Contribution à la parallélisation et au passage à l'échelle du code FLUSEPA

Jean Marie Couteyen Carpaye

▶ To cite this version:

Jean Marie Couteyen Carpaye. Contribution à la parallélisation et au passage à l'échelle du code FLUSEPA. Autre [cs.OH]. Université de Bordeaux, 2016. Français. NNT : 2016BORD0073 . tel-01399952

HAL Id: tel-01399952
https://theses.hal.science/tel-01399952

Submitted on 21 Nov 2016

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

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

université de BORDEAUX

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

Par **Jean Marie Couteyen Carpaye**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ INFORMATIQUE

Contributions à la parallélisation et au passage à l'échelle du code FLUSEPA

Soutenue le : 19 Septembre 2016

Devant la commission d'examen composée de :

Raymond Namyst	Professeur, Université de Bordeaux	Président du jury
Frédéric Desprez	Directeur de Recherche, Inria	Rapporteur
Xavier Gloerfelt	Professeur, Arts et Métiers ParisTech	Rapporteur
Vincent Moureau	Chargé de Recherche CNRS, CORIA	Examinateur
Jean Roman	Professeur, Inria et Bordeaux INP	Directeur de thèse
Pierre Brenner	Ingénieur-Expert CFD, Airbus Defence and Space	Co-encadrant

– 2016 –

Titre : Contributions à la parallélisation et au passage à l'échelle du code FLUSEPA.

Résumé :

Les satellites sont mis en orbite en utilisant des lanceurs dont la conception est une des activités principales d'Airbus Defence and Space. Pour ce faire, se baser sur des expériences n'est pas facile : les souffleries ne permettent pas d'évaluer toutes les situations auxquelles un lanceur est confronté au cours de sa mission. La simulation numérique est donc essentielle pour l'industrie spatiale. Afin de disposer de simulations toujours plus fidèles, il est nécessaire d'utiliser des supercalculateurs de plus en plus puissants. Cependant, ces machines voient leur complexité augmenter et pour pouvoir exploiter leur plein potentiel, il est nécessaire d'adapter les codes existants. Désormais, il semble essentiel de passer par des couches d'abstraction afin d'assurer une bonne portabilité des performances. ADS a développé depuis plus de 20 ans le code FLUSEPA qui est utilisé pour le calcul de phénomènes instationnaires comme les calculs d'onde de souffle au décollage ou les séparations d'étages. Le solveur aérodynamique est basé sur une formulation volume fini et une technique d'intégration temporelle adaptative. Les corps en mouvement sont pris en compte via l'utilisation de plusieurs maillages qui sont combinés par intersections.

Cette thèse porte sur la parallélisation du code FLUSEPA. Au début de la thèse, la seule version parallèle disponible était en mémoire partagée. Une première version parallèle en mémoire distribuée a d'abord été réalisée. Les gains en performance de cette version ont été évalués via l'utilisation de deux cas tests industriels. Un démonstrateur du solveur aérodynamique utilisant la programmation par tâche au dessus d'un runtime a aussi été réalisé.

Mots clés : Calcul haute performance, aérodynamique, corps en mouvement relatif.

Title : Contributions to the parallelization and the scalability of the FLUSEPA code.

Abstract :

There are different kinds of satellites that offer different services like communication, navigation or observation. They are put into orbit through the use of launchers whose design is one of the main activities of Airbus Defence and Space. Relying on experiments is not easy : wind tunnel cannot be used to evaluate every critical situation that a launcher will face during its mission. Numerical simulation is therefore mandatory for spatial industry.

In order to have more reliable simulations, more computational power is needed and supercomputers are used. Those supercomputers become more and more complex and this implies to adapt existing codes to make them run efficiently. Nowadays, it seems important to rely on abstractions in order to ensure a good portability of performance. Airbus Defence and Space developed for more than 20 years the FLUSEPA code which is used to compute unsteady phenomena like take-off blast wave or stage separation. The aerodynamic solver relies on a finite volume formulation and an explicit temporal adaptive solver. Bodies in relative motion are taken into account through the use of multiple meshes that are overlapped.

This thesis is about the parallelization of the FLUSEPA code. At the start of the thesis, the only parallel version available was in shared memory through OpenMP. A first distributed memory version was realized and relies on MPI and OpenMP. The performance improvement of this version was evaluated on two industrial test cases. A task-based demonstrator of the aerodynamic solver was also realized over a runtime system.

Keywords : High Performance Computing, Computational Fluid Dynamics, Bodies in relative motion.

Equipe-projet commune HiePACS (https://team.inria.fr/hiepacs/), Inria, LaBRI UMR CNRS 5800, Université de Bordeaux, Bordeaux INP
Airbus Defence and Space

Résumé étendu

Il existe de nombreux types de satellites qui fournissent des services utiles au quotidien : l'imagerie satellite, les télécommunications, la géolocalisation... Leur mise en orbite passe par l'utilisation de lanceurs dont la conception est une des activités d'Airbus Defence and Space. Pour la conception de lanceurs, l'accès à l'expérience n'est pas évident et l'utilisation de souffleries ne permet pas de tester toutes les situations critiques auxquelles un lanceur sera confronté au cours de sa mission. La simulation numérique est donc essentielle pour l'industrie aérospatiale. Pour disposer de simulations plus fidèles, il est nécessaire de disposer et de pouvoir exploiter une importante puissance de calcul via l'utilisation de *supercalculateurs*. Ces supercalculateurs évoluent rapidement et sont de plus en plus complexes ; il est alors nécessaire d'adapter les codes existants pour pouvoir les utiliser efficacement. Aujourd'hui, il semble de plus en plus nécessaire d'utiliser un modèle plus abstrait pour exprimer le parallélisme afin de pouvoir porter les codes sur les nouvelles machines avec un coût humain raisonnable et une bonne portabilité des performances.

Airbus DS a développé depuis plus de 20 ans le code de calcul FLUSEPA[1] qui convient particulièrement bien à la simulation des phénomènes instationnaires avec topologie variable apparaissant par exemple lors des séparations d'étages et des décollages de lanceurs spatiaux. Ce code est basé sur une formulation Volumes Finis. La prise en compte des mouvements relatifs repose sur une technique originale de chevauchement de maillages qui est conservative et une technique d'intégration temporelle adaptative explicite permet de calculer très efficacement les évolutions rapides de la physique du problème.

Les travaux réalisés durant cette thèse portent sur la parallélisation du code FLUSEPA qui au départ n'était parallélisé qu'en mémoire partagée via OpenMP. Une première version distribuée du code a été réalisée et utilise une programmation hybride MPI+OpenMP pour des clusters de calcul. Les gains apportés par cette version ont été évalués via l'utilisation de deux cas de calcul industriels. Un démonstrateur basé cette fois-ci sur un modèle de programmation à base de graphe de tâches avec l'utilisation d'un support d'exécution a aussi été réalisé pour répondre de manière plus adéquate aux problèmes d'efficacité posés par la version MPI+OpenMP.

La première partie du manuscrit présente la parallélisation hybride MPI+OpenMP du code. L'intégralité du code a été parallélisée et il s'agit désormais de la version de référence de production. Le code disposant de deux opérations principales distinctes (les calculs aérodynamiques et les calculs d'intersections de maillages), il a été décidé de retenir une parallélisation MPMD. Ce choix permet l'exploitation de l'indépendance des calculs entre calculs d'intersections et calculs aérodynamiques. Un processus maître est utilisé pour faire le contrôle. La méthode d'intégration temporelle adaptative est présentée. La parallélisation du solveur aérodynamique est décrite ainsi que les performances obtenues pour deux cas industriels. Bien qu'offrant des performances intéressantes (le gain par rapport à la version initiale est quand même de l'ordre de 15), l'efficacité du solveur est pénalisée par la rigidité imposée par la parallélisation mise en œuvre.

La seconde partie de la thèse présente les travaux portant sur la parallélisation du solveur aérodynamique en utilisant le runtime StarPU et dont l'objectif est d'offrir une solution aux limites rencontrées avec la version précédente. La parallélisation en tâches est décrite, depuis sa conception jusqu'à l'adaptation du code déjà existant. Cette implémentation exploite trois niveaux de parallélisme. Un premier niveau est induit par une décomposition de domaine qui permet d'utiliser plusieurs nœuds d'un cluster ; au sein d'un nœud, le second niveau vient de la description du problème via un graphe de tâches et le troisième niveau provient du fait que les

1. Marque déposée en France sous le numéro n^o134009261

tâches peuvent elles-mêmes être parallèles (via OpenMP). Après une validation effective sur un cas industriel de grande taille, on en déduit que l'approche est prometteuse et permettra à terme de réaliser des simulations encore plus complexes et encore plus fidèles.

Les travaux de cette thèse ont été réalisés en collaboration entre Inria (au sein de l'équipe HiePACS) et Airbus Defence and Space dans le cadre d'une convention CIFRE (2012/1152).

Remerciements

Il est donc temps de sortir de l'arène et de contempler le travail accompli. Au cours de cette thèse, j'ai eu l'occasion de fréquenter à la fois le milieu industriel et le milieu académique, entre Bordeaux et Les Mureaux. J'ai eu la chance de pouvoir travailler sur et avec des outils (FLUSEPA & StarPU principalement) très avancés et très prometteurs, fruit d'un long travail commencé par d'autres.

J'ai particulièrement apprécié le fait de me retrouver en dehors de ma zone de confort et de découvrir la CFD. Je me suis retrouvé au sein d'une équipe de gens passionnés et très compétents qui ont su me faire partager leur goût pour la mécanique des fluides. L'originalité du travail de Pierre et la ténacité de Jean-François sont pour beaucoup dans l'existence de cette thèse et la poursuite des travaux sur FLUSEPA. L'autre point sur lequel je voudrais insister est le fait d'avoir pu vraiment profiter de ma formation à Bordeaux au cours de cette thèse : j'ai eu l'occasion de découvrir les runtimes au cours de mon Master au travers de deux projets de programmation et d'un court stage. A mon arrivée à Bordeaux, je n'avais pas forcément en tête de poursuivre dans le calcul haute performance. C'est la qualité de l'enseignement et des enseignants qui m'a fait apprécié cette discipline plus que les autres.

J'aimerai revenir sur quelques moments clés qui ont marqué cette thèse et qui ont compté pour moi. Il va de soi que je remercie implicitement les personnes que je citerai à partir de maintenant. Tout d'abord, il y a eu ce calcul d'onde de souffle que Jean-Marc a passé avec la version MPI ; c'était la première utilisation concrète de mon travail dans un cadre industriel. Ce cas est par la suite devenu mon cas test type pour les développements qui ont concerné le solveur aérodynamique. Ensuite, il y a eu ce comité de pilotage Inria/EADS où j'ai pu présenter mon travail pour la première fois dans un contexte avec un peu d'enjeu. Ce jour-là, j'ai été chanceux qu'Isabelle soit présente. La décision de paralléliser en avance de phase le travail de Grégoire fut aussi un moment clé. Sa motivation lui a ainsi permis de devenir la première personne à coder dans la version MPI (à part moi, évidemment) ! Je crois qu'il ne le regrette pas.

Je souhaite remercier les personnes travaillant (et ayant travaillé) sur StarPU. Particulièrement, je tiens à remercier Andra et Terry pour les tâches parallèles et les contextes. J'ai apprécié les interactions pas forcément très nombreuses mais productives ! L'apport de Nathalie et Samuel a été aussi notable grâce à la correction de bugs qui m'ont concernés.

La qualité du manuscrit doit beaucoup à la rigueur de Jean et à son légendaire stylo rouge. Enfin... il a consommé tellement d'encre que j'ai fini par avoir des corrections en bleu. Pour la qualité des diagrammes de Gantt présents dans le manuscrit, Luka est le principal responsable, vu qu'il m'a partagé ses scripts et son savoir en R. Merci à Xavier Gloerfelt et Frédéric Desprez d'avoir accepté de rapporter mon manuscrit et d'avoir fait l'effort de s'intéresser au HPC pour l'un et à la CFD pour l'autre.

La soutenance fut un moment très spécial et très plaisant. C'est l'occasion pour moi de remercier les membres du jury pour leur intérêt concernant mon travail. Outre les rapporteurs et les encadrants, merci à Raymond d'avoir présidé ce jury et à Vincent Moureau d'avoir été présent. J'ai pu compter sur le soutien remarquable de ma famille. Notamment celui de mes parents et mes sœurs qui ont fait le déplacement. J'ai aussi une pensée particulière pour Marthe, qui bien qu'absente physiquement était bien représentée par ses amies Frédérique et Julie qui ont fourni une grande aide pour le pot. Merci à tout ceux qui ont pu être présents pour la soutenance et pour le cadeau que vous avez choisi ! Merci à Pierre pour l'organisation du restaurant et à Grégoire pour les cigares.

Pour finir, je voudrais remercier les membres d'HiePACS, pour la bonne ambiance de travail, le tarot et les divers apéros...

Table des matières

Table des figures ix

1 Introduction et contexte de la thèse **1**

1.1 Le besoin en simulation de l'industrie aérospatiale pour la conception des lanceurs 1

1.2 Contexte du HPC moderne 3

1.3 Positionnement, objectifs et plan de la thèse 4

2 Conception et mise en œuvre d'une première version MPMD basée sur un parallélisme hybride MPI+OpenMP **5**

2.1 Cadre numérique et description algorithmique du problème 5

2.1.1 Schéma global et différentes étapes du calcul 6

2.1.2 Solveur aérodynamique 6

2.1.3 Corps en mouvement relatif 10

2.2 Description des différents niveaux de parallélisme dans FLUSEPA 12

2.2.1 Structures de données existantes et évolutions apportées 13

2.2.2 Mise en œuvre parallèle MPMD avec un parallélisme hybride MPI+OpenMP 14

2.2.3 Solveur aérodynamique parallèle 15

2.2.4 Prise en compte des déplacements de maillage 18

2.3 Validation expérimentale et analyse de performance 22

2.3.1 Calcul d'une onde de souffle 23

2.3.2 Calcul de la séparation des EAP 41

3 Conception et mise en œuvre d'une version parallèle du solveur aérodynamique avec une expression en graphe de tâches et s'exécutant sur un runtime **47**

3.1 Motivations 47

3.2 Modèle d'expression du parallélisme en graphe de tâches et support d'exécution . 48

3.2.1 Expression du parallélisme et modèles de programmation utilisant un graphe de tâches 49

3.2.2 Runtime pour les architectures modernes 51

3.2.3 Description et fonctionnalités du runtime StarPU 52
3.3 Version parallèle de FLUSEPA en mémoire distribuée utilisant StarPU 55
3.3.1 Réalisation d'une version symbolique pour le solveur aérodynamique en mémoire partagée 56
3.3.2 Préliminaires pour la mise en place d'une version en tâches en mémoire partagée 64
3.3.3 Une version en tâches en mémoire partagée de FLUSEPA 66
3.3.4 Optimisation et validation expérimentale de la version en tâches en mémoire partagée 71
3.3.5 Mise en place d'une version distribuée 80

4 Conclusions et perspectives **89**
4.1 Bilan de la thèse 89
4.2 Perspectives 90

Bibliographie **93**

Liste des publications **99**

Annexes **101**

A Version MPI+OpenMP
Traces pour 16 nœuds **103**

B Performance en mémoire partagée de la version en tâches **105**

C Performance en mémoire partagée pour la version en tâches pour 32 ECs (configuration de calcul (8×2), θ=4) **119**

Table des figures

2.1 Temps atteint après chaque sous-itération pour chaque niveau temporel τ donné en abscisse ($\theta = 3$, 8 sous-itérations). 8
2.2 Intersection de maillages et construction du maillage de calcul. 11
(a) Maillages des différents corps 11
(b) Maillage de calcul 11
2.3 Illustration du calcul d'intersection entre 2 maillages. 11
2.4 Index de cellules. 13
2.5 Comparaison entre le fonctionnement de la version OpenMP et celui de la version MPMD. 16
(a) Version OpenMP 16
(b) Version MPMD 16
2.6 Illustration des différentes composantes pour deux domaines de calcul. 17
2.7 Index des cellules avec prise en compte des cellules de bord. 17
2.8 Déroulement des intersections dans le cas d'une intersection non rejetée. 21
2.9 EAP et corps central d'Ariane 5. 22
(a) Etage d'Accélération à Poudre 22
(b) Corps central d'Ariane 5 22
2.10 Déroulement d'un vol Ariane 5. 22
2.11 Géométrie utilisée pour le calcul. 23
2.12 Maillage utilisé pour le calcul (11M de cellules, 31.5M de faces). 23
(a) Vue de haut du maillage 23
(b) Vue de côté 23
(c) Zoom sur la tuyère de l'EAP 23
2.13 Pression au cours du temps pour un capteur numérique. 24
2.14 Illustration du calcul d'onde de souffle. 24
2.15 Répartition des mailles dans les différents niveaux temporels et coût comparatif avec un pas de temps global. 25
2.16 Proportion de mailles et proportion de calculs associés, avec θ variant de 0 à 6. . . 26
2.17 Emplacement des mailles selon leur niveau temporel ($\theta = 5$). 26
2.18 Accélérations attendue et mesurée pour l'utilisation du schéma d'intégration temporelle adaptatif. 27
2.19 Détails d'une itération et des coûts constatés par niveau temporel ($\theta = 4$). 28
2.20 Comparaison entre la proportion de calcul théorique (*Prop. th.*) et la proportion de calcul mesurée (*Prop. mes.*). 29
2.21 Accélération de la version OpenMP (une courbe par valeur de θ entre 0 et 6). . . 30
2.22 Détails des performances (temps en secondes, accélération, efficacité) pour les deux cas θ =2 ou 5. 30

(a) $\theta = 2$ . . . 30
(b) $\theta = 5$ . . . 30
2.23 Evolution dans les classes temporelles au cours du calcul. . . . 31
(a) Proportion des mailles . . . 31
(b) Proportion du coût de calcul . . . 31
(c) Coût d'une itération . . . 31
2.24 Projection des différents domaines sur la géométrie. . . . 32
2.25 Résultats obtenus pour une configuration avec 1 processus par nœud ($\#ppn = 1$). 34
2.26 Résultats obtenus pour une configuration avec 2 processus par nœud ($\#ppn = 2$). 35
2.27 Accélération relative par rapport au calcul sur un nœud. . . . 36
(a) 1 processus par nœud . . . 36
(b) 2 processus par nœud . . . 36
2.28 Accélération relative par rapport au calcul sur un nœud avec un pas de temps global. . . . 37
(a) 1 processus par nœud . . . 37
(b) 2 processus par nœud . . . 37
2.29 Proportion de synchronisation lorsque θ varie. . . . 37
(a) 4 nœuds . . . 37
(b) 8 nœuds . . . 37
(c) 16 nœuds . . . 37
2.30 Proportion de synchronisation lorsque le nombre de nœuds varie. . . . 38
(a) $\theta = 1$ . . . 38
(b) $\theta = 4$ . . . 38
(c) $\theta = 6$ . . . 38
2.31 Trace d'exécution pour $\theta = 2$ et 8 nœuds (1 processus par nœud). . . . 39
2.32 Trace d'exécution pour $\theta = 4$ et 8 nœuds (1 processus par nœud). . . . 39
2.33 Géométrie et maillages utilisés. . . . 41
(a) Géométrie . . . 41
(b) Maillages . . . 41
2.34 Illustration du calcul de séparation des EAP. . . . 41
2.35 Evolution de l'importance des intersections au cours du calcul. . . . 42
2.36 Coût du calcul des intersections en fonction du nombre de cœurs alloués et du niveau de robustesse utilisé. . . . 43
2.37 État des processus (fréquence de calcul des intersections faible). . . . 44
2.38 État des processus (fréquence de calcul des intersections moyenne). . . . 44
2.39 État des processus (calcul des intersection systématique). . . . 44

3.1 Connaissance globale du DAG. . . . 50
3.2 Communications explicites. . . . 50
3.3 Ordonnanceur dans StarPU. . . . 53
3.4 Ordonnanceur avec une politique "prio" dans StarPU. . . . 54
3.5 Entités de calcul (EC) et composantes considérées par la version symbolique. . . 57
3.6 Détails des accès provoqués par les différentes fonctions de génération de tâches. . 58
3.7 Entités de calcul et composantes considérées par la version symbolique avec distinction entre les cellules de bord et les cellules intérieures. . . . 59
3.8 Lien entre les différentes composantes. . . . 60
3.9 Détails des accès provoqués par les différentes fonctions de génération de tâches. . 60

