17         dd->computation_elements[i] = -1;
18 }
```

FIGURE 55 – Fonction réalisant le partitionnement *1 cellule = 1 domaine.*

Similairement à l'implémentation du partitionneur géométrique, cette fonction présente une boucle principale qui itère sur l'ensemble des huit millions de cellules. Par la suite, une condition est posée ligne 4 de la figure 55, afin d'éviter de traiter les cellules ayant déjà pour valeur de domaine -1 et de tester si la cellule se situe bien dans la première *zone* du maillage. En effet, avant d'obtenir la coupe optimale, de nombreux essais ont été réalisés. L'outil de visualisation Paraview a permis de séparer les cellules des deux zones du maillage afin d'avoir un aperçu de leur configuration. Après avoir défini un intervalle de coordonnées approximatives contenant un échantillon de l'ensemble des niveaux temporels existants, ces coordonnées sont raffinées jusqu'à obtenir une coupe représentative satisfaisante. La difficulté a été d'obtenir une coupe qui soit un peu isolée dans le sens où la topologie est au mieux respectée. Dans notre cas, la coupe obtenue forme un cercle tandis que le découpage est quadrangulaire. Le découpage ne tranche pas arbitrairement dans le maillage et conserve la forme initiale de cette région du maillage dans l'axe vertical. Ensuite, l'intervalle de la coordonnée X a été défini de manière à ce que l'extrait contienne le moins de cellule possible dans cette partie du maillage extrèmement dense. La coupe finale contient tout de même 32.330 cellules.

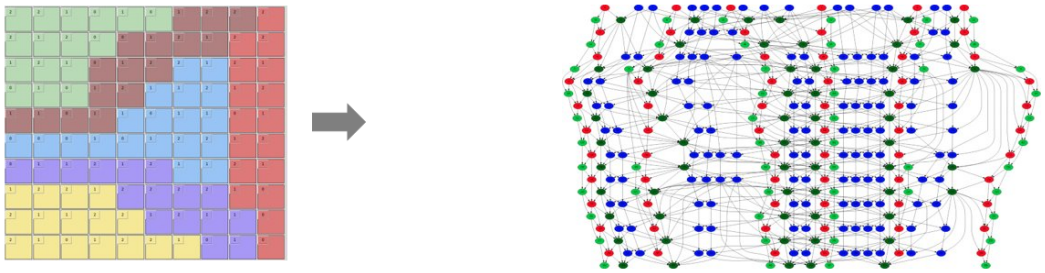
L'ensemble des mailles de la coupe se situe dans la première zone du maillage, contenant notamment la pièce centrale interne. Toutes les autres mailles sont mises à -1 directement ligne 17. Au contraire, si la maille est bien dans la première zone, les coordonnées X, Y, Z sont récupérées de manière similaire au partitionneur géométrique avec les champs *coord_ x/y/z* et *connectivity*. La nuance ici est que la coordonnée X est aussi récupérée puisque le cylindre est découpé horizontale-

ment. Si la maille est comprise dans les intervalles définis pour chaque coordonnée, alors un numéro de domaine lui est attribué ligne 12, grâce à un compteur *num_ domains* initialisé au début de la fonction. L'indice de la boucle principale qui itère sur les cellules pourrait théoriquement être utilisé dans ce but. En revanche, cette numérotation peut rendre difficile la détection d'erreur. Notamment au niveau de l'échelle dans Paraview qui indique la numérotation des domaines du maillage affiché et par déduction le nombre total de domaine présent. Ne pas inclure de compteur va faire en sorte que la borne supérieure présentée par l'échelle ne corresponde plus à la quantité de partition obtenue mais à l'indice le plus élevé de la maille qui a été incluse dans le nouveau maillage. Par exemple, la borne maximale peut avoir une valeur de 500.000 alors que le partitionnement s'est fait en 32.330 domaines, 500.000 étant l'indice de la maille dans *computation_ elements*. Enfin, la condition ligne 15 met à -1 les cellules incluses dans la première zone mais pas dans les intervalles de coordonnée définis.