3.10 Scénario d'insertion des tâches : comparaison entre une version qui ne distingue pas les bords des ECs (à gauche) et une version qui les distingue (à droite). . . . 62
(a) foreach_ci_fi(Ci : R ; Fi : RW) . . . 62
(b) foreach_ci_fi(Ci : R ; Fi : RW) . . . 62
(c) foreach_ci_cj_fij(Ci,Cj : R ; Fi : RW) . . . 62
(d) foreach_ci_cj_fij(Ci,Cj : R ; Fi : RW) . . . 62
(e) foreach_ci_fi(Ci : RW ; Fi : R) . . . 62
(f) foreach_ci_fi(Ci : RW ; Fi : R) . . . 62
(g) foreach_ci_cj_fij(Ci,Cj : RW ; Fi : R) . . . 62
(h) foreach_ci_cj_fij(Ci,Cj : RW ; Fi : R) . . . 62
3.11 Comparaison de deux stratégies pour foreach_ci_cj_fij pour le même scénario d'insertion. . . . 63
(a) 1 tâche par couple (i,j) . . . 63
(b) 2 tâches par couple (i,j) . . . 63
3.12 Entités de calculs entre deux CEs. . . . 66
3.13 Données triées et position des handles pour les cellules. . . . 67
3.14 Données triées et position des handles pour les faces. . . . 67
3.15 Utilisation d'une fonction de génération de tâche. . . . 68
3.16 Exemple d'implémentation de tâche. . . . 69
3.17 Réduction du coût du calcul des dépendances via l'utilisation de handles symboliques. . . . 72
3.18 Extrait de DAG où les chaînes sont mises en évidence. . . . 72
3.19 DAG pour un calcul avec 6 CEs et $\theta = 2$. . . . 74
3.20 Trace sans utilisation des priorités (128 ECs, $\theta = 4$: t=15.565s). . . . 75
3.21 Trace avec utilisation des priorités (128 ECs, $\theta = 4$: t=14.184s). . . . 75
3.22 Évolution du nombre de tâches prêtes en fonction de la stratégie de priorité. . . . 75
3.23 Influence de la stratégie de priorité avec une configuration (16×1). . . . 78
3.24 Temps d'insertion des tâches et temps CPU de la partie solveur en fonction du nombre d'ECs pour les 16 sous-itérations. . . . 79
3.25 Influence de l'utilisation des tâches parallèles sans stratégie de priorité. . . . 79
3.26 Influence de l'utilisation des tâches parallèles avec utilisation des priorités. . . . 80
3.27 Trace avec utilisation des priorités, colorié en fonction des sous-itérations. $\theta = 4$, 16 sous-itérations, 128 ECs, configuration de calcul (16×1) . . . 81
3.28 2 domaines de calcul avec 2 ECs chacun. . . . 83
3.29 Temps passé (en secondes) dans le solveur pour différentes configurations de calcul. 84
(a) 8 ECs par processus . . . 84
(b) 16 ECs par processus . . . 84
(c) 32 ECs par processus . . . 84
(d) Version MPI+OpenMP . . . 84
3.30 Temps passé (en seconde) pour chaque processus. 6 processus sont utilisés, avec 8 ECs par processus et une configuration (4×4). . . . 84
3.31 Temps pour une itération du solveur aérodynamique. . . . 86
3.32 Proportion du temps passé dans les différents états. . . . 87

Chapitre 1

Introduction et contexte de la thèse

Sommaire

1.1 Le besoin en simulation de l'industrie aérospatiale pour la conception des lanceurs . . . 1
1.2 Contexte du HPC moderne . . . 3
1.3 Positionnement, objectifs et plan de la thèse . . . 4

1.1 Le besoin en simulation de l'industrie aérospatiale pour la conception des lanceurs

Chez Airbus DS, l'utilisation de la simulation numérique est courante depuis une trentaine d'années. En effet, dans beaucoup de cas, l'expérience réelle n'est pas possible car beaucoup trop coûteuse ou trop difficile à mettre en place. Par exemple, Ariane 5, qui fait partie des cas d'étude de cette thèse, est un lanceur dont la hauteur est de 55m et de diamètre 5.4m. Sa vitesse dépasse Mach 20 au cours du vol.

La soufflerie est souvent utilisée pour faire des expériences mais c'est aussi une ressource coûteuse et pas toujours adaptée pour les lanceurs. Il existe différents types de souffleries de tailles diverses et aux caractéristiques différentes. On distingue les souffleries continues des souffleries à rafales. Parmi les souffleries continues, on peut citer S1MA et S2MA situées à Modane. S1MA permet de tester des maquettes faisant jusqu'à 4 mètres d'envergure. L'écoulement d'air autour de la maquette peut aller jusqu'à Mach 1 : c'est la seule soufflerie au monde qui permette de faire des expériences dans de telles conditions.

S2MA permet elle de monter jusqu'à un nombre de Mach égal à 3.1, mais pour des objets avec une envergure plus petite.

S4 est une soufflerie à rafales qui permet d'atteindre un nombre de Mach égal à 12. C'est une soufflerie hypersonique adaptée pour les véhicules spatiaux et les planeurs de rentrée dans l'atmosphère. Les limitations techniques font que l'essai ne peut durer que de 25 à 90 secondes et au maximum 5 rafales peuvent être effectuées au cours d'une journée.

Si ces différents types de souffleries sont très utiles, elles ne permettent pas de couvrir l'ensemble du domaine de vol d'un lanceur. De plus, avant d'aller en soufflerie, il faut disposer d'une maquette et donc être déjà suffisamment avancé dans la conception du lanceur. Pour certains cas pratiques, il peut y avoir en outre des effets indésirables dus au test en soufflerie : en plus des effets d'échelle, la soufflerie elle-même peut avoir une influence sur les résultats. Dans le cadre d'une confrontation entre résultats de simulation et essais en soufflerie, de meilleures résultats

peuvent parfois être obtenus en modélisant la soufflerie elle-même. De plus, certaines expériences ne peuvent pas être réalisées en soufflerie et nécessitent de faire des tests grandeur nature. Le Banc d'Essai des Accélérateurs à Poudre a par exemple permis de tester et de qualifier les EAP d'Ariane 5. Il s'agit d'allumer un EAP scellé au sol et de vérifier différents paramètres physiques. Étant donnée donc la difficulté de réaliser certaines expériences, la simulation numérique s'est vite imposée dans l'industrie aérospatiale. Elle est utilisée conjointement aux expériences, mais dans certains cas les conditions de vol ne peuvent pas être recréées au sol et seule la simulation numérique permet d'obtenir des informations prédictives.

Tout d'abord, une géométrie représentant l'objet que l'on souhaite modéliser doit être réalisée. A partir de cette géométrie, un maillage est conçu : il s'agit d'une discrétisation spatiale. Puis, en partant d'une solution initiale, une solution est calculée en utilisant un modèle physique. La qualité de la solution dépend de la finesse du maillage ainsi que du modèle utilisé : certains ont un coût de calcul prohibitif comme pour la simulation numérique directe (DNS [Ors70]). La simulation des grandes échelles (LES [Ger99]) est aussi inaccessible aujourd'hui pour des problèmes industriels en vrai grandeur.

La résolution des equations de Navier-Stokes [Cho68] pour les écoulements autour des lanceurs est une méthode couramment utilisée pour résoudre ce genre de problème. Cependant ce type de modélisation ne permet pas en général de traiter correctement les phénomènes de turbulence. Pour pouvoir traiter plus finement ces problèmes, des méthodes hybrides RANS/LES sont alors utilisées [Ger99].

Pour l'industrie spatiale, l'étude des phénomènes instationnaires a toujours eu une place importante, plus que dans le reste du domaine aéronautique, et ce notamment parce que certains problèmes nécessitent une solution en temps. En raison des vitesses des objets traités, l'aérodynamique utilisée est compressible. Les chocs sont aussi très présents. Ces spécificités font qu'il est nécessaire d'utiliser des approches particulières pour effectuer les calculs.

Pour simuler les écoulements autour des lanceurs, le code de mécanique des fluides (CFD) FLUSEPA a été développé depuis presque 30 ans par ADS. Sa spécificité initiale était liée à la modélisation des séparations d'étages [Bre91, Bre95]. C'est un phénomène particulièrement critique pour les lanceurs et aucun code du commerce ne permet de traiter de manière satisfaisante ce phénomène. Le code FLUSEPA est aussi adapté au traitement de différents problèmes au delà des séparations d'étages. Il est notamment utilisé pour la rentrée atmosphérique [CBP07], le calcul des ondes de souffle au décollage, l'étude des phénomènes de turbulence pour les lanceurs [Pon15]...

Autant que possible, l'industriel qui utilise la simulation numérique préfère se référer à des logiciels sur étagère, mais pour les phénomènes qui ne sont pas traités correctement par ces derniers, il est nécessaire de garder et de faire progresser les compétences en interne. L'originalité principale du code FLUSEPA provient de sa technique de chevauchement de maillages qui permet de simplifier la création de maillages et permet de traiter efficacement le problème de séparation d'étages.

Un autre problème pour l'utilisateur industrielle est la disponibilité de puissance de calcul. De nombreux calculs doivent être lancés en même temps : il n'est pas forcement possible de disposer régulièrement de l'intégralité d'une machine de calcul haute performance. Le coût des calculs du quotidien doit donc rester convenable : si des méthodes permettent d'effectuer les mêmes calculs à un coût moindre, elles sont donc d'un grand intérêt. Dans le code FLUSEPA cela se traduit par exemple par l'utilisation d'une méthode d'intégration temporelle adaptative explicite qui est assez peu utilisée par ailleurs [KBW92, Bre93].

Afin d'accéder à des simulations plus fidèles à la réalité, il est nécessaire aussi de travailler sur des maillages plus fins avec des modèles plus précis que ceux qui sont actuellement utilisés. Ce

besoin implique alors de pouvoir disposer d'une version parallèle de FLUSEPA passant vraiment à l'échelle sur un grand nombre de cœurs de calcul.

1.2 Contexte du HPC moderne

Le classement TOP500 [2], qui recense les 500 supercalculateurs les plus puissants de la planète a été publié pour la première fois en 1993. L'ordinateur classé en première position à l'époque était le CM5/1024 composé de 1024 cœurs de calculs pour une puissance total de 57 GFlops. En novembre 1994, une machine japonaise nommée Numerical Wind Tunnel (qui signifie littéralement soufflerie numérique) prend la première place du classement. Dédiée aux simulations d'écoulements de fluides pour le National Aerospace Laboratory of Japan, cette machine pris la première place grâce à 140 processeurs vectoriels. L'utilisation de machines de calcul puissantes pour la mécanique des fluides est donc survenue très tôt.

Au cours des années, l'architecture des machines a beaucoup évolué. Au sein des processeurs "grand public", les multi-cœurs sont devenus communs dès 2005. Jusque-là, il était encore possible d'obtenir des gains de performance en augmentant la fréquence d'horloge d'un petit nombre de cœurs de calcul. Concernant la structure des machines de calcul, l'utilisation de la mémoire distribuée est vite devenue nécessaire. Au lieu d'avoir une seule mémoire partagée accessible par toutes les unités de calcul, on a vu apparaître des clusters où les unités de calcul se repartissent au sein de plusieurs nœuds à mémoire partagée et qui sont connectés par un réseau haut débit. Pour partager des données entre nœuds, des communications doivent être effectuées.

Si à une époque, des processeurs spécifiques (souvent vectoriels) ont pu être utilisés au sein des supercalculateurs, la tendance actuelle est à l'adaptation de technologies grand public. Ainsi, les GPUs [OHL+08] ont été détournés de leur usage principal (carte graphique) pour servir au calcul scientifique : le succès de cette utilisation a fini par pousser les constructeurs à proposer des versions dédiées au calcul scientifique de leurs GPUs. Aujourd'hui, le type de machine qui domine le marché des calculateurs utilisés par les entreprises est donc le cluster de nœuds multi-cœurs, éventuellement couplés d'accélérateurs de calcul de type GPGPU ou Xeon Phi [JR13].

L'évolution des machines est bien plus rapide que celle des codes de calcul. Le développement d'un code industriel est une entreprise compliquée, qui nécessite de nombreuses compétences différentes. Ainsi, il faut tout d'abord modéliser la physique considérée à l'aide d'équations, disposer de techniques numériques pour résoudre les équations et enfin le mettre en œuvre de manière efficace sur un supercalculateur en maîtrisant adroitement l'informatique du HPC.

La volonté d'effectuer des simulations numériques toujours plus fidèles pousse à l'utilisation de calculateur de plus en plus puissants. Cependant, certaines avancées technologiques nécessitent l'adaptation du code déjà existant. Pour exploiter les machines à mémoire distribuée, le code doit être écrit en conséquence. De même, l'utilisation de GPUs ne peut se faire sans adaptation du code. Les évolutions à apporter au code peuvent donc nécessiter un temps de développement très long et il est nécessaire que chaque évolution soit validée. Dans le même temps, les machines évoluent vite et les efforts qui ont été faits pour optimiser les performances sur une machine donnée peuvent se révéler inutiles pour une machine future.

Si on prend le cas des accélérateurs de type GPU, il est nécessaire d'écrire du code dédié. Sous cette forme, les transferts entre la mémoire centrale et la mémoire de l'accélérateur doivent être décrits "à la main". Si un autre type d'accélérateur est ajouté à la machine, le code doit être modifié pour traiter ce cas. Il n'est pas envisageable d'écrire du code aussi spécifique dans le cadre d'un développement industriel pour cibler une architecture qui ne sera peut être plus la norme

2. http ://top500.org/

dans un futur proche. Si l'on souhaite pouvoir à la fois utiliser la puissance des machines actuelles et être prêt pour le futur, il est donc nécessaire de pouvoir disposer d'une couche d'abstraction (virtualisation) qui permettra une adaptation rapide du code pour une machine future avec une portabilité des performances raisonnable.

1.3 Positionnement, objectifs et plan de la thèse

Au départ de ce travail, FLUSEPA n'était parallélisé que pour les machines à mémoire partagée via l'utilisation d'OpenMP. Le code était auparavant utilisé sur des machines vectorielles de type CRAY, mais ce type d'architecture a fini par ne plus être accessible, ce qui a motivé le développement de la version OpenMP.

La première année de la thèse a servi à la finalisation d'une version distribuée hybride MPI+OpenMP. Cette version est devenue la version de référence du code et est désormais utilisée en production. Un calcul URANS [Ben13] de séparation des EAP d'Ariane 5 a été notamment réalisé avec cette version. Elle a aussi servi pour les calculs de la thèse de Grégoire Pont [Pon15] concernant les modèles de turbulence instationnaires.

Dans le cadre de la présente thèse, les performances de cette version ont été analysées dans le but de concevoir et de construire une version en tâches prête pour les architectures futures. Deux aspects justifient particulièrement le passage à un paradigme de programmation en tâches. D'une part le solveur et son intégration temporelle adaptative pourrait bénéficier d'une meilleure expression des dépendances. D'autre part, la simulation de corps en mouvement relatif passe par deux opérations très distinctes et on doit pouvoir exploiter l'asynchronisme sous-jacent.

Le chapitre 2 décrit la parallélisation du code en version MPI+OpenMP. Les méthodes numériques du code sont rappelées et les algorithmes utilisés sont décrits en détails, et plus précisément celui du schéma d'intégration temporelle adaptatif. La mise en œuvre de la version MPI+OpenMP est ensuite présentée avec des performances analysées pour deux cas de simulations industrielles.

Dans le chapitre 3, on introduit le modèle de programmation en tâches et les supports d'exécution. Ensuite, on décrit les étapes nécessaires à la construction d'une version en tâches du solveur aérodynamique du code FLUSEPA. Les étapes et les optimisations menant à une version performante en mémoire partagée ainsi qu'en mémoire distribuée sont présentées en détails. Nous utilisons le runtime StarPU [ATNW11].

Enfin, le chapitre 4 fait un bilan du travail effectué dans cette thèse et présente les différentes perspectives possibles pour les développements futurs de FLUSEPA.

Chapitre 2

Conception et mise en œuvre d'une première version MPMD basée sur un parallélisme hybride MPI+OpenMP

Sommaire

2.1 Cadre numérique et description algorithmique du problème 5
2.1.1 Schéma global et différentes étapes du calcul 6
2.1.2 Solveur aérodynamique 6
2.1.3 Corps en mouvement relatif 10
2.2 Description des différents niveaux de parallélisme dans FLUSEPA . 12
2.2.1 Structures de données existantes et évolutions apportées 13
2.2.2 Mise en œuvre parallèle MPMD avec un parallélisme hybride MPI+OpenMP 14
2.2.3 Solveur aérodynamique parallèle 15
2.2.4 Prise en compte des déplacements de maillage 18
2.3 Validation expérimentale et analyse de performance 22
2.3.1 Calcul d'une onde de souffle 23
2.3.2 Calcul de la séparation des EAP 41

2.1 Cadre numérique et description algorithmique du problème

La méthode des volumes finis [VM07] est utilisée pour discrétiser les équations simulées par le code FLUSEPA. Il s'agit d'une formulation intégrale des lois de conservation discrétisée directement dans l'espace physique. Dans FLUSEPA, on cherche à calculer la solution numérique de la décomposition de Reynolds appliquée aux équations de Navier-Stokes compressibles (RANS [Ger99]), ce qui conduit à la résolution de

$$\frac{d}{\partial t} \iiint_{\Omega_{CV}} \mathbf{w} d\Omega = - \oiint_{A_{CV}} \mathbf{F} \mathbf{n} dS + \iiint_{\Omega_{CV}} \mathbf{S} d\Omega \tag{2.1}$$

où Ω_{CV} est un volume de contrôle fixé (une cellule 3D) avec une frontière A_{CV} (des faces 2D), $\mathbf{n}$ étant le vecteur unitaire sortant, $\mathbf{w}$ le vecteur correspondant aux variables conservatives, $\mathbf{F}$ la densité de flux et $\mathbf{S}$ le vecteur correspondant aux termes sources.

Le solveur manipule donc principalement des cellules et des faces. Les valeurs des champs (pression, température...) sont calculées pour chaque cellule et les flux sont évalués entre les faces des cellules.

L'avantage principal de cette discrétisation est sa conservativité, ce qui permet une très bonne approximation de la physique. L'utilisation de la méthode de Godounov [GZI+79] pour résoudre le problème de Riemann permet d'obtenir une grande robustesse et ce même quand des ondes de chocs sont présentes. Pour prendre en compte le mouvement des mailles lors de séparation d'étages, une formulation ALE [HAC74] est utilisée : quand les flux sont calculés, la vitesse intrinsèque de chaque cellule est prise en compte. Le maillage de calcul est obtenu en intersectant plusieurs maillages (ceux des corps en mouvement) via des intersections géométriques. Contrairement aux méthodes CHIMERE [GMP95, KLC94] classiques, aucune interpolation des valeurs n'est effectuée.

Les équations RANS pour les écoulements 3D compressibles instationnaires et réactifs sont résolues autour des corps en mouvements relatifs. Le déplacement de volume est donc aussi intégré dans le schéma numérique. Quand le mouvement relatif entre deux maillages est faible, calculer l'intersection n'est pas obligatoire et ainsi, le changement de volume peut être évalué pratiquement (grâce au calcul du volume balayé par chaque face) à la précision désirée. Pour les autres équations, la formulation est classique.

2.1.1 Schéma global et différentes étapes du calcul

Les deux principales étapes de calcul effectuées par le code FLUSEPA à chaque itération sont résumées dans l'Algorithme 1. Le solveur aérodynamique et le calcul des intersections sont reliés par le calcul de la cinématique. Pendant les calculs aérodynamiques, des efforts sont calculés autour des corps. En utilisant ces efforts, une cinématique est calculée. Si les corps ont bougé de manière trop importante depuis la dernière intersection, les maillages sont déplacés en prenant en compte la cinématique calculée. Sinon, la loi de conservation géométrique est utilisée pour prendre en compte le déplacement sans recalculer une nouvelle intersection de maillages.

Algorithme 1 Une itération de FLUSEPA

1: Calcul du solveur aérodynamique
2: Calcul d'une nouvelle cinématique
3: **si** Déplacement important depuis le dernier calcul d'intersection des maillages **alors**
4:     Déplacement des corps (nouvelle cinématique)
5:     Calcul d'intersection des maillages
6: **fin si**

2.1.2 Solveur aérodynamique

Pour chaque itération du calcul, le solveur aérodynamique est utilisé. Il est basé sur une formulation "centrée aux éléments" pour laquelle seules les faces et les cellules sont utiles. Il n'y a pas de valeurs aux nœuds du maillage. L'état dans chaque cellule est caractérisé par des variables dites extensives et intensives. Les variables *extensives* sont celles qui sont régies par les lois de conservation (par exemple la masse, le volume). Les variables *intensives* caractérisent la valeur en tout point (par exemple la température, la densité,...). Les variables extensives sont liées aux variables intensives. Il est donc possible d'actualiser la valeur des variables intensives à partir de celles des variables extensives.

La discrétisation spatiale est basée sur une méthode MUSCL [VL03] qui est constituée de 3 grandes parties :

- *l'algorithme de reconstruction* qui permet le calcul des variables en tout point dans chaque cellule et en particulier sur ses faces ;
- *l'algorithme de limitation* qui assure la positivité du schéma (et donc l'absence de création de maxima locaux) ;
- *l'intégration des flux* qui permet le calcul des flux pour chaque face des cellules et qui permet la mise à jour des variables extensives.

L'algorithme de reconstruction est basé sur le calcul d'une approximation des gradients des variables intensives au centre de gravité de chaque volume de contrôle. L'opérateur numérique pour le calcul des gradients est basé sur la formule de Green et n'a besoin que des voisins directs. L'algorithme de limitation est une généralisation multidimensionnelle de l'algorithme de limitation de Van Leer [VL03].

Deux schémas d'intégration temporelle sont disponibles dans FLUSEPA. Un premier schéma d'intégration temporelle est implicite. Il est utilisé principalement pour les calculs stationnaires ou quasi-stationnaires. Il a été parallélisé dans la version MPI+OpenMP mais il n'est pas considéré dans ce manuscrit. Le second schéma d'intégration temporelle est explicite et est basé sur un pas de temps local de manière à être efficace pour les simulations instationnaires. C'est ce dernier qui a été au centre de ce travail de thèse. Sa parallélisation va être présentée en détails en version MPI+OpenMP, ainsi qu'en version basée sur une expression du parallélisme via un graphe de tâches et sur une exécution via un support d'exécution.

2.1.2.1 Intégration temporelle

L'algorithme d'intégration temporelle adaptatif est utilisé pour calculer l'évolution de chaque cellule à un pas de temps proche de son pas de temps maximum. Il est particulièrement adapté quand des petites échelles de temps physique doivent être utilisées. Les solveurs implicites perdent leur avantage dans cette situation à cause de leur coût qui peut devenir prohibitif comparé à celui des solveurs explicites. D'autre part, ces deux types de solveurs ont le désavantage d'utiliser a priori un pas de temps global.

L'intégration temporelle adaptative a été appliquée à différents types de problèmes dans le passé et elle ne se limite pas aux équations de Navier-Stokes. L'un des principaux problèmes à résoudre a été l'aptitude à échanger des informations précises entre deux cellules intégrées à des pas de temps différents. Les premières méthodes ne découplaient pas la discrétisation spatiale de la discrétisation temporelle et nécessitaient plusieurs domaines de calcul. Oshers et Sanders [OS83] ont proposé une technique pour déterminer la valeur à l'interface en 1D, mais elle nécessitait de stocker des résultats intermédiaires. Puis, Löhner [LMPZ85], Pervaiz et Baron [PB88] ont mis au point une méthode où des domaines 2D avec des pas de temps différents se recouvraient afin d'obtenir les valeurs aux interfaces. Berger [Ber87] a introduit une approche qui ne nécessitait pas de recouvrement. Kleb [KBW92] a finalement proposé une approche générale séparant l'intégration temporelle de la discrétisation spatiale et il a détaillé les performances obtenues pour des ordinateurs vectoriels pour des équations 2D de Navier-Stokes. Il a aussi vérifié que la qualité de la solution était bien préservée.

Lorsqu'on utilise une formulation temporelle explicite, le pas de temps maximum autorisé dans une cellule est donné par une condition de stabilité dite de CFL (Courant-Friedrichs-Lewy) qui est une grandeur dépendant principalement du volume de la cellule. Pour un solveur explicite classique, le pas de temps est contraint par la cellule la plus lente (celle pour laquelle le pas de temps est le plus faible lorsqu'on respecte la condition CFL).

Comme on traite des cas complexes en taille réelle, on utilise des maillages dont la résolution n'est pas uniforme ; utiliser le pas de temps le plus faible pour les grosses cellules seraient très pénalisant. Un des avantages principaux de l'intégration temporelle adaptative est qu'elle permet de calculer chaque cellule avec une condition CFL proche de 1, mais toujours inférieure, et ce en classant les cellules dans des niveaux temporels (numérotés de 0 à θ).

La méthode est particulièrement adaptée quand la propagation d'un choc est étudiée ou quand des échelles de temps faibles sont nécessaires. Elle pourrait aussi s'avérer utile pour les simulations VLES [Ger99] où de très faibles CFL sont souvent utilisées dans les zones d'intérêt. Cependant pour être efficace, il est nécessaire d'avoir peu de cellules avec un pas de temps faible car sinon, il n'y a pas de réel gain comparé à un pas de temps global.

Algorithme

La méthode d'intégration temporelle adaptative utilisée est décrite dans l'Algorithme 2. Elle utilise un schéma prédicteur-correcteur, la méthode de Heun, et est donc à l'ordre 2. Dans cet algorithme, les différentes étapes de calcul sont celles citées plus haut.

Ligne 1, le pas de temps maximum autorisé est calculé pour toutes les cellules. La cellule la plus lente est repérée ce qui définit Δt, le pas de temps minimal du calcul. A la ligne 2, à partir de Δt et de son pas de temps maximum autorisé, chaque cellule se voit affecter un niveau temporel. Au sein d'un niveau temporel τ, les cellules partagent un même pas de temps qui est $2^\tau * \Delta t$.

Une itération de l'algorithme est composée de plusieurs sous-itérations. Il y a 2^θ sous-itérations (numérotées de 1 à 2^θ, 0 étant l'état initial, sur la Figure 2.1), θ étant le niveau temporel des cellules les plus rapides, c'est à dire celles qui n'ont besoin que d'une sous-itération pour atteindre le temps physique de la fin de l'itération.

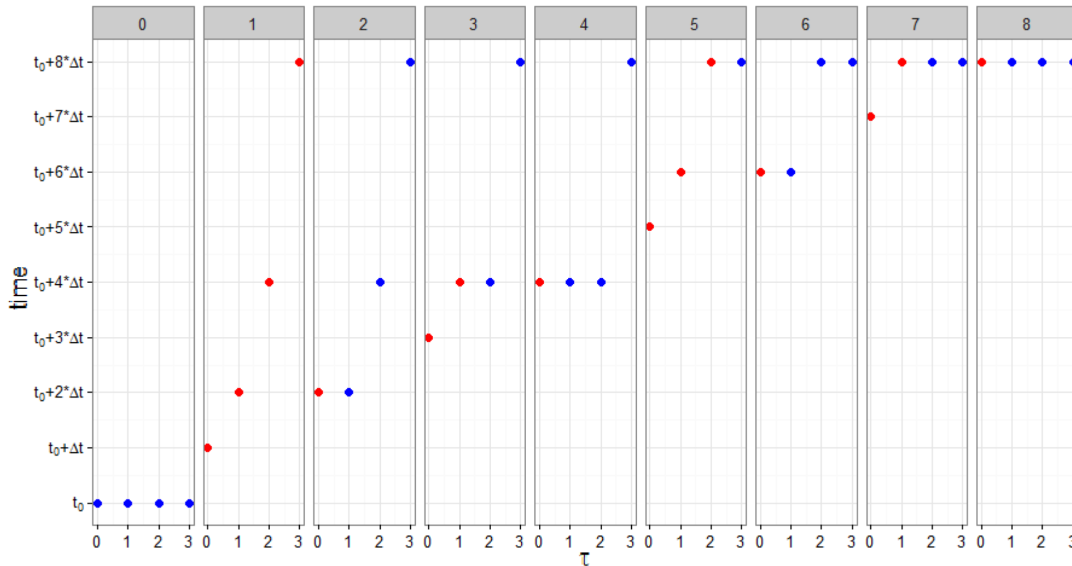


FIGURE 2.1 – Temps atteint après chaque sous-itération pour chaque niveau temporel τ donné en abscisse ($\theta = 3$, 8 sous-itérations).

Les niveaux temporels considérés au cours d'une sous-itération sont déterminés des lignes 5 à 9. Par exemple pour $\theta = 3$, τ prendra successivement les valeurs 3,0,1,0,2,0,1,0. Cette valeur correspond au niveau temporel le plus élevé concerné par la sous-itération, les niveaux inférieurs

Algorithme 2 Intégration temporelle adaptative de FLUSEPA : une itération du solveur

1: Calcul du pas de temps
2: Classification des cellules dans un niveau temporel
3: *Boucle d'intégration temporelle adaptative :*
4: **pour** sous-itération=1 **à** 2^θ **faire**
5: $\quad \tau = 0$
6: $\quad$ **pour** $tmp = 1$ **à** θ **faire**
7: $\quad\quad$ **si** $(mod(\text{sous-itération}-1, 2^{tmp}) == 0)$ **alors**
8: $\quad\quad\quad \tau = tmp$
9: $\quad\quad$ **fin si**
10: $\quad$ **fin pour**
11: $\quad$ **si** sous-itération>1 **alors**
12: $\quad\quad$ Correction des variables intensives $(\{0...\tau\})$
13: $\quad\quad$ Interpolation des variables intensives $(\tau + 1)$
14: $\quad$ **fin si**
15: $\quad$ *Prédicteur :*
16: $\quad$ Calcul des gradients $(\{0...\tau\})$
17: $\quad$ Limitation et reconstruction des flux $(\{0...\tau\})$
18: $\quad$ Repositionnement des flux $(\tau + 1)$
19: $\quad$ Solveur de Riemann $(\{0...\tau\})$
20: $\quad$ Sommation des flux sur les cellules $(\{0...\tau\})$
21: $\quad$ **si** $\tau \neq \theta$ **alors**
22: $\quad\quad$ Repositionnement des variables intensives $(\tau + 1)$
23: $\quad$ **fin si**
24: $\quad$ **pour** $\tau' = \tau$ **à** 0 **faire**
25: $\quad\quad$ *Fin du prédicteur pour* τ' *:*
26: $\quad\quad$ Prédiction des variables extensives (τ')
27: $\quad\quad$ Prédictions des variables intensives (τ')
28: $\quad\quad$ *Correcteur :*
29: $\quad\quad$ Calcul des gradients (τ')
30: $\quad\quad$ Limitation et reconstruction des flux (τ')
31: $\quad\quad$ Interpolation des flux (τ')
32: $\quad\quad$ Solveur de Riemann (τ')
33: $\quad\quad$ Sommation des flux sur les cellules (τ')
34: $\quad\quad$ Correction des variables extensives (τ')
35: $\quad\quad$ Correction des variables intensives (τ')
36: $\quad$ **fin pour**
37: **fin pour**

étant eux-aussi traités. La Figure 2.1 montre comment évolue chaque niveau temporel pour $\theta = 3$ au cours des sous-itérations. Il y a donc 4 niveaux temporels (numérotés de 0 à 3) et 8 sous-itérations. Les cellules du niveau τ_3 ont un pas de temps 2^3 supérieur à celui du niveau temporel τ_0. Lorsqu'une classe temporelle est traitée au cours d'une sous-itération, elle est en rouge. Les cellules de niveau τ_0 sont traitées à toutes les sous-itérations. Le temps physique atteint après une itération complète du solveur est l'équivalent du temps physique atteint pour 8 itérations avec un pas de temps global ($\theta = 0$).

Pour rester consistant en temps, les calculs doivent être effectués dans un ordre spécifique. Quand on calcule un flux entre deux cellules, celles-ci doivent être au même temps physique. Les cellules ne peuvent avoir pour cellules voisines que des cellules du même niveau temporel τ, de niveaux $\tau + 1$ ou $\tau - 1$. Si une cellule est voisine d'une cellule d'un niveau temporel différent, elle sera positionnée à un temps qui garantira un calcul consistant en temps (lignes 18, 22, 30, 34). C'est en ce sens que l'ordre des calcul est strict et que chaque niveau temporel est calculé à un moment spécifique.

Évaluation des gains potentiels par rapport à un pas de temps global

Pour une répartition donnée en terme de niveaux temporels, il est possible d'estimer le gain obtenu par l'intégration temporelle adaptative. Dans cette estimation, on omet les interpolations et les extrapolations pour deux raisons : elles sont effectuées sur peu de cellules et représentent peu d'opérations. Notons $\Omega(\tau)$ l'ensemble des cellules d'un niveau temporel τ et Ω l'ensemble des cellules du domaine.

Le coût calculatoire d'un niveau temporel τ peut être estimé par $C(\tau) = 2^{\theta-\tau} * |\Omega(\tau)|$ et le coût d'une itération du solveur par $\sum_{\tau=0}^{\theta} C(\tau)$.

Il faut comparer cette valeur au coût pour atteindre le même temps physique avec un pas de temps global : dans ce cas, l'ensemble des cellules étant intégrées au pas de temps le plus faible, ce coût est de $2^\theta * |\Omega|$.

On obtient finalement le ratio suivant $\frac{2^\theta * |\Omega|}{\sum_{\tau=0}^{\theta} 2^{\theta-\tau} * |\Omega(\tau)|}$ qui est toujours supérieur à 1.

Cette estimation est une borne supérieure à cause des surcouts provenant de la méthode. En effet, plusieurs interpolations ne sont pas prises en compte, alors que leur nombre augmente avec θ. Avoir plus de niveaux temporels implique un plus grand nombre de cellules concernées par ces interpolations et les cellules avec un pas de temps faible sont interpolées plus souvent.

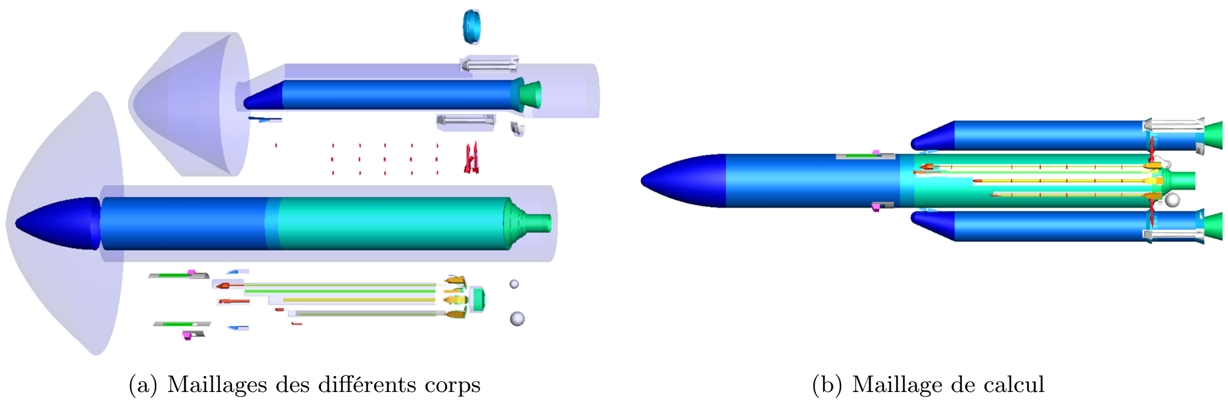
2.1.3 Corps en mouvement relatif

2.1.3.1 Intersection de maillages

Le principe de base repose sur un maillage de calcul obtenu en intersectant plusieurs maillages spécifiques qui se recouvrent. Chaque maillage a sa propre priorité et quand plusieurs maillages sont présents au même endroit géométrique, le maillage avec la plus haute priorité écrase les autres. Au frontière d'un maillage recouvrant, une intersection géométrique est donc calculée avec le maillage de plus faible priorité présent à cet endroit. Quand les cellules possèdent des conditions limites pariétales, il est possible de leur donner des priorités supérieures afin de préserver les parois. Il n'y a pas de contraintes concernant l'inclusion des maillages.

La figure 2.2 montre comment peuvent être associés ces différents maillages : chaque partie de la géométrie est maillée séparément (figure 2.2a) et les maillages sont ensuite intersectés pour obtenir le maillage de calcul (figure 2.2b).

Le calcul d'intersection se fait en deux parties :



(a) Maillages des différents corps (b) Maillage de calcul

FIGURE 2.2 – Intersection de maillages et construction du maillage de calcul.

— pour chaque face, le calcul de la proportion de sa surface qui est recouverte par un maillage de priorité supérieure ;
— le calcul de la partie de la surface d'intersection incluse dans chaque cellule et qui est utilisée pour fermer les cellules comme c'était le cas pour leur faces originales.

Cette dernière étape crée de nouvelles interfaces entre les cellules recouvrantes et les cellules coupées qui sont utilisées par le solveur aérodynamique comme des interfaces normales entre cellules d'un même maillage. Enfin, grâce à la formulation de Gauss-Green [Fed45], les caractéristiques de chaque cellule sont déterminées, à savoir le volume, le centre de gravité, les moments d'inerties.

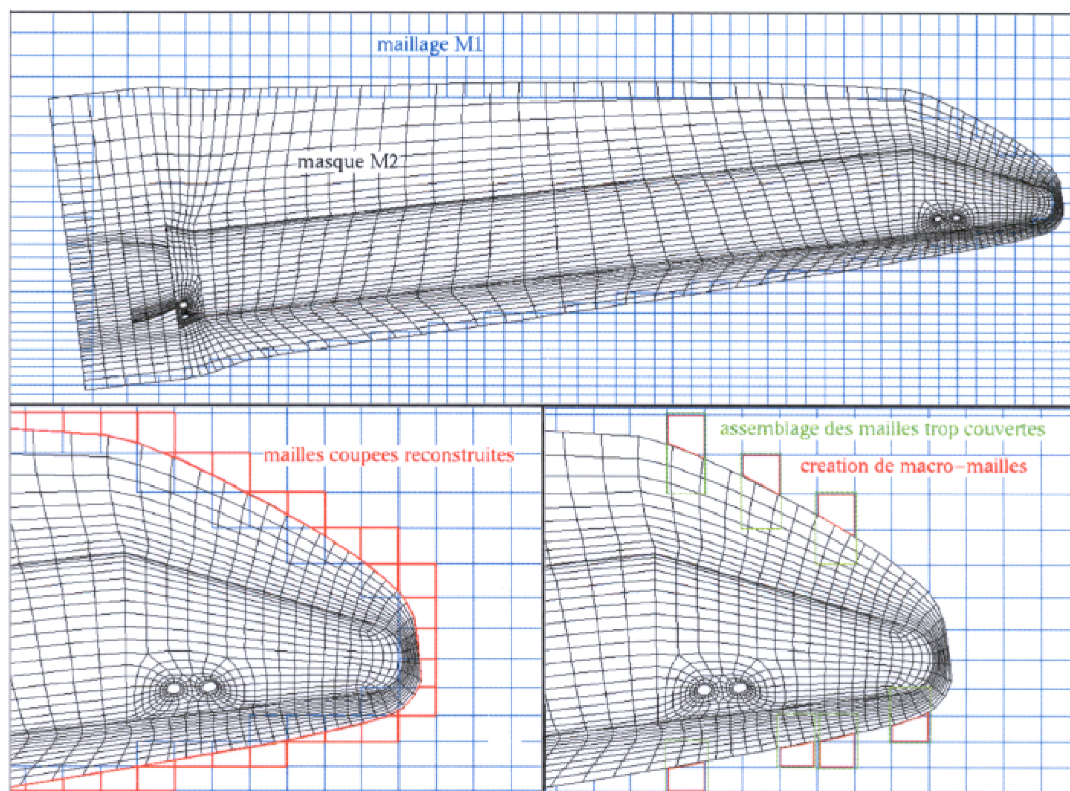


FIGURE 2.3 – Illustration du calcul d'intersection entre 2 maillages.

Après une intersection, 3 états sont possibles pour les cellules : couvertes, coupées ou non-modifiées. Les cellules couvertes sont exclues du calcul. Les cellules non modifiées sont inchangées et seront simplement utilisées telles quelles.

A cause des intersections, des cellules coupées apparaissent donc. Ces cellules sont partiellement couvertes et possèdent donc des volumes modifiés. Quand les cellules sont très recouvertes, leur volume peut devenir si petit qu'elles peuvent pénaliser grandement le solveur aérodynamique. Pour limiter ce problème, les cellules très recouvertes sont associées avec une cellule adjacente. On appelle cette cellule adjacente “cellule principale” de la cellule associée considérée. La Figure 2.3 illustre ce fait : un maillage M1 est écrasé par un maillage M2 qui est plus prioritaire. On peut voir les mailles coupées sur la deuxième partie de la figure ainsi que les petites mailles qui sont associées avec une maille principale sur la troisième.

Pour le solveur aérodynamique, les intersections sont transparentes. Les flux entre deux cellules de deux maillages différents sont traités de la même manière que ceux entre cellules d'un même maillage. Avant chaque calcul d'intersection, pour les cellules associées, les variables extensives sont redistribuées dans chaque cellule en accord avec leur volume estimé au moment de l'intersection. Après le calcul d'intersection, les cellules sont ré-assemblées et leurs variables extensives sommées.

2.1.3.2 Mouvement des corps

Pour simplifier la modélisation, chaque maillage est lié à un seul corps, mais plusieurs maillages peuvent être liés au même corps. Les corps sont indépendants et peuvent se déplacer d'un mouvement arbitraire. Un formulation ALE [HAC74] est utilisée : chaque nœud du maillage se déplace selon un vecteur directionnel qui lui est propre.

L'intégration des efforts aérodynamiques autour de chaque corps autorise le calcul de la cinématique qui va fournir le mouvement appliqué aux différents corps. Afin de ne pas calculer une intersection à chaque itération, la modification de volume est calculée via la loi de conservation géométrique. Quand le déplacement cumulé devient trop important, les intersections sont recalculées. Après une intersection, certaines cellules qui étaient partiellement coupées auparavant sont désormais totalement couvertes (et inversement). Comme le déplacement est faible entre deux calculs d'intersection, les cellules qui apparaissent après un calcul d'intersection seront associées avec une cellule qui était précédemment présente. Ce fait est exploité par l'algorithme car les cellules considérées durant le mouvement sont moins nombreuses que lors du premier calcul d'intersection. Il y a en effet une très grande différence de coût entre le premier calcul d'intersection et un calcul d'intersection qui suit un déplacement des maillages.