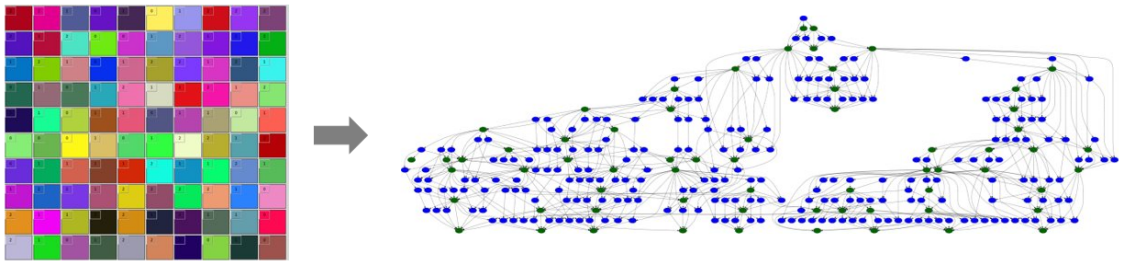
6.4 Partitionnement extrême et génération du graphe

Á partir du maillage 2D défini dans l'outil de visualisation, la figure 56 présente les différents graphes de tâche obtenus en faisant varier le partitionnement utilisé. La figure 56a expose celui provenant d'un partitionnement en 6 domaines, tandis que la figure 56b celui en appliquant le nouveau partitionneur *1 cellule = 1 domaine*.

Une première observation est que ce dernier ne contient que deux types de tâches puisque le maillage ne contient que des cellules et faces de bord. De plus, le graphe est plus développé en largeur, traduisant des tâches moins imbriquées dans les dépendances et éventuellement plus de tâches disponibles à chaque instant t de l'exécution.



(a) Partitionnement en 6 domaines arbitraires



(b) Partitionnement 1 cellule = 1 domaine

FIGURE 56 – Comparaison des graphes obtenus en fonction du partitionneur utilisé sur le maillage 2D de l'outil de visualisation implémenté.

Appliqué sur le nouveau domaine 3D, le partitionneur *1 cellule = 1 domaine* permet un gain de +81.8652% par rapport à une exécution appelant le partitionneur multi-critère implicite. Et ce avec une configuration identique, à savoir 512 cœurs répartis entre 16 processus. Comme attendu, la meilleure exécution contient 467.500 tâches, soit presque dix milles fois plus que celle comparée. Une observation intéréssante est que la compression des tâches de l'exécution avec 32.330 partitions confirme l'allure particulière du graphe, pressentie en appliquant ce même partitionnement sur le maillage minime 2D de l'outil de visualisation figure 56b.

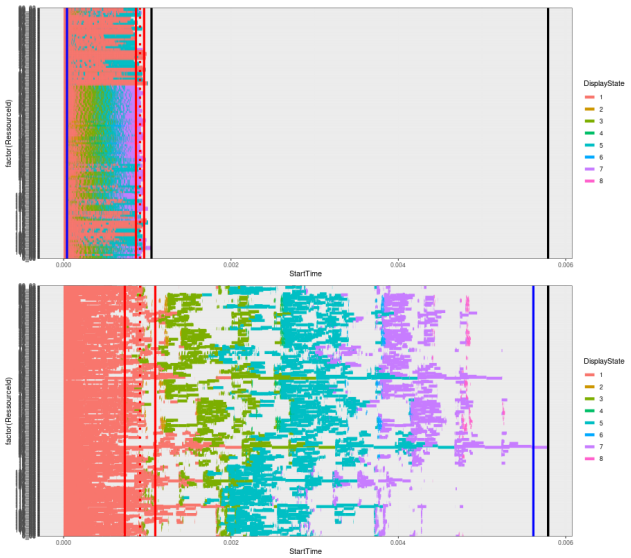


FIGURE 57 – Comparaison des traces d'exécution obtenues avec un partitionnement géométrique en 512 domaines (en bas) et avec un partitionnement *1 cellule = 1 domaine* en 32.330 domaines (en haut)+81.8652%.

Cette nouvelle approche offre des résultats probants. Bien qu'un tel niveau de granularité ne puisse pas être appliqué sur le maillage de huit millions de cellules, il est possible d'augmenter drastiquement le nombre de partitions jusqu'à atteindre la limite de tâche supportée par le simulateur. La figure 58 présente une exécution où le maillage a été divisé en 32.330 domaines, le code générant ainsi 2.503.290 tâches. La fonction de partitionnement utilisant Scotch est utilisée afin de conserver au mieux la topologie dans ce cas-ci. La trace résultante, visible sur la figure 58, affiche une exécution extrèmement compressée et ne présentant pas de période d'inactivité flagrante.

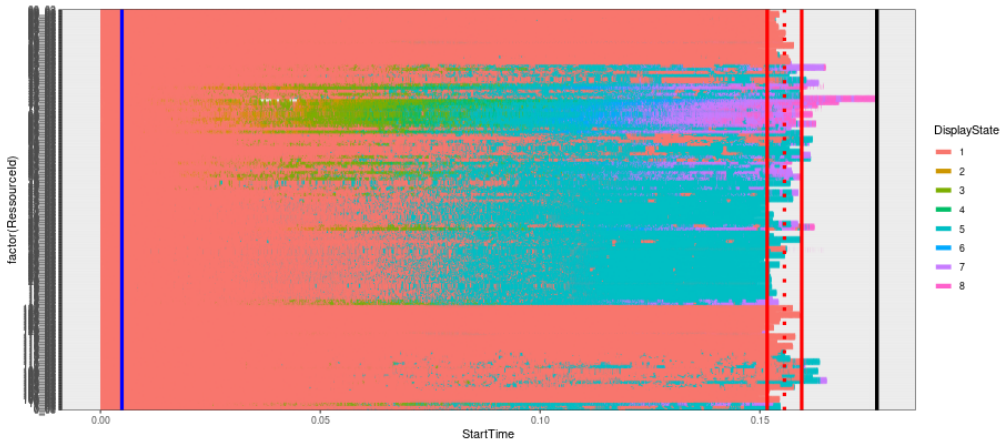


FIGURE 58 – Trace d'exécution obtenue en partitionnant le maillage original en 32.330 domaines avec plusieures des optimisations implémentées.

Cette dernière trace, en plus d'appliquer une granularité favorable à l'exécution, concentre plusieures optimisations, à savoir un partitionnement équilibrant les coûts opératoires entre domaines et une distribution (mapping) équilibrant ces coûts entre processus. Et ce, en plus de l'utilisation de la stratégie d'ordonnancement aux faces externes distribuées. Comparée à une exécution de 1024 partitions lancée dans les premiers jours du stage, le gain s'élève à 61.6212% et ce sur une configuration identique de 128 processus ayant 4 cœurs chacun.

Paramétrer correctement la granularité, en fonction d'un maillage et d'une configuration donnée, livre des résultats prometteurs. En effet, développés dans le sens de la largeur, ces graphes permettent d'éviter les périodes d'inactivité en proposant une quantité de tâche prête importante à chaque instant de l'exécution. Les effets des dépendances et des synchronisations sont fortement réduits dès que les tâches sont présentes en quantité suffisante dans la structure *ready_ tasks* afin de nourrir tous les cœurs.

Quatrième partie

Conclusion

Les travaux menés durant ce stage ont permis d'obtenir des calculs de simulation qui s'exécutent de manière efficace sur une machine parallèle et ce, en déterminant les principales sources des périodes d'inactivité présentes sur les cœurs. Une exécution idéale, dans un environnement qui l'est aussi, repose sur un équilibrage des coûts opératoires et des niveaux temporels entre partitions du maillage, sur une stratégie d'ordonnancement permettant un parcours du graphe de tâches optimal et enfin sur une granularité adaptée au problème.

L'émulateur FLUSIM, imitant le comportement du code de production FLUSEPA, permet d'isoler et d'intervenir sur les trois étapes majeures du fonctionnement de simulation numérique. Les outils de visualisation étendus et implémentés durant ce stage ont permis de saisir l'influence de chacun des paramètres et variables sur la génération du graphe de tâches, notamment celles des discrétisations spatiales et temporelles. Un graphe très imbriqué dans les dépendances et la présence de point de contention au sein de celui-ci ont résulté en l'implémentation de plusieures politiques d'ordonnancement. *As soon as possible* et *As late as possible*, inspirées d'un article scientifique [6], attribuent une priorité en fonction de la position d'une tâche dans une exécution idéale. En revanche, la stratégie que nous avons implantée permet de cibler et prioriser des tâches spécifiques menant à des points de contention du graphe. Grâce aux outils de visualisation de graphe, ces tâches sont identifiées comme celles traitant les faces externes dont les domaines de bordure sont distribués sur des processus différents. La politique priorise en conséquence les autres tâches menant jusqu'à elles et situées sur les chaînes les plus longues du graphe. Guider de cette manière l'ordre d'exécution permet de débloquer plus rapidement de nouvelles tâches *prêtes* sur des nœuds distants.

Bien que par la suite l'ensemble des expériences recourt à cette priorité, amenant un gain de 6% sur la plupart des exécutions, de régulières périodes de pénurie de tâches demeurent au sein des traces. Celles-ci sont cependant réduites de plus de 45% grâce à l'implémentation d'un partitionneur géométrique, équilibrant les coûts opératoires des cellules et la répartition des niveaux temporels entre domaines. Ce nouveau partitionneur multi-critère implicite produit un découpage de qualité et donc une exécution plus efficace puisque l'effet se répercute dans le processus de simulation à travers le graphe de tâche généré.