2.2 Description des différents niveaux de parallélisme dans FLUSEPA

Le parallélisme potentiel du code FLUSEPA provient notamment d'une certaine indépendance entre les calculs d'intersection de maillages et les calculs aérodynamiques.

Le solveur aérodynamique est actuellement la partie la plus coûteuse du code pour les cas de calcul considérés. Le parallélisme potentiel du solveur est principalement spatial car on peut partager le domaine de calcul entre plusieurs processus (approche décomposition de domaine).

A l'échelle d'un processus, et que ce soit pour le solveur aérodynamique ou le calcul d'intersection, les boucles sont écrites de manière à ce qu'elles soient parallélisables car chaque itération

est indépendante. On peut donc exploiter un parallélisme basé sur des directives OpenMP (*OMP DO*) au sein des processus en mémoire partagée.

2.2.1 Structures de données existantes et évolutions apportées

2.2.1.1 Types de données

Dans FLUSEPA, le solveur aérodynamique manipule différents types de données. On y trouve ainsi des tableaux stockant des valeurs pour les cellules et d'autres pour les faces. On peut distinguer les cellules "réelles", des cellules "virtuelles". Les cellules réelles sont les cellules du maillage alors que les cellules virtuelles sont des cellules qui servent à stocker des valeurs aux conditions limites du maillage. On dispose donc de différents tableaux pour stocker des valeurs pour les cellules réelles, pour les cellules virtuelles et enfin pour stocker les valeurs pour les faces.

2.2.1.2 Index de cellules et de faces

On utilise des index (qui sont des tableaux d'indices se rapportant aux tableaux de stockage de valeurs) pour identifier les cellules et les faces en fonction de leurs caractéristiques. Ces index disposent de "bornes" qui sont des indices de début et de fin qui permettent de parcourir les éléments présentant une caractéristique donnée.

Gestion des cellules A cause du mécanisme d'intersection, il existe plusieurs types de cellules "réelles" : les cellules non-associées, les cellules principales et les cellules associées. Pour rappel, lorsqu'une cellule a un trop faible volume ou lorsqu'elle est trop "plate", elle est associée à une cellule plus grande. Les cellules disposant de cellules associées sont dites "principales".

La Figure 2.4 montre l'organisation de l'index de cellules. Les bornes CNA et CA permettent de distinguer les cellules en fonction de leur statut (non-associée ou associée).

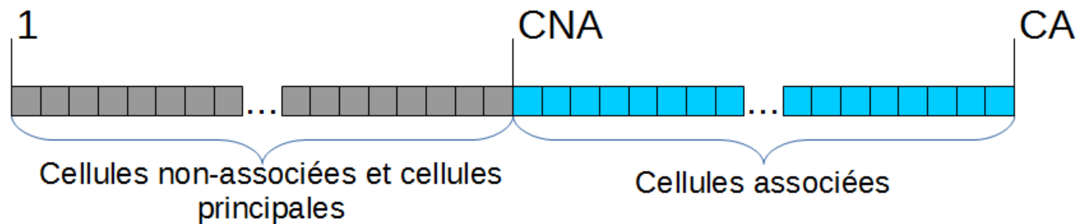


FIGURE 2.4 – Index de cellules.

Il existe aussi un tableau qui permet de récupérer la cellule principale pour chaque cellule associée.

Gestion des faces Pour faire la liaison entre les cellules et les faces, il existe un tableau multidimensionnel indicé par le numéro de face et dont les valeurs sont les numéros de cellules partageant la face. Chaque face est liée à deux cellules : une première cellule est située "à gauche de la face" et une seconde est située "à sa droite".

Les faces sont placées dans différents index. Un premier index contient toutes les faces et les discrimine selon qu'elles soient des faces entre les cellules du maillage ou des faces correspondant à des conditions limites. Il y a aussi plusieurs index concernant uniquement les faces des conditions limites. Ces index discriminent les faces selon d'autres critères (par exemple le type de paroi, le type de condition limite fluide...).

2.2.1.3 Tri des index pour la gestion de l'intégration temporelle adaptative

Comme indiqué dans la section 2.1.2, les cellules sont classées à chaque itération dans un niveau temporel. Cela se reflète sur le contenu des index qui sont triés en accord avec ces niveaux temporels. Ensuite, au sein de chaque niveau temporel, un deuxième tri a lieu pour identifier les cellules qui sont en interaction avec des cellules de niveaux temporels différents. Cette structuration est nécessaire pour garder la consistance en temps. De nouvelles bornes sont alors calculées. Pour chaque élément e (face ou cellule), on note $\tau(e)$ son niveau temporel. Le niveau temporel d'une face est le niveau temporel le plus élevé parmi celui de ses 2 cellules. On note $\tau_v(e)$ l'ensemble des valeurs des niveaux temporels présents dans le voisinage de l'élément e. $min(\tau_v(e))$ est le niveau temporel minimum parmi le voisinage de e et $max(\tau_v(e))$ est le niveau temporel maximum parmi le voisinage de e.

Pour chaque niveau temporel τ et pour chaque index, on trie les éléments de la manière suivante :
- $\forall e \in$ index(BORNE(0,τ)+1,BORNE(1,τ)) : $min(\tau_v(e)) = \tau(e) = max(\tau_v(e))$;
- $\forall e \in$ index(BORNE(1,τ)+1,BORNE(2,τ)) : $min(\tau_v(e)) < \tau(e) = max(\tau_v(e))$;
- $\forall e \in$ index(BORNE(2,τ)+1,BORNE(3,τ)) : $min(\tau_v(e)) < \tau(e) < max(\tau_v(e))$;
- $\forall e \in$ index(BORNE(3,τ)+1,BORNE(4,τ)) : $min(\tau_v(e)) = \tau(e) < max(\tau_v(e))$.

Cette organisation permet d'identifier correctement les éléments nécessaires lors des étapes de l'Algorithme 2. Par exemple, on va traiter les éléments situés entre les bornes BORNE(1,$\tau+1$) et BORNE(3,$\tau+1$) à la ligne 13.

2.2.2 Mise en œuvre parallèle MPMD avec un parallélisme hybride MPI+OpenMP

Algorithme 3 Boucle du maître

1: Collecte des métriques pour le rééquilibrage
2: **si** Déséquilibre > seuil autorisé ET pas de rééquilibrage en cours **alors**
3: Rééquilibrage asynchrone
4: **fin si**
5: Réception des efforts sur les différents corps
6: Mise à jour de la cinématique
7: Détermination des actions à effectuer
8: Envoi des actions à effectuer aux autres processus
9: Exécution des actions de l'itération courante
10: **si** Fin du calcul **alors**
11: Quitter la boucle
12: **fin si**

Le code FLUSEPA met en œuvre deux types de calcul : les calculs aérodynamiques et les calculs d'intersection. Ces calculs sont liés par le calcul de la cinématique mais restent indépendants. Un des objectifs de la parallélisation de FLUSEPA est d'exploiter cette indépendance en permettant un calcul asynchrone des intersections pendant les calculs aérodynamiques.

Nous avons donc choisi une approche MPMD pour paralléliser FLUSEPA avec 3 types de processus : processus maître qui réalise le contrôle, processus pour les calculs aérodynamiques et processus pour les calculs d'intersection. Pour initier un calcul global, il faut un processus maître, au moins un processus pour les calculs aérodynamiques et zero, un ou plusieurs processus pour les calculs d'intersection.

Le processus maître coordonne donc l'ensemble de la simulation en décidant si un calcul d'intersection doit être lancé ou pas ; il est aussi en charge du calcul de la cinématique qui est une opération peu coûteuse. Étant donné que le niveau temporel des mailles au sein d'un domaine change, la charge évolue au sein des différents processus aérodynamiques. Le maître se charge donc aussi de l'équilibrage de charge et des entrées/sorties.

Une version simplifiée des opérations du maître est décrite dans l'Algorithme 3. Après chaque itération du solveur, le maître doit prendre des décisions concernant le rééquilibrage, la production éventuelle de sorties intermédiaires, la gestion des mouvements... Cet ensemble d'opérations s'effectue en 3 temps :

— tout d'abord, toutes les actions à effectuer sont recensées ;
— ensuite le maître prévient les autres processus des actions qui seront effectuées ;
— et enfin, toutes les actions sont effectuées avant la prochaine itération du solveur.

La Figure 2.5a illustre le déroulement du calcul pour 3 itérations dans la version d'origine OpenMP : un seul processus se charge de toutes les opérations. Un calcul d'intersection a lieu après la deuxième itération. La Figure 2.5b montre le même calcul avec la version MPMD : 3 processus dédiés aux calculs aérodynamiques sont présents, 1 processus pour les calculs d'intersection ainsi qu'un processus maître. Sur cette figure, lors de l'étape "Contrôle (1)", la décision de pré-calculer une intersection est prise. La nouvelle topologie est appliquée lors de l'étape "Contrôle (2)". Cette organisation permet de gagner du temps : la parallélisation par découpage spatial du solveur aérodynamique réduit tout d'abord le temps passé dans le solveur et la possibilité de calculer les intersections de manière asynchrone peut permettre un bon recouvrement de leur temps de calcul.

2.2.3 Solveur aérodynamique parallèle

2.2.3.1 Parallélisme spatial par décomposition de domaine

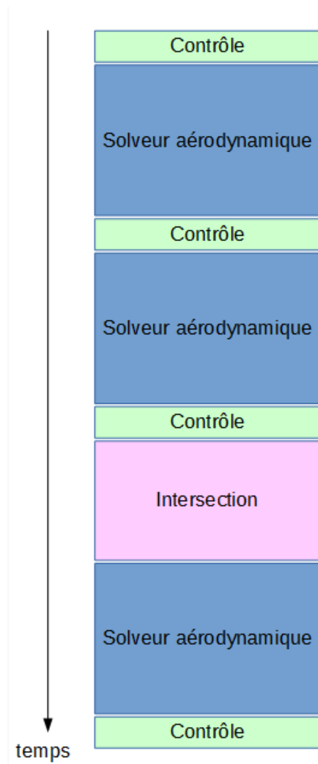
Pour pouvoir effectuer les calculs de manière parallèle en mémoire distribuée, nous utilisons une décomposition de domaine. Cette décomposition est effectuée sur le maître et utilise le partitionneur de graphes et de maillages SCOTCH [PR96]. A partir du maillage, un graphe est construit : les faces du maillage correspondent aux arêtes du graphe et les cellules du maillage correspondent à ses nœuds. Comme le coût opératoire par cellule n'est pas le même à cause du schéma d'intégration temporelle adaptatif, les nœuds du graphe sont pondérés en accord avec le niveau temporel des cellules auxquelles ils correspondent. Pour éviter l'apparition de cellules de bord avec un faible niveau temporel, les cellules contigües de niveau temporel inférieur à un niveau temporel fixé et les arêtes entre ces cellules sont regroupées. Elles sont représentées par un seul nœud dans le graphe avec un poids ajusté en conséquence.

Une fois la décomposition de domaine effectuée, chaque processus possède ses faces et ses cellules. Les faces de bord sont dupliquées : les deux domaines en relation possèdent une copie de la face de bord. Chaque cellule n'appartient qu'à un seul domaine. De chaque côté d'une face de bord, il y a une cellule locale et une cellule dite "fantôme". Une cellule fantôme est une cellule pour laquelle les valeurs ne sont pas calculées mais mises à jour par une communication. Les cellules fantômes ne sont pas considérées comme des cellules du domaine.

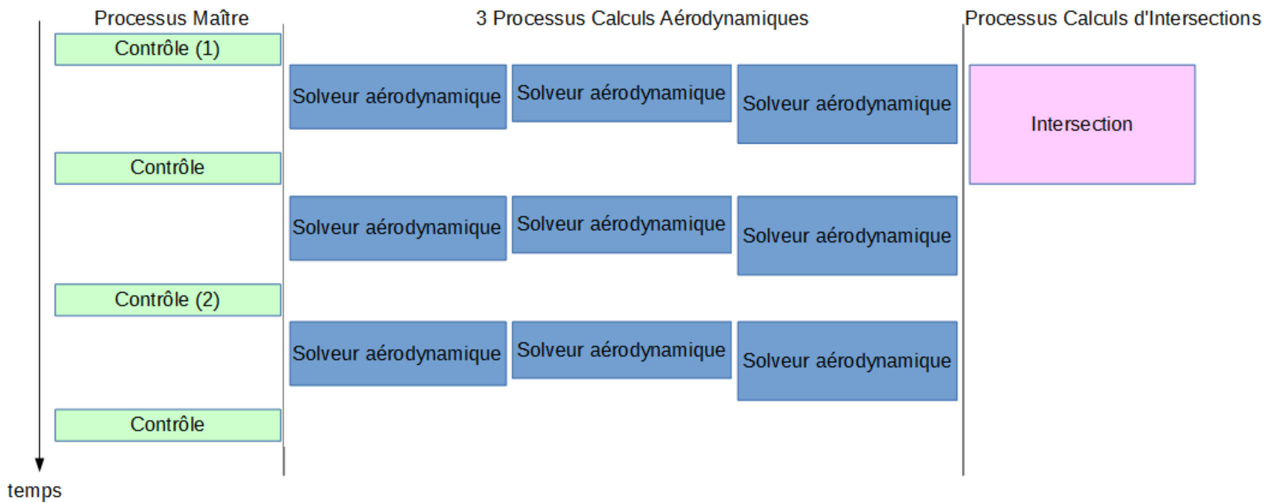
La Figure 2.6 illustre deux domaines, un vert et un rouge. Si on considère le domaine vert, les cellules de bord du domaine rouge seront des cellules fantômes pour ce domaine vert.

2.2.3.2 Évolution des index pour la version hybride MPI+OpenMP

Chaque processus de calcul dispose de ses propres index pour les faces et cellules qu'il possède.



(a) Version OpenMP



(b) Version MPMD

FIGURE 2.5 – Comparaison entre le fonctionnement de la version OpenMP et celui de la version MPMD.

Dans la version distribuée, on distingue les cellules de bord des cellules internes (situées à l'intérieur du domaine) afin de pouvoir les traiter en priorité. Pour cela, on rajoute de nouvelles bornes dans l'index des cellules. La Figure 2.7 illustre cette modification.

L'ajout de bornes à l'index des faces permet de connaître la connectivité entre les différentes cellules de bord des différents domaines de calcul. Les faces situées entre deux domaines de calcul

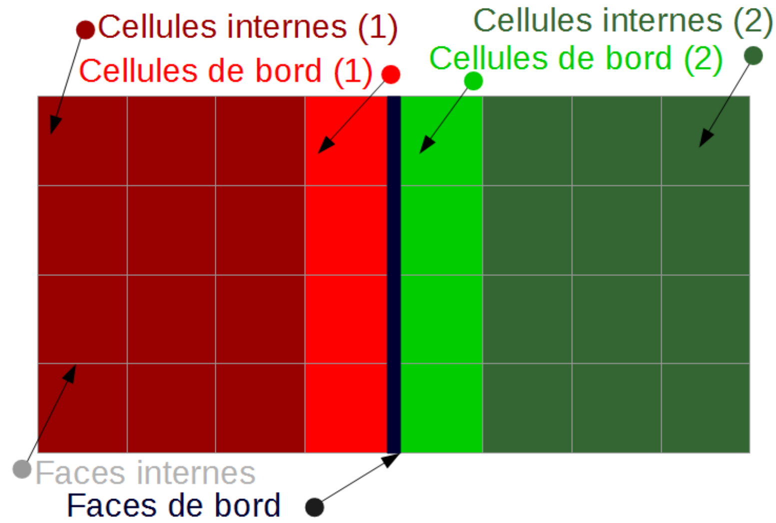


FIGURE 2.6 – Illustration des différentes composantes pour deux domaines de calcul.

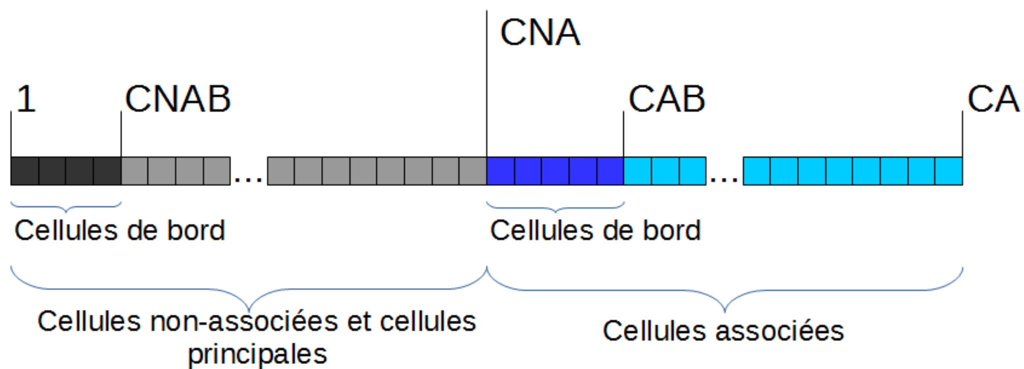


FIGURE 2.7 – Index des cellules avec prise en compte des cellules de bord.

sont forcément des faces entre cellules réelles. Pour les faces de bord, on rajoute donc des bornes pour chaque voisin. Ces bornes permettent de savoir avec quel domaine voisin la face est partagée et de quelle côté se situe la cellule locale.

Le tri effectué pour l'intégration temporelle adaptative tient compte de ces nouvelles bornes.

2.2.3.3 Communications

Concernant le solveur aérodynamique, et étant donné que les faces de bord sont dupliquées, les seules valeurs à échanger correspondent aux cellules de bord (en envoi) et aux cellules fantômes (en réception). Des types MPI permettant d'effectuer les communications sont créés à l'aide de l'index des faces. Ces types existent pour chaque niveau temporel et sont recréés à chaque itération. Concernant les communications à proprement dit, il faut remplir les tableaux de valeurs correspondant aux cellules fantômes avec une communication avant que ces valeurs ne soient nécessaires en lecture. Donc, lorsqu'on recalcule des valeurs pour les faces de bord, il faut s'assurer que les données correspondant aux cellules fantômes soient bien reçues. Pour maximiser

le recouvrement calcul-communication, les valeurs pour les cellules fantômes sont envoyées "au plus tôt", juste après avoir été calculées. Le pseudo-code de l'Algorithme 4 devient donc celui de l'Algorithme 5.

Algorithme 4 Pseudo-code sans communication

1: Calcul impliquant les cellules pour un niveau temporel τ
2: Calcul impliquant les faces pour un niveau temporel τ

Algorithme 5 Pseudo-code avec communications

1: Calcul impliquant les cellules de bord pour un niveau temporel τ
2: Lancement des communications asynchrones (envoi / réception, niveau temporel τ)
3: Calcul impliquant les cellules internes pour un niveau temporel τ
4: Calcul impliquant les faces internes pour un niveau temporel τ
5: Attente des communications asynchrones
6: Calcul impliquant les faces de bord pour un niveau temporel τ

2.2.3.4 Rééquilibrage asynchrone de la charge

Il est parfois nécessaire de rééquilibrer la charge, principalement parce que les niveaux temporels des mailles évoluent au cours du calcul. Ainsi, à chaque itération, le processus maître récupère le niveau temporel de chacune des mailles provenant des processus aérodynamiques. En effet, c'est le niveau temporel d'une maille qui détermine son coût de calcul. Si le déséquilibre est important, un nouvel équilibrage est calculé par le processus maître : un nouveau partitionnement de graphe est calculé, en ajustant les poids des nœuds au regard des informations reçues. On autorise un seul calcul de rééquilibrage à la fois.

Pour que ce calcul puisse se faire de manière asynchrone, on utilise une tâche OpenMP (!$OMP TASK) et des variables globales. **REEQ_EN_COURS** indique si un rééquilibrage est en cours, **REEQ_DISPO** indique que le dernier rééquilibrage calculé est prêt à être appliqué.

L'Algorithme 6 décrit la boucle du maître en montrant uniquement les opérations qui concernent le rééquilibrage avec les pragmas OpenMP nécessaires pour pouvoir effectuer ce rééquilibrage de manière asynchrone. Le fait d'utiliser une tâche OpenMP (lignes 7 à 10) permet de lancer le rééquilibrage de manière asynchrone. On garantit qu'il n'y a qu'un seul rééquilibrage à la fois via la variable **REEQ_EN_COURS**. Avant de créer la tâche de rééquilibrage, la variable est mise à 1 et une fois que le rééquilibrage sera appliqué, elle est remise à 0. Le fait d'utiliser une tâche OpenMP permet au rééquilibrage de durer a priori plusieurs itérations, sans interférer avec le contrôle que le maître doit exercer.

La variable **REEQ_DISPO** est mise à 1 avant la fin de la tâche OpenMP. Quand elle vaut 1, le processus maître peut appliquer le nouveau rééquilibrage et remettre les variables globales **REEQ_DISPO** et **REEQ_EN_COURS** à zéro.

2.2.4 Prise en compte des déplacements de maillage

Dans la version OpenMP originale, un calcul d'intersection est lancé dès que la loi de conservation géométrique qui se base sur un déplacement de volume entre les cellules n'est plus satisfaisante pour approximer ce déplacement. Quand c'est le cas, la cinématique actuelle est appliquée

Algorithme 6 Boucle de contrôle du maître avec les opérations concernant le rééquilibrage

1: *!$OMP PARALLEL*
2: *!$OMP SINGLE*
3: **tant que** VRAI **faire**
4: Collecte des métriques pour le rééquilibrage
5: **si** Déséquilibre > seuil autorisé ET **REEQ_EN_COURS**==0 **alors**
6: **REEQ_EN_COURS**=1
7: *!$OMP TASK*
8: Rééquilibrage
9: **REEQ_DISPO**=1
10: *!$OMP END TASK*
11: **fin si**
12: ...
13: **si REEQ_DISPO**==1 **alors**
14: Appliquer nouveau rééquilibrage
15: **REEQ_DISPO** = 0
16: **REEQ_EN_COURS** = 0
17: **fin si**
18: **si** Fin du calcul **alors**
19: Quitter la boucle
20: **fin si**
21: **fin tant que**
22: *!$OMP END SINGLE*
23: *!$OMP END PARALLEL*

Algorithme 7 Prise de décision concernant le calcul d'une nouvelle intersection

1: **si** Pas de calcul d'intersection en cours **alors**
2: **si** Critère 1 **alors**
3: Extrapolation de la cinématique
4: Envoi de la cinématique extrapolée au processus d'intersection
5: Création d'une tâche d'intersection
6: **fin si**
7: **fin si**
8: **si** Critère 2 **alors**
9: **si** Pas de calcul d'intersection en cours **alors**
10: Envoi de la cinématique au processus d'intersection
11: Création d'une tâche d'intersection
12: **fin si**
13: Application de l'intersection
14: **fin si**

aux différents corps et une nouvelle position est calculée pour chacun d'entre eux. S'en suit le calcul d'intersection.

Dans la version distribuée, on souhaite pouvoir faire les calculs d'intersection pendant les calculs aérodynamiques. On ne peut donc pas se contenter d'effectuer un calcul avec le déplacement actuel et l'appliquer plus tard : le déplacement de volume sera trop important pour que la loi de conservation géométrique soit respectée.

Pour palier à ce problème, on va calculer une extrapolation du déplacement pour un instant t, puis faire le calcul d'intersection pour cet instant et appliquer la nouvelle topologie uniquement quand le solveur aérodynamique aura atteint cet instant t. La prise de décision concernant le lancement du calcul d'intersection revient au processus maître qui enverra les données nécessaires au processus d'intersection. Le maître se chargera aussi de choisir le moment où appliquer la nouvelle topologie.

Il ne peut y avoir qu'un calcul d'intersection à la fois. Il y a deux critères pour savoir si un calcul d'intersection doit avoir lieu (critère 1 et critère 2) et un critère pour savoir si une intersection doit être appliquée. L'Algorithme 7 décrit le processus de décision concernant le calcul d'une nouvelle intersection. L'extrapolation est calculée pour un temps physique où la loi de conservation géométrique ne sera plus suffisante. Les critères 1 et 2 peuvent changer mais actuellement ils sont définis de cette manière :

— critère 1 : "dans 20 itérations, la loi de conservation géométrique ne sera plus respectée" ;
— critère 2 : "la loi de conservation géométrique n'est plus respectée OU le solveur a atteint un temps physique qui correspond à la cinématique extrapolée".

L'Algorithme 8 montre le mécanisme d'application d'une intersection : il permet notamment de ne pas appliquer une intersection de mauvaise qualité dans le cas où l'extrapolation de la cinématique serait éloignée de la vraie cinématique au moment de l'application.

Algorithme 8 Application de l'intersection

1: Réception des calculs d'intersection
2: **si** Cinématique extrapolée proche de la cinématique réelle **alors**
3: Garder la différence entre la cinématique extrapolée et la vrai cinématique pour le prochain déplacement
4: **sinon**
5: Envoi de la cinématique au processus d'intersection
6: Création d'une tâche d'intersection
7: Réception des calculs d'intersection
8: **fin si**
9: Prise en compte de l'intersection

Le déroulement des intersections asynchrones dans le cas où le calcul d'intersection n'est pas rejeté est illustré sur la Figure 2.8. L'asynchronisme provient du fait qu'on peut avoir plusieurs itérations du solveur aérodynamique pendant un calcul d'intersection. Le scénario de la figure décrit un cas où 3 itérations du solveur aérodynamique se déroulent, aucune tâche d'intersection n'est en cours au départ. Après le premier calcul de la cinématique, le processus maître vérifie s'il est judicieux de démarrer une tâche d'intersection. Dans le scénario qu'on a choisi, c'est le cas, le O signifiant OUI. Ce résultat implique le déclenchement des opérations liées à la création d'une tâche d'intersection. Le maître envoie donc les données au processus d'intersection qui effectuera ensuite le calcul d'intersection. Dès que ce calcul est terminé, le processus d'intersection poste les requêtes concernant l'envoi de la nouvelle topologie pour le processus maître. Au moment où

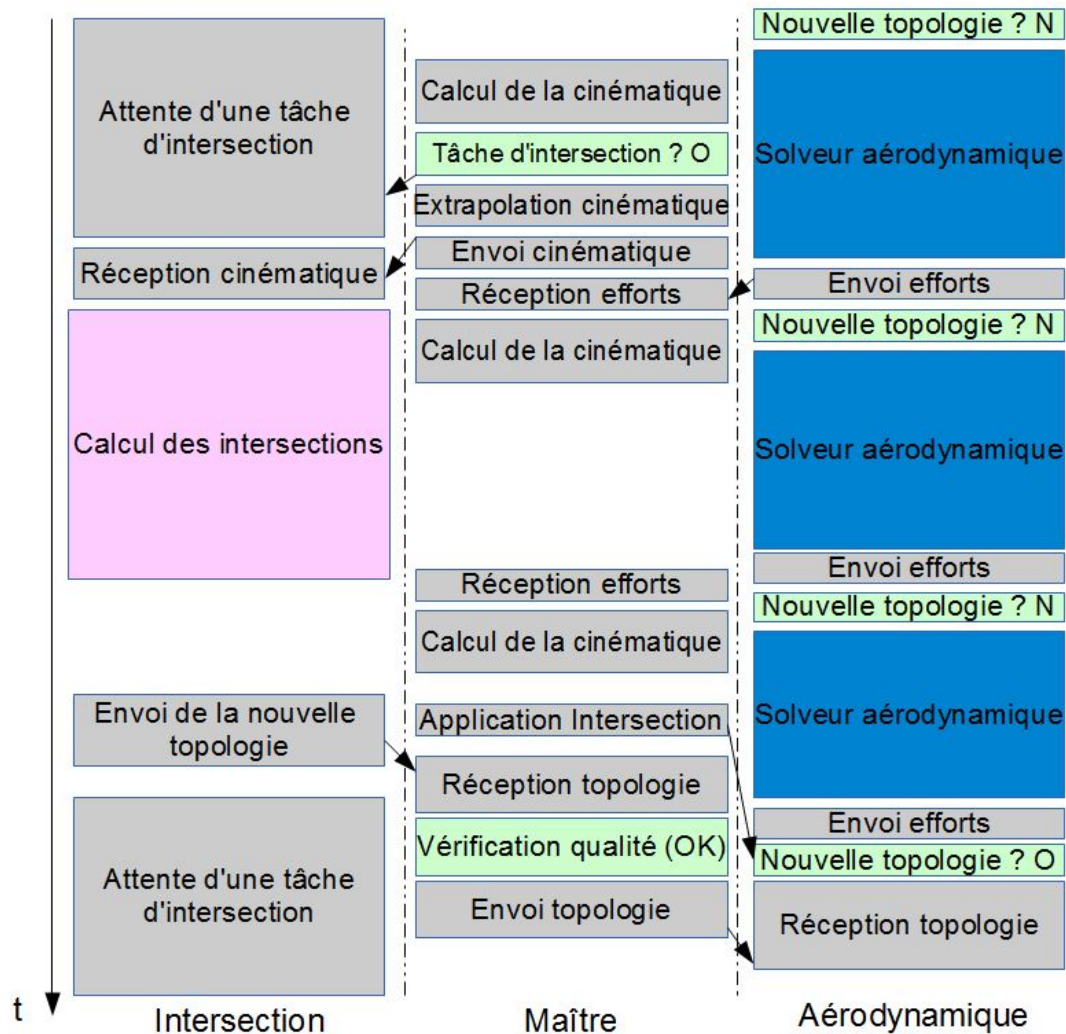


FIGURE 2.8 – Déroulement des intersections dans le cas d'une intersection non rejetée.

la nouvelle topologie est prête à être appliquée, le processus maître l'envoie aux processus dédiés aux calculs aérodynamiques.

Après l'application d'une intersection et avant l'envoi aux processus aérodynamiques, les cellules nouvellement découvertes (qui appartenaient donc à aucun domaine) se voient attribuer un domaine. Pour ne pas à avoir à ré-effectuer un partitionnement avec SCOTCH, on utilise un algorithme moins couteux. Pour chaque cellule nouvellement découverte, on regarde à quels domaines appartiennent ses cellules voisines et on lui attribue le domaine le plus présent parmi ses voisines.

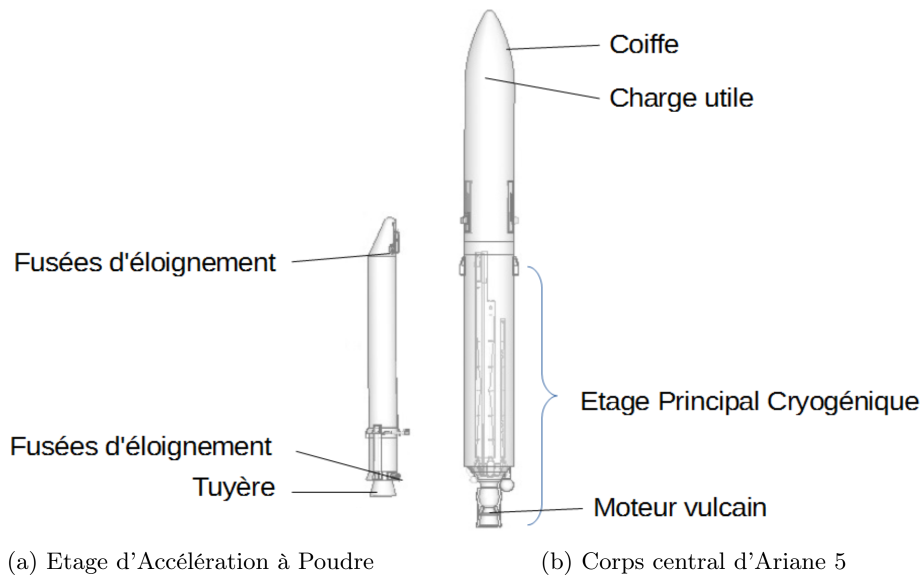
Calcul parallèle des intersections Il est possible d'utiliser plusieurs processus d'intersection pour accélérer le traitement du calcul des intersections. Lorsque plus d'un processus dédié au calcul d'intersection est présent, l'un d'entre eux a un rôle particulier. Il communique directement avec le processus maître et distribue le travail aux autres processus d'intersection.

En pratique, cette fonctionnalité n'est pas utilisée : les calculs aérodynamiques dominent largement les simulations actuelles et le fait de pouvoir effectuer les calculs d'intersection de

manière asynchrone suffit à les recouvrir.

2.3 Validation expérimentale et analyse de performance

La mission d'un lanceur est de mettre en orbite des satellites qui peuvent remplir différentes missions : réaliser des observations, assurer des communications ou servir à la navigation. Le confort de la charge utile et la garantie de son intégrité sont une condition essentielle de la réussite de la mission.



(a) Etage d'Accélération à Poudre (b) Corps central d'Ariane 5

FIGURE 2.9 – EAP et corps central d'Ariane 5.

La propulsion d'Ariane 5 est assurée par un étage principal (Etage Principal Cryogénique - EPC) et de deux propulseurs d'appoint (Etages d'Accélération à Poudre - EAP), comme illustré sur la Figure 2.9. Il y a différentes étapes critiques dans le déroulement d'une mission ; celles-ci sont décrites sur la Figure 2.10.

H0	Allumage du moteur Vulcain
H0 + 7s	Allumage des Etages d'Accélération à Poudre
H0 + 2min21s	Séparation des Etages d'Accélération à Poudre
H0 + 3min20s	Séparation de la coiffe
H0 + 8min55s	Séparation de l'étage principal
H0 + 8min59s	Allumage du moteur de l'étage supérieur
H0 + 25min	Extinction de l'étage supérieur
H0 + 25min	Séparation des satellites

FIGURE 2.10 – Déroulement d'un vol Ariane 5.

Les cas tests que nous allons présenter correspondent à deux de ces étapes. Le premier correspond à l'étape "H0 + 7s" avec l'étude de l'onde de souffle à l'allumage des EAP. C'est un calcul qui n'implique pas de déplacement de maillages. L'autre cas correspond à l'étape "H0 + 2min21s" avec la séparation des EAP.

2.3.1 Calcul d'une onde de souffle

Le confort de la charge utile au décollage est un facteur important de la réussite de la mission pour un lanceur. Il est essentiel d'avoir une idée du niveau de sollicitation de la charge utile au décollage afin de s'assurer qu'elle ne subisse pas de dégât. La simulation numérique est un élément essentiel pour estimer cette sollicitation. Pour une fusée de type Ariane 5, le moteur principal n'est pas suffisant pour permettre le décollage, mais il est démarré 7 secondes avant les deux moteurs des EAP. 90% de la poussée au décollage est fournie par les EAP.

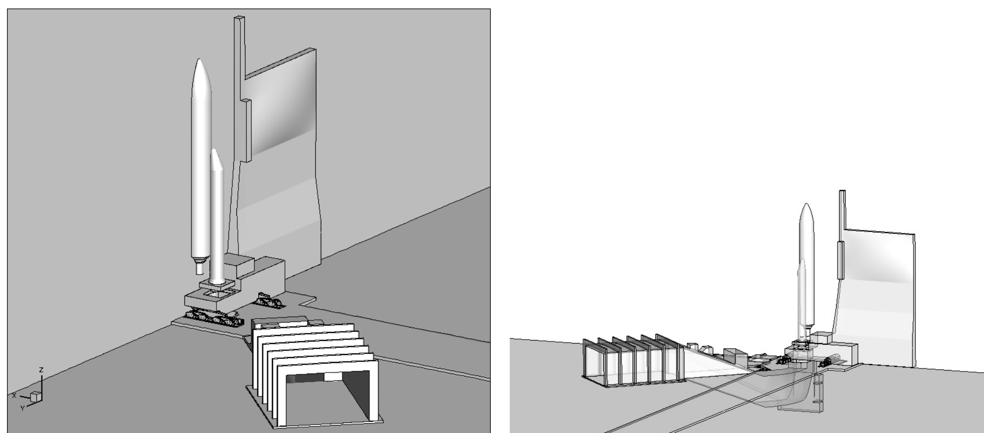
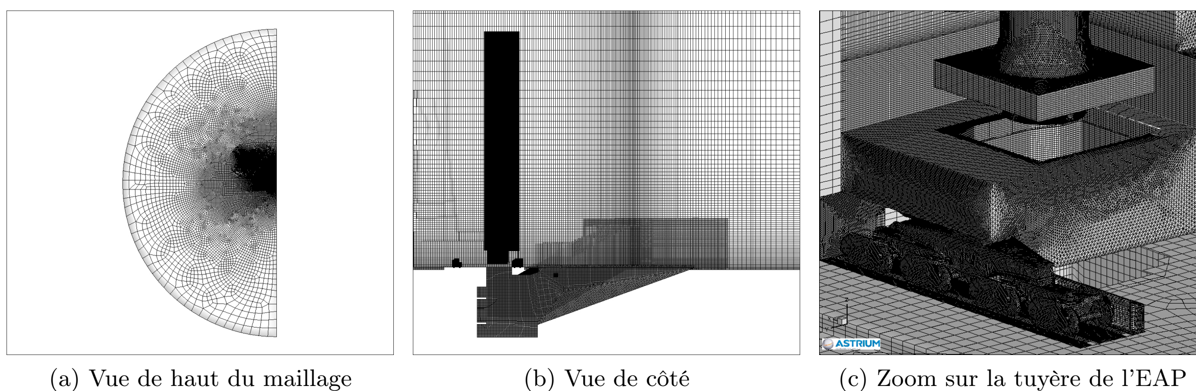


FIGURE 2.11 – Géométrie utilisée pour le calcul.



(a) Vue de haut du maillage (b) Vue de côté (c) Zoom sur la tuyère de l'EAP

FIGURE 2.12 – Maillage utilisé pour le calcul (11M de cellules, 31.5M de faces).

Ce cas est l'occasion de faire une analyse du schéma d'intégration temporelle adaptatif sur un cas industriel et de montrer son intérêt. Dans ce calcul, on simule l'allumage de l'EAP et on étudie l'évolution de l'onde de souffle. La géométrie utilisée pour ce calcul est présentée sur la Figure 2.11, il contient 11M de cellules. Le pas de tir est considéré comme symétrique, donc seule une moitié est modélisée et l'autre moitié est reconstruite par symétrie. Une condition limite de symétrie est donc utilisée pour la simulation. On peut voir un carneau sur la géométrie ; il s'agit d'une tranchée servant à canaliser les jets de gaz brûlés.

Le maillage utilisé pour le calcul est présenté sur la Figure 2.12. Bien qu'il s'agisse d'un calcul sans déplacement, les intersections de maillages sont utilisées pour simplifier la conception du maillage final comme montré sur la Figure 2.12c. La vue de haut (Figure 2.12a) montre bien la

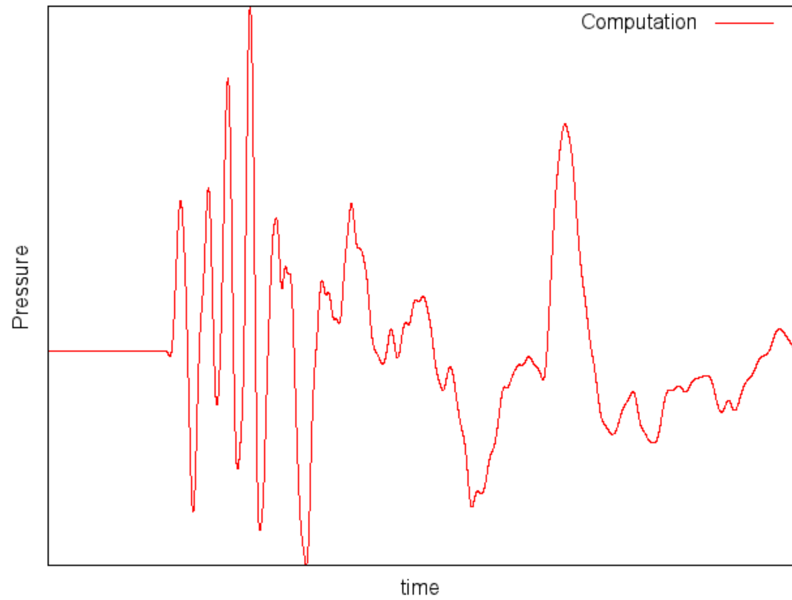


Figure 2.13 – Pression au cours du temps pour un capteur numérique.