Afin de traquer les dernières périodes oisives restantes, et puisque le simulateur le permet en ignorant les coûts associés aux communications entre processus, la granularité est paramétrée à l'extrême. En ce sens, une nouvelle implémentation permet d'extraire un maillage 3D réduit et d'y appliquer un partitionnement où une cellule constitue maintenant un domaine. Avec une amélioration de 80% du temps d'exécution, le paramètre de granularité est adapté en conséquence dans le cadre de l'utilisation du maillage original, en augmentant fortement le nombre de partitions voulues. Un graphe évitant l'occurence de situation de famine, permet de réduire le temps d'exécution de plus de 60% sur certaines configurations.

Ce stage étant un préambule à une thèse, le passage au code de simulation numérique FLUSEPA utilisé en production nécéssite quelques ajustements. En effet, l'émulateur fait abstraction des coûts engrendrés par les communications entre processus MPI mais aussi ceux relatifs à la création et l'ordonnancement de tâche. La soumission de tâche à la bibliothèque StarPU représente par exemple 20% du temps d'exécution. En conséquence, une telle granularité n'est pas applicable dans le code FLUSEPA. L'optique est d'obtenir d'aussi bons résultats en diminuant la quantité de partition.

Une solution à ce problème de granularité est l'utilisation d'un partitionneur multi-critère adapté. Celui implémenté dans ce stage divise le maillage de manière arbitraire en jouant sur la symétrie de géométrie et de niveau temporel de celui-ci. D'une part, les maillages ne présenteront pas toujours ces caractèristiques avantageuses. D'autre part, la coupe n'est pas faite de manière à limiter les arêtes entre partitions et donc les communications et synchronisations en résultant, au contraire de Scotch. En revanche, cet apport identifie clairement les critères de partitionnement décisifs. Une perspective envisagée est d'obtenir une bibliothèque Scotch qui implémente un algorithme de partitionnement initial multi-critère tout en limitant la création d'arêtes entre partitions. Il peut aussi être intéressant de tester d'autres logiciels de partitionnement multi-niveau, comme l'outil Jostle [27] par exemple.

Parmi les autres perspectives envisagées, certaines concernent directement l'émulateur FLUSIM. D'une part, des maillages asymétriques et présentant plusieurs foyers de cellules coûteuses seront fournis en entrée du simulateur afin d'observer le comportement des nouvelles implémentations. D'autre part, il peut être intéressant de modifier le processus d'interprétation du maillage afin de générer des composantes différentes et permettre éventuellement une meilleure exécution.

Dans une perspective à plus long terme, une approche à base de tâches parallèles (plusieurs cœurs peuvent calculer une même tâche) ou hiérarchiques (les tâches peuvent être divisées en un ensemble de sous-tâches plus fines) offrirait une souplesse intéressante en terme de granularité adaptative.

Références

[1] JAMES AHRENS, BERK GEVECI, and CHARLES LAW. 36 - paraview : An end-user tool for large-data visualization. In Charles D. Hansen and Chris R. Johnson, editors, *Visualization Handbook*, pages 717–731. Butterworth-Heinemann, Burlington, 2005.

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

[3] Remi Barat. *Load Balancing of Multi-physics Simulation by Multi-criteria Graph Partitioning*. Theses, Université de Bordeaux, December 2017.

[4] Rémi Barat, Cédric Chevalier, and François Pellegrini. Partitionnement multi-critères de graphes pour l'équilibrage de charge de simulations multi-physiques. In *Conférence d'informatique en Parallélisme, Architecture et Système (COMPAS)*, Lorient, France, July 2016.

[5] Remi Barat, Cédric Chevalier, and François Pellegrini. Multi-criteria Graph Partitioning with Scotch. In Fredrik Manne, Peter Sanders, and Sivan Toledo, editors, *SIAM Workshop on Combinatorial Scientific Computing*, Proceedings of the Seventh SIAM Workshop on CSC, pages 66–75, Bergen, Norway, June 2018. Society for Industrial and Applied Mathematics and University of Bergen, Society for Industrial and Applied Mathematics.

[6] Olivier Beaumont, Julien Langou, Willy Quach, and Alena Shilova. A Makespan Lower Bound for the Scheduling of the Tiled Cholesky Factorization based on ALAP Schedule. In *Euro-Par 2020 - 26th International European Conference on Parallel and Distributed Computing*, Proceedings of EuroPar 2020, Warsaw / Virtual, Poland, August 2020. Springer.