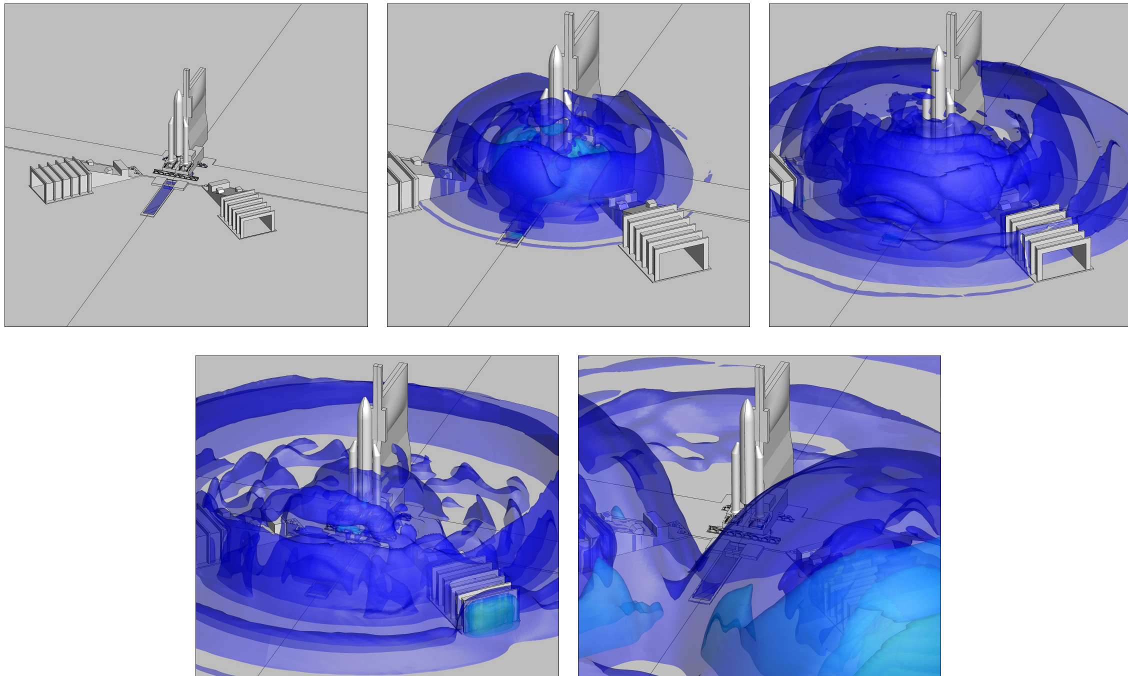


Figure 2.14 – Illustration du calcul d'onde de souffle.

différence de densité en mailles en fonction de la position dans le maillage : les mailles sont plus fines autour du lanceur et encore plus fines autour de la tuyère.

La Figure 2.13 présente un capteur numérique (une cellule du maillage pour laquelle on va faire un relevé de certaines valeurs physiques au cours du temps). Deux surpressions sont notables et bien capturées par le calcul. La première est due au décollage et est appelée *Ignition Over*

Pressure (IOP). La seconde provient du retour de l'onde après son passage dans le carneau et est appelée *Duct Over Pressure* (DOP).

La Figure 2.14 illustre le déplacement de l'onde de souffle. Il s'agit de la géométrie utilisée, mais on reconstruit l'ensemble du pas de tir par symétrie. A partir de la quatrième image, on peut voir le retour de l'onde de souffle.

2.3.1.1 Impact du schéma d'intégration temporelle adaptatif

Dans cette section, on étudie en mémoire partagée le comportement de l'algorithme d'intégration temporelle adaptatif. Le même calcul global est repris en faisant varier le niveau temporel maximum θ et les différents résultats sont comparés à un temps physique proche. Les notations introduites dans la section 2.1.2.1 sont utilisées. Cependant, on compte le nombre de faces plutôt que le nombre de cellules, les opérations sur les faces étant au final plus chères que les opérations sur les cellules. On a donc un coût relatif au nombre d'opérations sur les faces.

La Figure 2.15 dénombre le nombre de faces présentes pour chaque classe temporelle. Elle illustre aussi les coûts associés en utilisant soit l'intégration temporelle adaptative, soit un pas de temps global. On compare le coût en équivalent "opérations sur les faces" entre une intégration temporelle adaptative et une intégration temporelle avec un pas de temps global pour atteindre le même pas de temps physique.

La colonne $\sum_{\tau=0}^{\theta} C(\tau)$ indique le coût d'une itération du solveur ; il est obtenu par l'utilisation de la formule $C(\tau) = 2^{\theta-\tau} * |\Omega(\tau)|$. La colonne $2^{\theta} * |\Omega|$ correspond au coût pour atteindre le même pas de temps physique mais avec un pas de temps global. On constate que sur ce cas test, la répartition des mailles est compatible avec l'utilisation du schéma d'intégration temporelle adaptatif : avec $\theta = 5$, on a un rapport de 16.7 par rapport à l'utilisation d'un pas de temps global. Pour ce cas, il y a 32 sous-itérations au cours d'une itération du solveur. D'après cette estimation, ces 32 sous-itérations sont moins coûteuses que 2 itérations avec un pas de temps global.

θ	$\lvert\Omega(\tau_0)\rvert$	$\lvert\Omega(\tau_1)\rvert$	$\lvert\Omega(\tau_2)\rvert$	$\lvert\Omega(\tau_3)\rvert$	$\lvert\Omega(\tau_4)\rvert$	$\lvert\Omega(\tau_5)\rvert$	$\sum_{\tau=0}^{\theta} C(\tau)$	$2^{\theta} * \lvert\Omega\rvert$
0	31,5M	0	0	0	0	0	31,5M	31,5M
1	17k	31,5M	0	0	0	0	31,5M	63M
2	13k	630k	31M	0	0	0	32M	126M
3	14k	662k	977k	30M	0	0	34,5M	252M
4	17k	763k	932k	1,8M	28M	0	42M	505M
5	16k	763k	933k	1,8M	4,4M	23,5M	60M	1G

FIGURE 2.15 – Répartition des mailles dans les différents niveaux temporels et coût comparatif avec un pas de temps global.

La Figure 2.16 montre comment se répartissent les mailles dans les différents niveaux temporels lorsque l'on fait varier θ. Les groupes de barres se lisent deux par deux : la première barre de chaque couple représente la proportion des mailles dans les niveaux temporels, la seconde barre représente la proportion théorique de calculs associés. Jusqu'à $\theta = 5$, les mailles du niveau θ lui-même considéré représentent la majorité des mailles présentes dans le calcul (au moins 75%). Jusqu'à $\theta = 4$, elles représentent aussi une part de calcul qui est supérieure à celui de l'ensemble des autres niveaux.

On constate que la proportion de mailles de niveau τ_0 est faible quel que soit $\theta > 0$; la proportion de calcul associée reste marginale comparée aux autres classes.

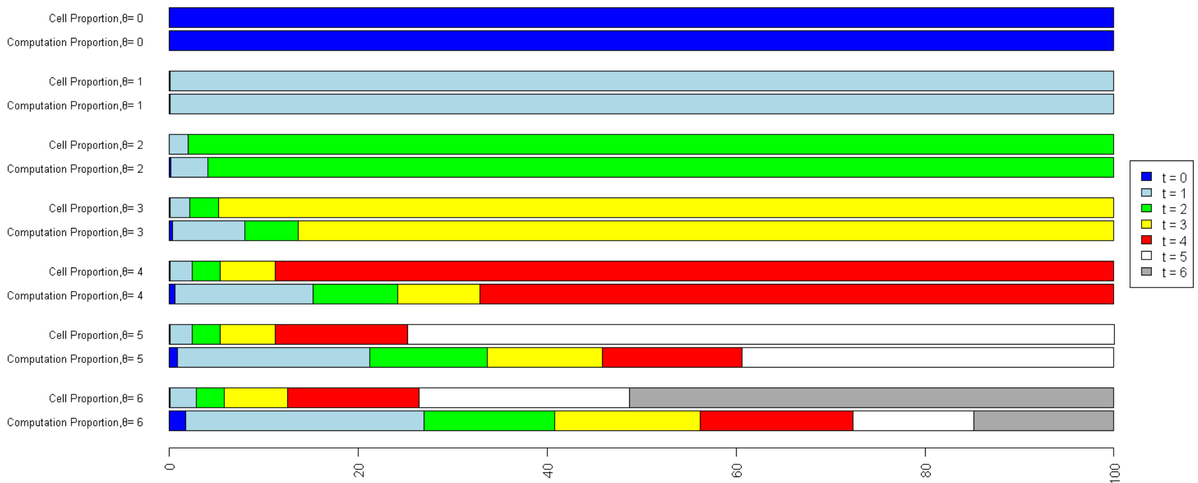


FIGURE 2.16 – Proportion de mailles et proportion de calculs associés, avec θ variant de 0 à 6.

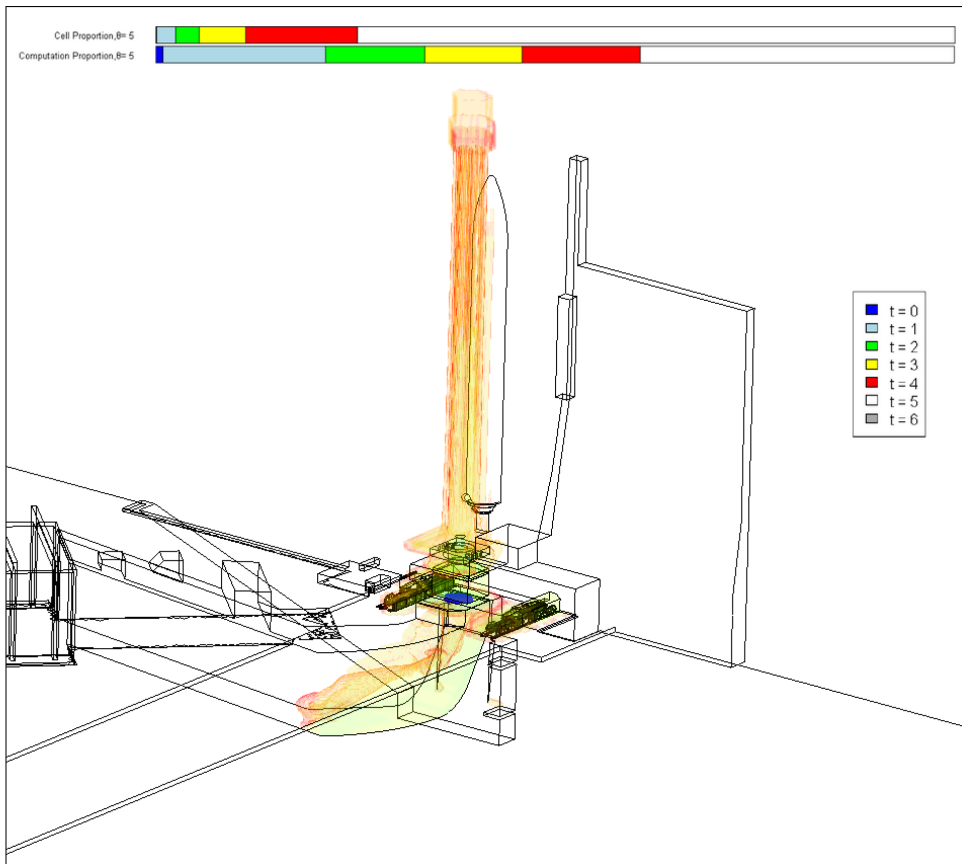


FIGURE 2.17 – Emplacement des mailles selon leur niveau temporel ($\theta = 5$).

Pour $\theta > 2$, la seconde barre montre que les mailles de niveau τ_1 sont celles pour lesquelles la proportion de calcul augmente le plus significativement. Ainsi, $|\Omega(\tau_1)|$ est du même ordre que $|\Omega(\tau_2)|$ d'après la première barre, mais on itère deux fois plus sur les mailles de niveau τ_1. A θ fixé > 2, et si on excepte les mailles du niveau θ, les mailles de niveau τ_1 sont celles qui représentent la plus grande proportion de calcul.

Un autre point important concerne la localisation géométrique des mailles : les dépendances de calcul inter-mailles étant locales, l'emplacement des mailles de faibles niveaux temporels à une incidence importante sur les synchronisations. En mémoire distribuée, on souhaitera absolument éviter en particulier l'existence de frontière entre mailles de niveaux τ_0 : celles-ci sont intégrées à toutes les sous-itérations et impliquent donc un nombre important de synchronisations. La Figure 2.17 montre l'emplacement des mailles en fonction de leur niveau temporel. Les mailles de niveau temporel θ (τ_5 dans ce cas) ne sont pas coloriées. On constate que les mailles de niveaux inférieurs à 5 sont situées autour de l'EAP. Les mailles de très faibles niveaux (τ_0 et τ_1) sont situées en sortie de tuyère. Les différents niveaux temporels s'articulent un peu comme des "pelures d'ognons", les niveaux les plus faibles étant "les pelures les plus intérieures". On constate qu'il y a une corrélation entre la densité des mailles (visible sur la Figure 2.12) et les niveaux temporels ; cela était prévisible vue l'importance du volume des mailles dans le calcul de la condition CFL.

Dans ce cas de calcul, les mailles de niveaux $\tau < 5$ sont toutes regroupées au même endroit. Les proportions de nombres de mailles et de calcul sont données sur la partie haute de la Figure 2.17. Bien que situées au même endroit, il reste possible de faire en sorte que les mailles de niveau τ_0 soient situées dans le même domaine : elles représentent suffisamment peu de calcul pour ne pas remettre en cause l'équilibre global de calcul. Par contre, ce n'est pas possible pour les mailles de niveau τ_1 car la proportion de calcul associée est trop importante et ces mailles sont situées au même endroit.

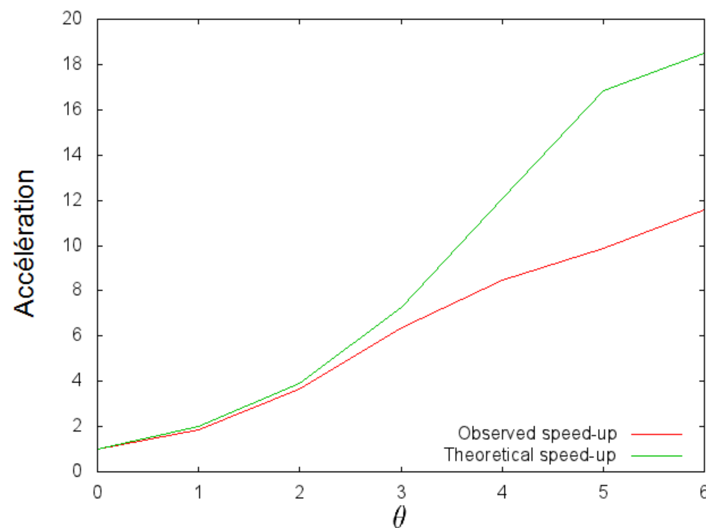


FIGURE 2.18 – Accélérations attendue et mesurée pour l'utilisation du schéma d'intégration temporelle adaptatif.

Analyse de la performance effective liée à l'utilisation du schéma d'intégration temporelle adaptatif

La Figure 2.18 montre l'accélération théorique attendue suite à l'utilisation du schéma d'intégration temporelle adaptatif comparée à l'utilisation d'un pas de temps global, puis l'accélération réellement obtenue en utilisant un nœud de calcul [3] pour le cas test de l'onde de souffle. Pour

3. La partition standard du calculateur *AIRAIN* du CCRT a été utilisée. Les nœuds sont composés de 2×8 cores SandyBridge@2.7Ghz, 64 GB de RAM. Le compilateur *ifort 15* a été utilisé pour toutes les expériences. Les

effectuer cette comparaison, on mesure le temps écoulé nécessaire pour atteindre un temps physique donné. Si jusqu'à $\theta = 3$, le gain obtenu est proche du gain attendu, il est moins important pour des niveaux temporels plus grands : pour $\theta = 5$, on a 9.8 au lieu des 16.7 espérés (ratio 1.72) et pour $\theta = 6$, on a 11.8 au lieu de 18.2 (ratio 1.54). Cependant ce gain est en fait très intéressant car il est obtenu sans utiliser de ressources supplémentaires.

Pour analyser plus finement ce gain et vérifier la qualité de notre estimation du coût initial, on mesure le temps passé (en secondes) par maille selon leur niveau temporel. Le code est instrumenté de manière à pouvoir obtenir le temps passé dans chaque sous-itération. Plusieurs niveaux temporels étant traités à chaque sous-itération, on redistribue le temps passé dans chaque niveau en fonction de la proportion de mailles traitées. Cela permet d'obtenir une estimation du temps passé par niveau temporel. La Figure 2.19 montre le détail de ces mesures pour $\theta = 4$ fixé et la répartition obtenue. Un seul cœur a été utilisé. On constate que les proportions théoriques de calcul et celles mesurées sont proches pour chaque τ. Globalement, ces proportions sont légèrement sous-estimées pour $\tau < \theta$ et surestimées pour $\tau = \theta$.

	τ_0	τ_1	τ_2	τ_3	τ_4	total
Nombres de Faces	5606	631991	961495	158412	28424335	31607939
Proportion théorique calcul	0.22%	12.46%	9.48%	7.81%	70.03%	100%
Sous-itération 1 (s)	0.03	3.38	5.14	8.48	152	169
Sous-itération 2 (s)	0.052	0	0	0	0	0.052
Sous-itération 3 (s)	0.0327	3.69	0	0	0	3.72
Sous-itération 4 (s)	0.0523	0	0	0	0	0.0523
Sous-itération 5 (s)	0.0347	3.91	5.95	0	0	9.9
Sous-itération 6 (s)	0.0524	0	0	0	0	0.0524
Sous-itération 7 (s)	0.0325	3.67	0	0	0	3.7
Sous-itération 8 (s)	0.0523	0	0	0	0	0.0523
Sous-itération 9 (s)	0.0387	4.36	6.64	10.9	0	22
Sous-itération 10 (s)	0.052	0	0	0	0	0.052
Sous-itération 11 (s)	0.0325	3.67	0	0	0	3.7
Sous-itération 12 (s)	0.0522	0	0	0	0	0.0522
Sous-itération 13 (s)	0.0347	3.91	5.96	0	0	9.91
Sous-itération 14 (s)	0.0523	0	0	0	0	0.0523
Sous-itération 15 (s)	0.0327	3.69	0	0	0	3.72
Sous-itération 16 (s)	0.0522	0	0	0	0	0.0522
Temps cumulé (s)	0.686	30.3	23.7	19.4	152	226
Proportion mesurée	0.30%	13.39%	10.48%	8.59%	67.24%	100%

FIGURE 2.19 – Détails d'une itération et des coûts constatés par niveau temporel ($\theta = 4$).

La Figure 2.20 reprend le même calcul pour θ variant maintenant de 0 à 6. Cette fois-ci, on observe la qualité de l'estimation pour un nombre de CPUs (colonne #C du tableau) qui varie et on veut vérifier si l'utilisation d'OpenMP en mémoire partagée a un impact concernant le temps passé pour chaque classe de maille.

Sur le tableau on voit que pour les mailles de niveau τ_0, la proportion mesurée est toujours beaucoup plus importante que la proportion théorique. Cependant, on reste sur des proportions de calcul très faibles, au pire de l'ordre de 2% du total dans le cas $\theta = 6$ pour 16 CPUs. La

expériences impliquant MPI ont été faites avec *BullXMPI*.

	θ	#C	τ_0	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6
Prop. th.	1	-	0.04 %	99.96 %					
Prop. mes.	1	1	0.05 %	99.95 %					
Prop. mes.	1	8	0.15 %	99.85 %					
Prop. mes.	1	16	0.21 %	99.79 %					
Prop. th.	2	-	0.07 %	3.91 %	96.02 %				
Prop. mes.	2	1	0.10 %	4.14 %	95.76 %				
Prop. mes.	2	8	0.31 %	4.79 %	94.91 %				
Prop. mes.	2	16	0.47 %	4.89 %	94.65 %				
Prop. th.	3	-	0.13 %	7.32 %	5.57 %	86.98 %			
Prop. mes.	3	1	0.18 %	7.98 %	5.99 %	85.84 %			
Prop. mes.	3	8	0.55 %	9.43 %	6.56 %	83.46 %			
Prop. mes.	3	16	0.82 %	9.67 %	6.72 %	82.79 %			
Prop. th.	4	-	0.22 %	12.46 %	9.48 %	7.81 %	70.04 %		
Prop. mes.	4	1	0.30 %	13.43 %	10.47 %	8.58 %	67.22 %		
Prop. mes.	4	8	0.84 %	15.18 %	11.36 %	9.06 %	63.56 %		
Prop. mes.	4	16	1.22 %	15.42 %	11.53 %	9.09 %	62.74 %		
Prop. th.	5	-	0.32 %	17.75 %	13.53 %	11.13 %	14.72 %	42.56 %	
Prop. mes.	5	1	0.34 %	17.78 %	13.99 %	11.76 %	15.04 %	41.08 %	
Prop. mes.	5	8	1.08 %	18.27 %	14.34 %	12.07 %	14.94 %	39.30 %	
Prop. mes.	5	16	1.67 %	18.39 %	14.36 %	12.03 %	14.93 %	38.62 %	
Prop. th.	6	-	0.37 %	20.88 %	15.95 %	13.06 %	17.32 %	14.84 %	17.57 %
Prop. mes.	6	1	0.39 %	20.55 %	16.28 %	13.71 %	17.71 %	14.60 %	16.77 %
Prop. mes.	6	8	1.22 %	20.85 %	16.57 %	13.94 %	17.51 %	14.04 %	15.86 %
Prop. mes.	6	16	1.86 %	20.90 %	16.46 %	13.84 %	17.48 %	13.90 %	15.57 %

FIGURE 2.20 – Comparaison entre la proportion de calcul théorique (*Prop. th.*) et la proportion de calcul mesurée (*Prop. mes.*).

proportion effective de calcul pour les mailles de niveau $\tau = \theta$ est toujours plus faible que l'estimation théorique. Pour $\tau < \theta$, on a toujours une sous-estimation et celle-ci a tendance à se dégrader en utilisant plusieurs CPUs.

La capacité à estimer correctement le coût des mailles est importante pour la version distribuée. Dans cette version, on distribuera des portions du maillage sur différents processus et il faudra une bonne fonction de coût pour le partitionneur afin d'avoir un bon équilibre de la charge. L'étude faite ci-dessus montre que notre estimation théorique du coût de calcul des mailles sera une bonne base pour la fonction de coût du partitionneur.

2.3.1.2 Mise en œuvre et analyse d'une version OpenMP

Dans l'optique de faire une version hybride MPI+OpenMP, il est important de savoir comment se comporte la version OpenMP. En effet, plusieurs stratégies sont possibles lorsqu'il s'agit d'utiliser les ressources de calcul avec une version distribuée, et au sein d'un nœud de calcul, il est possible de mettre un ou plusieurs processus utilisant la mémoire partagée. Il est donc important de savoir comment se comporte l'implémentation OpenMP en mémoire partagée.

La Figure 2.21 montre comment le code se comporte en mémoire partagée pour θ variant de

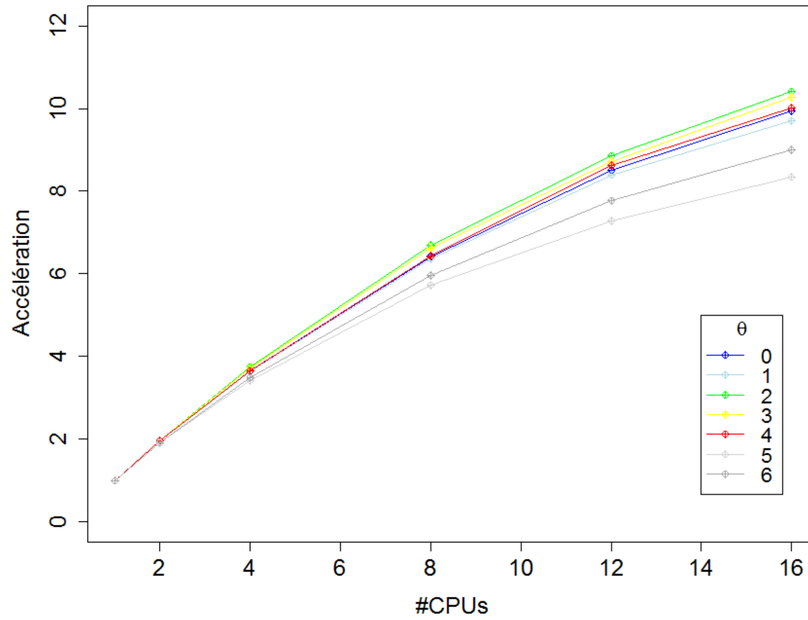


FIGURE 2.21 – Accélération de la version OpenMP (une courbe par valeur de θ entre 0 et 6).

0 à 6 sur un nœud et en faisant varier le nombre de CPUs de 1 à 16. Pour $\theta <= 4$, les courbes sont très proches. On constate un décrochement pour θ =5 et 6. Pour 8 CPUs, ce qui correspond à une socket sur la machine considérée, on obtient des accélérations autour de 6. Si on utilise l'ensemble du nœud, les accélérations varient entre 8 et 10.

θ	#CPUs	Temps (s)	Acc.	Eff.
2	1	175.016	1.0	1.0
2	2	89.174	1.96	0.98
2	4	46.784	3.74	0.94
2	8	26.134	6.69	0.84
2	12	19.779	8.84	0.74
2	16	16.813	10.40	0.65

(a) $\theta = 2$

θ	#CPUs	Temps (s)	Acc.	Eff.
5	1	349.716	1.0	1.0
5	2	183.432	1.90	0.95
5	4	102.357	3.41	0.85
5	8	61.096	5.72	0.71
5	12	48.033	7.28	0.60
5	16	41.914	8.34	0.52

(b) $\theta = 5$

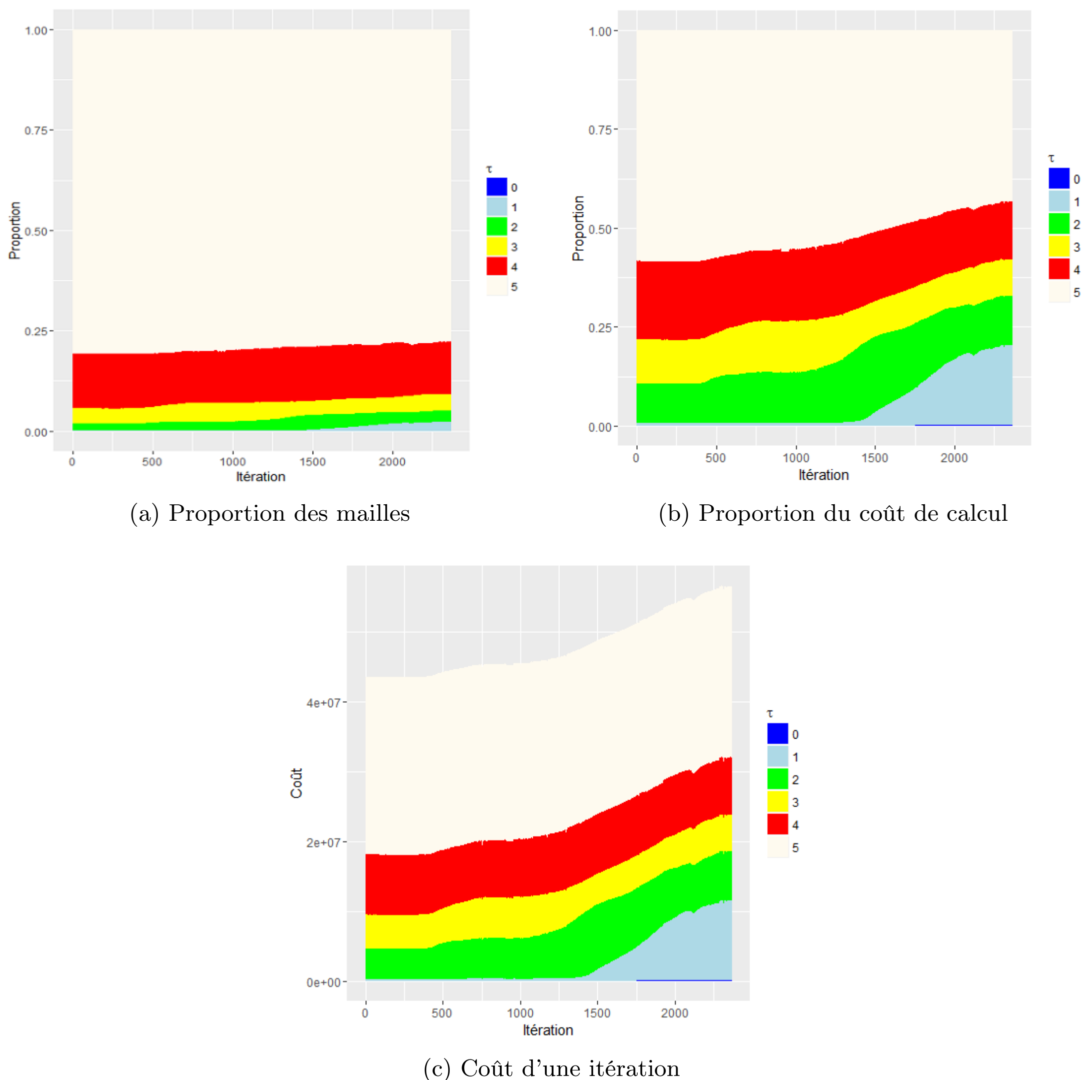
FIGURE 2.22 – Détails des performances (temps en secondes, accélération, efficacité) pour les deux cas θ =2 ou 5.

La Figure 2.22 détaille les performances pour une itération du solveur dans le cas $\theta = 2$ et 5 en faisant varier le nombre de CPUs utilisés par OpenMP sur un noeud de calcul. Le point positif est l'augmentation permanente de l'accélération avec le nombre de CPUs pour les cas considérés ($\theta = 2$ ou 5). Par contre, à θ fixé et donc à problème constant, l'efficacité décroit avec le nombre de CPUs, ce qui est habituel, mais de manière lente. Cela peut aussi mettre en évidence la limite d'une parallélisation strictement OpenMP et plaider en faveur d'avoir plusieurs processus OpenMP par nœud. Un autre point à noter concerne le fait que la parallélisation OpenMP est

moins efficace quand θ augmente. Dans ce cas, une itération se compose de beaucoup plus de sous-itérations. Les boucles qui traitent les faibles niveaux temporels concernent très peu d'éléments et sont très nombreuses. Comme on l'avait vu sur la Figure 2.1 pour $\theta = 3$, une itération sur deux concerne uniquement les mailles de niveau temporel τ_0, or on a vu avec les Figures 2.19 et 2.20 que les mailles de niveau τ_0 représentent une part négligeable du calcul.

Analyse de l'évolution de la charge de calcul

Un autre aspect qu'il ne faut pas négliger concernant le schéma d'intégration temporelle adaptatif est le fait que le niveau temporel des mailles évolue en cours de calcul. Le calcul du pas de temps pour une cellule ne fait pas intervenir uniquement le volume, mais aussi d'autres critères physiques, et c'est pour cette raison que les mailles peuvent changer de niveau temporel.



(a) Proportion des mailles

(b) Proportion du coût de calcul

(c) Coût d'une itération

FIGURE 2.23 – Evolution dans les classes temporelles au cours du calcul.

La Figure 2.23 illustre cette évolution pour les 2300 premières itérations, avec un θ fixé à 5. Le coût est toujours défini par $C(\tau)$ ($C(\tau) = 2^{\theta-\tau} * |\Omega(\tau)|$, cf section 2.1.2.1). Le nombre de mailles reste constant au cours du temps vu qu'il n'y a pas de déplacement de maillage. La

sous-figure 2.23a montre l'évolution de la proportion des mailles par niveau temporel en cours de calcul. L'évolution semble limitée, il y a de légères différences, mais globalement les mailles de niveau τ_5 représentent toujours la part la plus importante. On constate par contre une réelle différence avec les mailles de niveau τ_1 : invisibles au début de calcul, leur proportion devient visible aux alentours de la 1600ème itération. Par contre à aucun moment, la proportion des mailles de niveau τ_0 n'est visible.

La sous-figure 2.23b montre elle l'évolution de la proportion des coûts de calcul de chaque niveau temporel. On constate qu'en fait la proportion des mailles de niveau τ_1 évolue plus tôt que sur la précédente figure. Pour ce niveau temporel τ_1, une faible évolution de la proportion a une répercussion importante sur le coût. Anecdotique au départ, la proportion de calcul représentée par les mailles de niveau temporel τ_1 devient très importante : elle se classe deuxième derrière les mailles de niveau τ_5. Il faut noter qu'il y a 77% de mailles de niveau τ_5 pour seulement 2.27% de mailles de niveau τ_1. On voit aussi apparaître très légèrement les mailles de niveau τ_0 bien qu'elles soient très peu nombreuses (0.01% du total).

La dernière sous-figure montre l'évolution du coût de calcul d'une itération : avec l'évolution de la proportion des mailles des faibles niveaux, les itérations coûtent de plus en plus cher. Le coût des différentes classes évolue peu à l'exception notable des mailles de niveau τ_1. La 2300ème et dernière itération coûte près de 30% plus cher que la première.

Cette évolution importante du coût en cours de calcul et de la proportion des mailles dans les niveaux temporels constitue un point critique et montre bien la nécessité d'un équilibrage de charge en cours du calcul pour la version finale.

2.3.1.3 Mise en œuvre et analyse d'une version MPI+OpenMP

La première étape va consister à distribuer le travail entre les différents processus. Ces derniers n'ayant pas une vision globale commune, il est nécessaire qu'ils disposent de leur sous-domaine de travail propre et des informations nécessaires concernant les domaines voisins. On utilise donc un découpage spatial pour ce premier niveau de parallélisme.

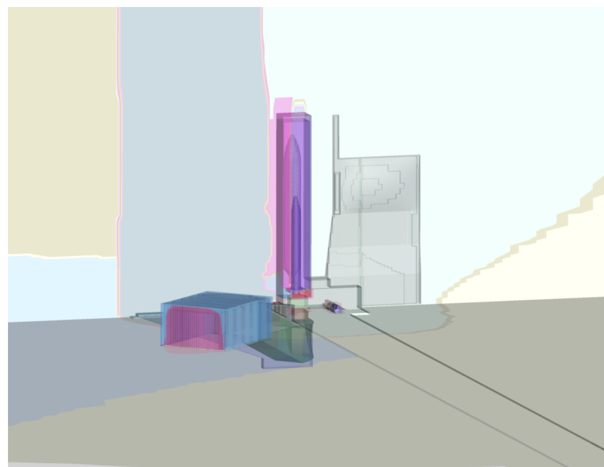


FIGURE 2.24 – Projection des différents domaines sur la géométrie.

Une illustration de la décomposition de domaine est montrée sur la Figure 2.24. Le maillage est traduit sous forme de graphe, puis partitionné par le logiciel SCOTCH. On peut constater

que les domaines ne semblent pas de même taille.

D'une part, les mailles n'ont pas le même volume initialement (cf Figure 2.12), et d'autre part, on ne donne pas la même importance aux cellules du maillage en fonction de leur niveau temporel. Dans la section 2.1.2.1, on a estimé de manière théorique et évalué de manière pratique le coût des différents niveaux temporels. On réutilise donc cette estimation pour donner un poids différent à chaque nœud du graphe. Ainsi, avec SCOTCH on partitionne un graphe pondéré et on souhaite équilibrer le poids des différentes partitions.

Désormais, on évalue le comportement de la version hybride MPI+OpenMP ; les résultats sont présentés sur les Figures 2.25 et 2.26.

Le calcul est effectué sur des nœuds disposant comme auparavant de deux sockets de 8 cœurs. On va placer un processus par nœud ou un processus par socket. Malheureusement, il n'a pas été possible de placer plus de deux processus par nœud : les mailles ayant un poids différent, il peut arriver qu'un processus dispose d'un trop grand nombre de mailles pour sa capacité mémoire ce qui ne permet pas de mener le calcul au bout [4]. Ce cas de figure peut aussi arriver avec une configuration avec deux processus par nœud et c'est pour cela que certaines configurations n'apparaissent pas sur la Figure 2.26.

Il s'agit à chaque fois du même calcul ; les temps physiques des mailles considérées sont donc les mêmes. *#ppn* indique le nombre de processus par nœud : il vaut 1 sur la Figure 2.25 et 2 sur la Figure 2.26. La colonne θ correspond au niveau temporel maximum autorisé pour le calcul considéré : θ varie de 0 à 6. $\#N$ indique le nombre de nœuds utilisés : il varie de 1 à 16. Le nombre de sous-domaines et de processus correspond donc à $\#ppn * \#N$. On donne ensuite dans les colonnes suivantes les informations relatives à l'itération générale : *Elaps* correspond au temps écoulé pour une itération générale ; *Acc.* correspond à l'accélération par rapport à la configuration avec un processus par nœud et ceci est valable aussi pour les configurations avec $\#ppn = 2$. L'efficacité est donnée dans la colonne *Eff.* et utilise le même calcul de référence. La colonne *Comm.* donne la proportion en temps passée dans les communications : il s'agit du ratio entre le temps passé dans les communications et le temps total. On donne enfin les informations relatives à l'intégration temporelle adaptative. Le pas de temps d'une itération générale varie de 1.45 μs (pour $\theta = 0$) à 93.93 μs (pour $\theta = 6$) ; il est multiplié par 2 à chaque fois que θ augmente. La colonne *h/ss* correspond au nombre d'heures de calcul qu'il faut pour simuler une seconde physique du phénomène avec la configuration donnée. Dans le meilleur cas présenté ($\theta = 6$, 16 nœuds), il faut moins de deux jours pour simuler une seconde physique. La dernière colonne *Acc_Tg* correspond à l'accélération globale obtenue comparée à celle qui serait obtenue par un calcul sur un seul nœud et avec un pas de temps global ($\theta = 0$). Les résultats présentés dans ces tableaux sont ensuite repris sur les Figures 2.27 et 2.28 afin d'être analysés de manière plus synthétique.

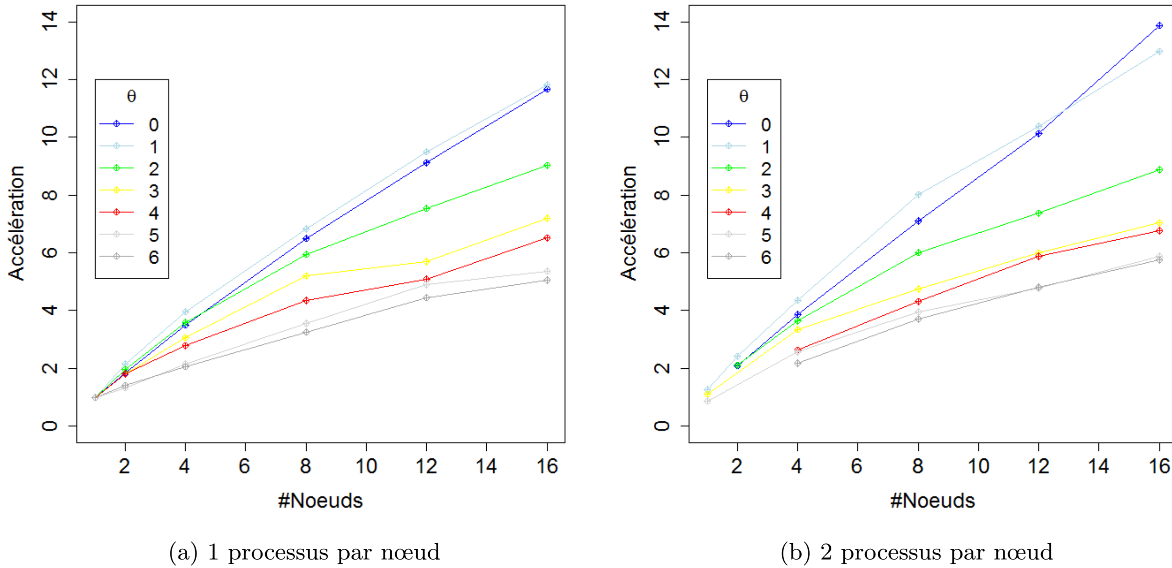
4. Il n'était pas possible de swapper avec le cluster utilisé, la taille du swap prédéfinie étant nulle.

θ	#N	Elaps	Acc.	Comm.	Eff.	h/ss	Acc_Tg
0	1	18.0	1	0 %	1	3448.3	1
0	2	9.82	1.83	5.76 %	0.918	1877.3	1.83
0	4	5.15	3.50	5.63 %	0.876	984.13	3.50
0	8	2.77	6.49	6.28 %	0.812	530.87	6.49
0	12	1.98	9.10	5.73 %	0.759	378.43	9.11
0	16	1.54	11.6	4.98 %	0.729	295.65	11.6
1	1	18.37	1	0 %	1	1754.9	1.9
1	2	8.50	2.15	2.30 %	1.079	812.65	4.24
1	4	4.63	3.96	6.87 %	0.990	442.8	7.78
1	8	2.69	6.82	19.54 %	0.853	257	13.4
1	12	1.93	9.49	21.47 %	0.791	184.85	18.6
1	16	1.55	11.8	20.63 %	0.739	148.38	23.2
2	1	17.34	1	0 %	1	828.22	4.163
2	2	8.77	1.972	5.14 %	0.988	419.1	8.22
2	4	4.83	3.58	10.15 %	0.896	230.9	14.9
2	8	2.91	5.94	13.79 %	0.743	139.25	24.7
2	12	2.29	7.54	18.01 %	0.628	109.75	31.4
2	16	1.92	9.02	20.15 %	0.564	91.75	37.5
3	1	20.1	1	0 %	1	482.02	7.1
3	2	11.21	1.79	15.13 %	0.899	267.77	12.8
3	4	6.59	3.06	21.53 %	0.765	157.4	21.9
3	8	3.88	5.20	33.10 %	0.650	92.65	37.2
3	12	3.54	5.68	33.34 %	0.474	84.583	40.7
3	16	2.80	7.19	35.52 %	0.449	66.95	51.5
4	1	28.30	1	0 %	1	337.55	10.2
4	2	15.61	1.81	16.98 %	0.906	186.2	18.52
4	4	10.18	2.77	32.03 %	0.694	121.43	28.39
4	8	6.50	4.35	41.08	0.54384	77.583	44.47
4	12	5.56	5.08	50.10 %	0.423	66.35	51.97
4	16	4.34	6.51	48.47 %	0.407	51.817	66.54
5	1	46.05	1	0 %	1	273.83	12.59
5	2	34.45	1.33	39.89 %	0.668	204.85	16.83
5	4	21.39	2.15	45.22 %	0.538	127.18	27.11
5	8	12.95	3.55	51.95 %	0.444	77.033	44.76
5	12	9.41	4.89	53.06 %	0.407	55.967	61.61
5	16	8.57	5.37	53.44 %	0.335	50.967	67.65
6	1	79.47	1	0 %	1	235.02	14.67
6	2	56.63	1.40	34.07 %	0.701	167.47	20.59
6	4	38.50	2.06	48.14 %	0.515	113.87	30.28
6	8	24.34	3.26	57.69 %	0.408	71.983	47.90
6	12	17.85	4.45	56.39 %	0.371	52.783	65.3
6	16	15.72	5.05	57.88 %	0.315	46.483	74.18

FIGURE 2.25 – Résultats obtenus pour une configuration avec 1 processus par nœud ($\#ppn = 1$).