[7] Astrid Casadei, Pierre Ramet, and Jean Roman. An improved recursive graph bipartitioning algorithm for well balanced domain decomposition. In *IEEE International Conference on High Performance Computing (HiPC 2014)*, pages 1–10, Goa, India, December 2014.

[8] Cédric Chevalier and François Pellegrini. PT-Scotch : Un outil pour la renumérotation parallèle efficace de grands graphes dans un contexte multi-niveaux. In *RenPar'17 / SympA'2006 / CFSE'5 / JC'2006*, page 8 pages, Canet en Roussillon, France, October 2006. 8 pages.

[9] Cédric Chevalier and François Pellegrini. PT-Scotch : A tool for efficient parallel graph ordering. *Parallel Computing*, 34(6-8) :318–331, July 2008.

[10] Terry Cojean, Abdou Guermouche, Andra Hugo, Raymond Namyst, and Pierre-André Wacrenier. Resource aggregation for task-based Cholesky Factorization on top of heterogeneous machines. In *HeteroPar'2016 worshop of Euro-Par*, Grenoble, France, August 2016.

[11] Jean Marie Couteyen Carpaye. *Contribution à la parallélisation et au passage à l'échelle du code FLUSEPA*. Theses, Université de Bordeaux, September 2016.

[12] Jean Marie Couteyen Carpaye, Jean Roman, and Pierre Brenner. Towards an efficient Task-based Parallelization over a Runtime System of an Explicit Finite-Volume CFD Code with Adaptive Time Stepping. In *International Parallel and Distributed Processing Symposium*, PDSEC'2016 workshop of IPDPS, page 10, Chicago, IL, United States, May 2016.

[13] Jean Marie Couteyen Carpaye, Jean Roman, and 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. *International Journal of Computational Science and Engineering*, pages 1 – 22, 2017.

[14] Devine, Karen and Boman, Erik and Riesen, Lee and Catalyurek, Umit and Chevalier, Cédric. *Zoltan : Parallel toolkit for combinatorial scientific computing : dynamic partitioning, graph coloring and ordering*, 2022.

[15] Alice Gatti, Zhixiong Hu, Pieter Ghysels, Esmond G. Ng, and Tess E. Smidt. Graph partitioning and sparse matrix ordering using reinforcement learning. *CoRR*, abs/2104.03546, 2021.

[16] Karypis, George and Kumar, Vipin. *hMETIS : A Hypergraph Partitioning Package*, 1998.

[17] Karypis, George and Kumar, Vipin. *ParMETIS : Parallel graph partitioning and sparse matrix ordering library*, 2020.

[18] Karypis, George and Kumar, Vipin. *METIS : A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices*, 2022.

[19] Message Passing Interface Forum. *MPI : A Message-Passing Interface Standard Version 4.0*, June 2021.

[20] François Pellegrini. *Contributions au partitionnement de graphes parallèle multi-niveaux.* Habilitation à diriger des recherches, Université Sciences et Technologies - Bordeaux I, December 2009.

[21] François Pellegrini. Scotch and libScotch 7.0 User's Guide. Technical report, Université Sciences et Technologies - LaBRI, UMR CNRS 5800 - TadAAM team, INRIA Bordeaux Sud-Ouest, February 2022.

[22] François Pellegrini and Cédric Lachat. Process Mapping onto Complex Architectures and Partitions Thereof. Research Report RR-9135, Inria Bordeaux Sud-Ouest, December 2017.

[23] Pellegrini, François. *SCOTCH : Static mapping, graph partitioning, and sparse matrix block ordering package*, December 2021.

[24] Raphael Prat. *Équilibrage dynamique de charge sur supercalculateur exaflopique appliqué à la dynamique moléculaire.* Theses, Université de Bordeaux, October 2019.

[25] Peter Sanders and Christian Schulz. Think locally, act globally : Highly balanced graph partitioning. In *Experimental Algorithms, 12th International Symposium, SEA 2013, Rome, Italy, June 5-7, 2013. Proceedings*, volume 7933, pages 164–175. Springer, 2013.

[26] Ümit V. Çatalyürek and Cevdet Aykanat. *PaToH : Partitioning Tool for Hypergraphs*, 2020.

[27] Chris Walshaw and Mark Cross. Jostle : Parallel multilevel graph-partitioning software - an overview. *Mesh Partitioning Techniques and Domain Decomposition Techniques*, 01 2007.