θ	#N	Elaps	Acc.	Comm.	Eff.	h/ss	Acc_Tg
0	2	8.59	2.09	7.87 %	1.04	1642.4	2.09
0	4	4.67	3.86	6.28 %	0.96	892.63	3.86
0	8	2.54	7.10	8.04 %	0.88	485.35	7.10
0	12	1.78	10.12	5.66 %	0.84	340.45	10.12
0	16	1.30	13.87	4.32 %	0.86	248.6	13.87
1	1	14.48	1.26	2.99 %	1.26	1384	2.49
1	2	7.62	2.41	6.32 %	1.20	727.92	4.73
1	4	4.21	4.35	16.1 %	1.08	402.58	8.56
1	8	2.29	8.01	22.64 %	1.00	218.98	15.74
1	12	1.76	10.38	21.47 %	0.86	168.95	20.41
1	16	1.41	12.96	17.07 %	0.81	135.38	25.47
2	2	8.22	2.10	9.53 %	1.05	392.93	8.77
2	4	4.76	3.63	20.71 %	0.90	227.68	15.14
2	8	2.88	6.01	19.89 %	0.75	137.72	25.03
2	12	2.35	7.36	20.59 %	0.61	112.38	30.68
2	16	1.95	8.87	24.30 %	0.55	93.283	36.96
3	1	18.44	1.09	14.59 %	1.09	440.33	7.83
3	4	6.01	3.35	25.61 %	0.83	143.62	24.01
3	8	4.25	4.74	36.60 %	0.59	101.47	33.98
3	12	3.37	5.98	43.87 %	0.49	80.467	42.85
3	16	2.87	7.03	48.81 %	0.43	68.517	50.32
4	4	10.74	2.63	46.12 %	0.65	128.13	26.91
4	8	6.55	4.31	52.92 %	0.53	78.167	44.11
4	12	4.81	5.87	60.70 %	0.48	57.4	60.07
4	16	4.18	6.76	63.08 %	0.42	49.883	69.12
5	1	52.69	0.874	39.44 %	0.87	313.32	11.00
5	4	17.90	2.57	56.44 %	0.64	106.48	32.38
5	8	11.65	3.95	56.36 %	0.49	69.283	49.77
5	12	9.66	4.76	54.08 %	0.39	57.467	60.00
5	16	7.83	5.87	53.61 %	0.36	46.567	74.05
6	4	36.44	2.18	61.46 %	0.54	107.77	31.99
6	8	21.46	3.70	61.88 %	0.46	63.45	54.34
6	12	16.50	4.81	61.27 %	0.40	48.8	70.66
6	16	13.82	5.74	57.61 %	0.35	40.883	84.34

FIGURE 2.26 – Résultats obtenus pour une configuration avec 2 processus par nœud ($\#ppn = 2$).



(a) 1 processus par nœud

(b) 2 processus par nœud

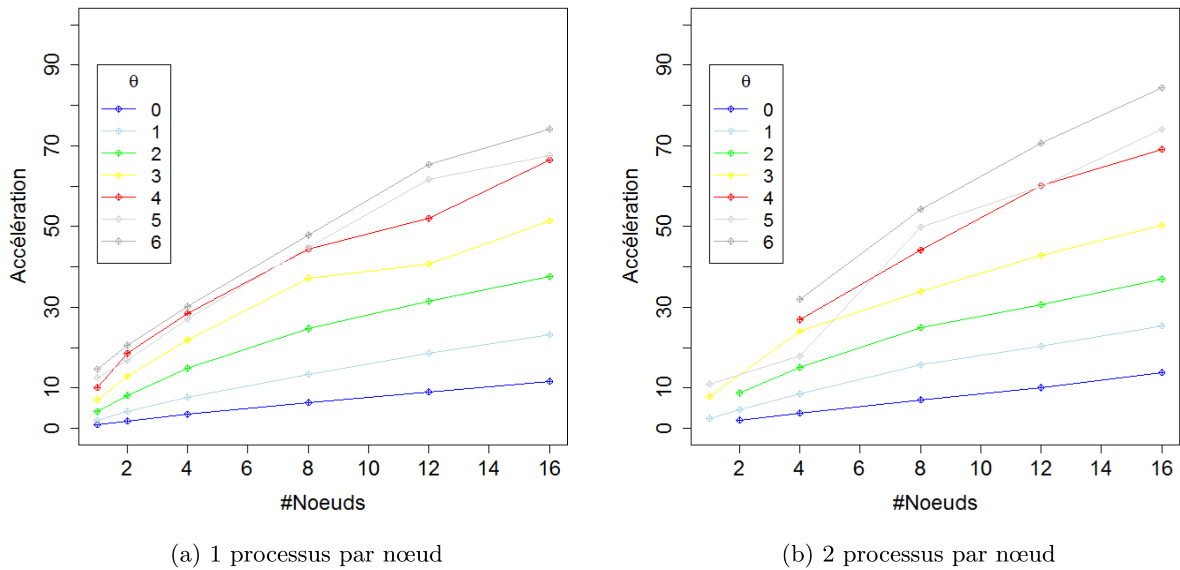
FIGURE 2.27 – Accélération relative par rapport au calcul sur un nœud.

Les figures 2.27 et 2.28 illustrent les performances de cette version. Les courbes de la figure 2.27 montrent l'accélération relative par rapport à une exécution sur un nœud pour chaque niveau temporel. Pour chaque courbe, la référence est donc le calcul avec le θ considéré et sur un nœud. Globalement, on constate que plus le niveau temporel est élevé, moins l'accélération est bonne.

Pour $\theta = 0$ et $\theta = 1$, on obtient une bien meilleure scalabilité par rapport aux autres valeurs de θ. Le fait qu'il n'y ait pas pour ces valeurs de θ de dégradation de performance peut s'expliquer par le fait qu'avec une infime proportion de mailles de niveau τ_0, aucune frontière de faible niveau n'existe. De plus, comme il n'y a que 2 sous-itérations dans ce cas, il n'y a donc pas de problème de synchronisation. Pour ce cas précis, on constate aussi que les calculs avec 2 processus par nœud offrent de meilleures performances que pour les calculs avec 1 processus par nœud. Le gain obtenu par l'utilisation de 8 cœurs au lieu de 16 est donc prépondérant par rapport au fait d'avoir plus de domaines.

Concernant les autres valeurs de θ (2 à 6), plus θ est grand, moins la scalabilité est bonne. On constate que les courbes avec 1 processus par nœud et 2 processus par nœud sont très proches. La version à 1 processus par nœud a pour elle l'avantage d'un plus faible nombre de domaines alors que la version à 2 processus par nœud bénéficie d'une meilleure efficacité de la parallélisation OpenMP lorsqu'on utilise 8 cœurs par rapport à 16 (cf. conclusions de l'étude en mémoire partagée).

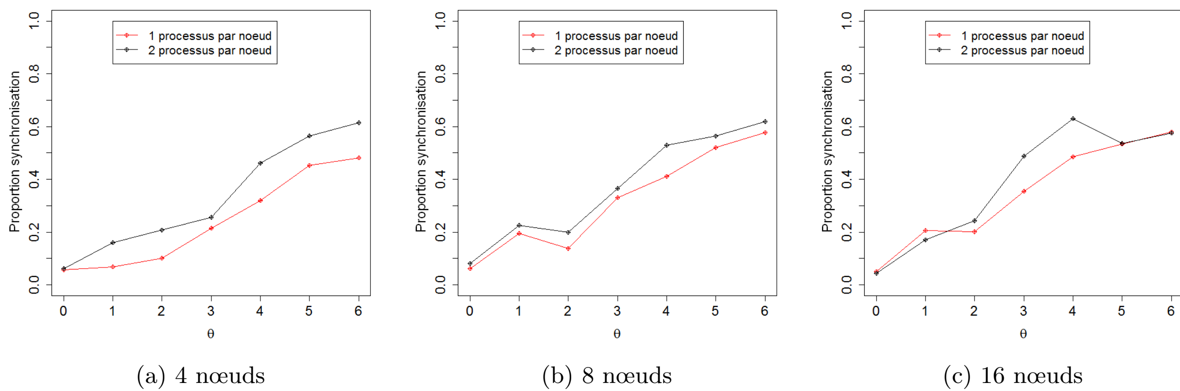
Les Figures 2.28a et 2.28b reprennent les mêmes mesures, mais cette fois-ci la référence est le calcul avec un pas de temps global. L'accélération est calculée par rapport au calcul $\theta = 0$ pour 1 nœud, cette fois-ci en prenant en compte le temps écoulé nécessaire pour atteindre le même temps physique. On constate que malgré une scalabilité globale moindre lorsqu'on augmente θ, la version MPI+OpenMP bénéficie fortement de l'utilisation du schéma d'intégration temporelle adaptatif. Avec un nombre de nœuds inférieur à 8, on constate que pour un processus par nœud, les performances sont similaires entre $\theta = 4$ et $\theta = 6$. Pour un plus grand nombre de nœuds, les calculs effectués avec $\theta = 6$ offrent de meilleurs performances.



(a) 1 processus par nœud

(b) 2 processus par nœud

FIGURE 2.28 – Accélération relative par rapport au calcul sur un nœud avec un pas de temps global.



(a) 4 nœuds

(b) 8 nœuds

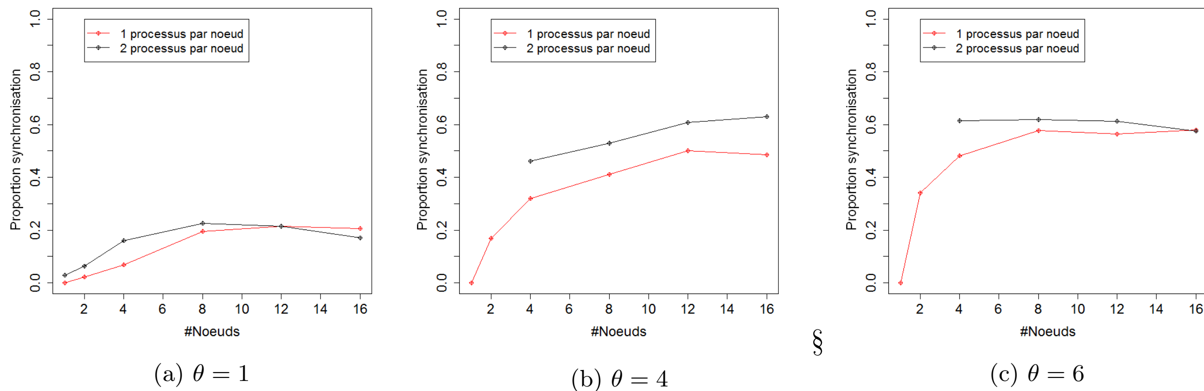
(c) 16 nœuds

FIGURE 2.29 – Proportion de synchronisation lorsque θ varie.

Analyse de l'évolution de l'importance des synchronisations

La Figure 2.29 montre l'évolution de la proportion de synchronisation (il s'agit du rapport entre temps passé dans les synchronisations et temps total) en fonction de θ pour différents nombres de nœuds. On mesure le temps passé dans la fonction *MPI_ Waitall* qui permet d'attendre la complétion de requêtes asynchrones.

On constate que plus θ augmente, plus la proportion de synchronisation augmente. Lorsqu'on utilise 2 processus par nœud, on a naturellement une proportion de synchronisation plus grande, étant donné qu'on a deux fois plus de sous-domaines. Cependant, pour 16 nœuds et $\theta = 6$, on constate qu'en utilisant 1 ou 2 processus par nœud, on obtient les mêmes proportions de



(a) $\theta = 1$ (b) $\theta = 4$ § (c) $\theta = 6$

FIGURE 2.30 – Proportion de synchronisation lorsque le nombre de nœuds varie.

synchronisation. On peut voir à l'Annexe A le détail de ce cas : la répartition des mailles dans les différents domaines n'est pas plus pénalisante lorsqu'on utilise 2 fois plus de domaines.

La Figure 2.30 montre l'évolution de la proportion de synchronisation en fixant θ et en faisant varier le nombre de nœuds. 3 valeurs de θ sont utilisées. On constate que plus θ augmente, plus la proportion de synchronisation augmente. Pour $\theta = 1$, la proportion de synchronisation augmente jusqu'à 8 nœuds, pour finalement se stabiliser. Pour $\theta = 4$ et 1 processus par nœud, la proportion de synchronisation augmente jusqu'à 12 nœuds et diminue pour 16 nœuds. Pour 2 processus par nœud, elle ne fait qu'augmenter. Pour $\theta = 6$ et 1 processus par nœud, la proportion de synchronisation augmente très vite pour atteindre 60% pour 8 nœuds et se stabilise à cette valeur. Les valeurs dont on dispose pour 2 processus par nœud sont toutes situées aux alentours de 60 %. Elles n'augmentent pas et diminuent même légèrement.

Globalement, pour notre cas test de référence (onde de souffle, 11M de cellules), on constate donc que la proportion de synchronisation augmente avec θ et le nombre de nœuds. Une autre information intéressante concerne le positionnement de ces synchronisations dans le flot de calcul complet.

Analyse des traces d'exécution

Les traces d'exécution donnent des informations concernant l'état des différents processus au cours du temps. Chaque barre du diagramme correspond à un processus et les couleurs correspondent à des états.

On a instrumenté le code de telle manière à pouvoir identifier dans quelle sous-itération on se situe. Dans nos traces, chaque sous-itération est colorée avec une couleur différente. Quand une sous-itération ne traite que les mailles de niveau τ_0 - c'est le cas d'une sous-itération sur deux, elle est en bleu cyan ; les autres sous-itérations disposent chacune de leurs couleurs. Les sections liées aux communications sont aussi identifiées : lorsqu'on démarre les communications (*MPI_Startall*) et lorsqu'on les attend (*MPI_Waitall*). Ces sections de synchronisation sont en **rouge**. En pratique, le temps passé dans l'ensemble des *MPI_Startall* est négligeable. Les sections avant et après le solveur aérodynamique sont en **noir**. Après la dernière section en **noir**, un processus n'a plus aucune opération à effectuer pour l'itération courante.

La Figure 2.31 montre le déroulement d'une itération pour un $\theta = 2$ et 8 processus.

Pour $\theta = 2$, il y a quatre sous-itérations :

FIGURE 2.31 – Trace d'exécution pour $\theta = 2$ et 8 nœuds (1 processus par nœud).

— à la première sous-itération (en **vert** sur la trace), on va traiter les mailles de niveaux temporels τ_2,τ_1,τ_0 ;
— à la seconde (en **bleu cyan**), uniquement les mailles de niveau τ_0 ;
— à la troisième (en **jaune**), les mailles de niveaux τ_1 et τ_0 ;
— et enfin, à la dernière (en **bleu cyan**) uniquement les mailles de niveau τ_0.

Hormis le **noir** et le **rouge**, on distingue majoritairement deux couleurs correspondant aux sous-itérations 1 (**vert**) et 3 (**jaune**). Comme on peut le voir sur la trace, on passe beaucoup moins de temps dans les sous-itérations (2 et 4) qui ne traitent que les mailles de niveau τ_0 (en **bleu cyan**). On constate que seuls les processus 1 et 2 en possèdent : on peut apercevoir du **bleu cyan** entre le **vert** et le **jaune** ainsi qu'à la fin, juste avant la fin de l'itération.

Le fait que le processus 1 ne puisse pas traiter directement les sous-itérations 2 et 4 implique qu'il partage une frontière de faible niveau temporel avec un autre processus, en l'occurrence le seul qui traite aussi ces mailles, le processus 2. Bien qu'initialement lors de la décomposition de domaine, on tente de ne pas faire apparaître de frontière de niveau 0, elles peuvent apparaître par la suite car les mailles peuvent changer de niveau temporel.

On peut aussi constater qu'il y a un problème d'équilibrage de charge : le temps passé à calculer (toutes les couleurs sauf le **rouge**) pour le processus 2 est plus important que pour les autres. Cependant, on remarque que ce processus passe aussi du temps dans la synchronisation au cours de la première sous-itération.

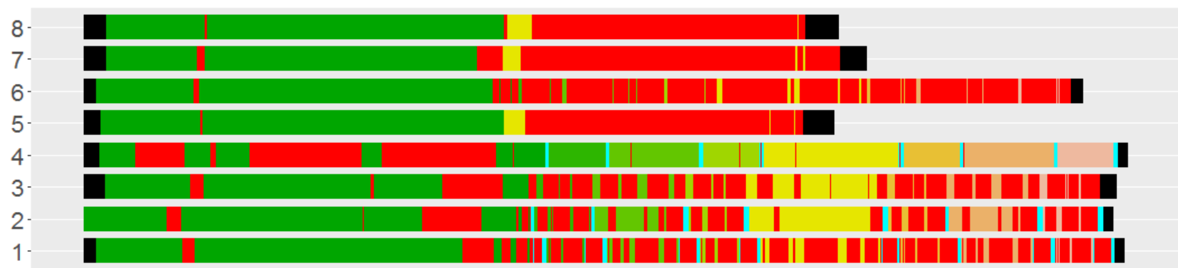


FIGURE 2.32 – Trace d'exécution pour $\theta = 4$ et 8 nœuds (1 processus par nœud).

La Figure 2.32 présente un calcul avec $\theta = 4$ et 8 processus. Cette fois-ci, il y a 16 sous-itérations. Le processus 4 est celui qui conditionne la durée finale de l'itération, car il est le dernier à terminer. On constate que ce processus traite en majorité des mailles de niveau temporel inférieur à θ : au cours de la première sous-itération, ce processus passe beaucoup de temps dans les synchronisations. Par contre, au cours des autres sous-itérations, ce processus n'est quasiment plus ralenti par des synchronisations.

Il y a d'autres informations intéressantes sur cette trace : on peut constater que les processus 5, 7 et 8 peuvent traiter la 9ème sous-itération (en jaune) en avance. Au cours de la 9ème sous-itération, les mailles traitées sont celles de niveaux τ_3,τ_2,τ_1 et τ_0.

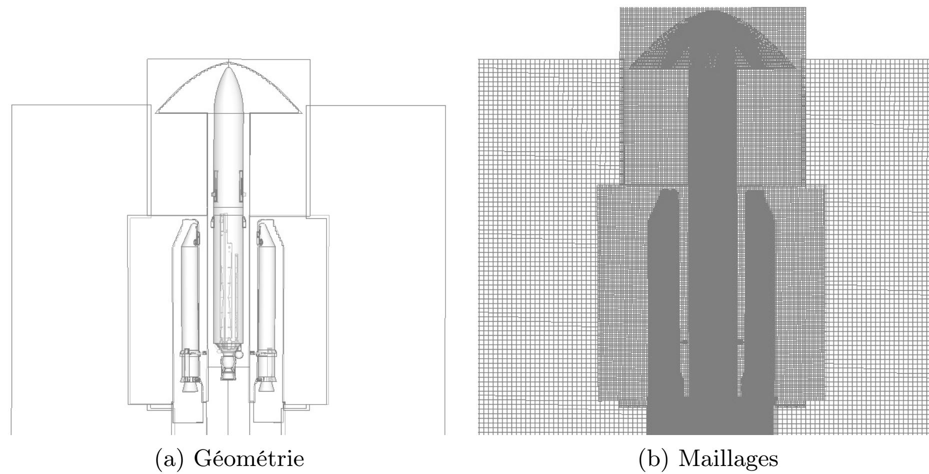
Cela signifie que les mailles de niveau τ_3 de ces domaines (5,7 et 8) ne sont pas liées aux mailles τ_3 des autres domaines. Les processus 1, 2, 3, 4 et 6 ne peuvent traiter cette sous-itération que beaucoup plus tard.

Discussion

Cette étude du comportement de l'algorithme d'intégration temporelle adaptatif montre qu'il est compétitif sur des clusters de calcul avec un parallélisme hybride MPI+OpenMP. Le gain est notable, bien qu'on constate qu'il reste une grande marge de progression. Le temps passé dans les synchronisations est important et il est principalement dû à la rigidité de la parallélisation telle qu'elle est écrite. D'un point de vue numérique, les dépendances sont beaucoup plus locales alors que la parallélisation MPI rajoute des dépendances dûes au fait qu'on doit traiter les différents niveaux temporels dans un ordre strict.

Ces raisons nous ont poussé à envisager la description du problème avec un parallélisme de tâches plus à même à ne capturer que les dépendances réelles. La parallélisation en tâches du solveur est décrite dans le chapitre suivant.

2.3.2 Calcul de la séparation des EAP



(a) Géométrie (b) Maillages

FIGURE 2.33 – Géométrie et maillages utilisés.

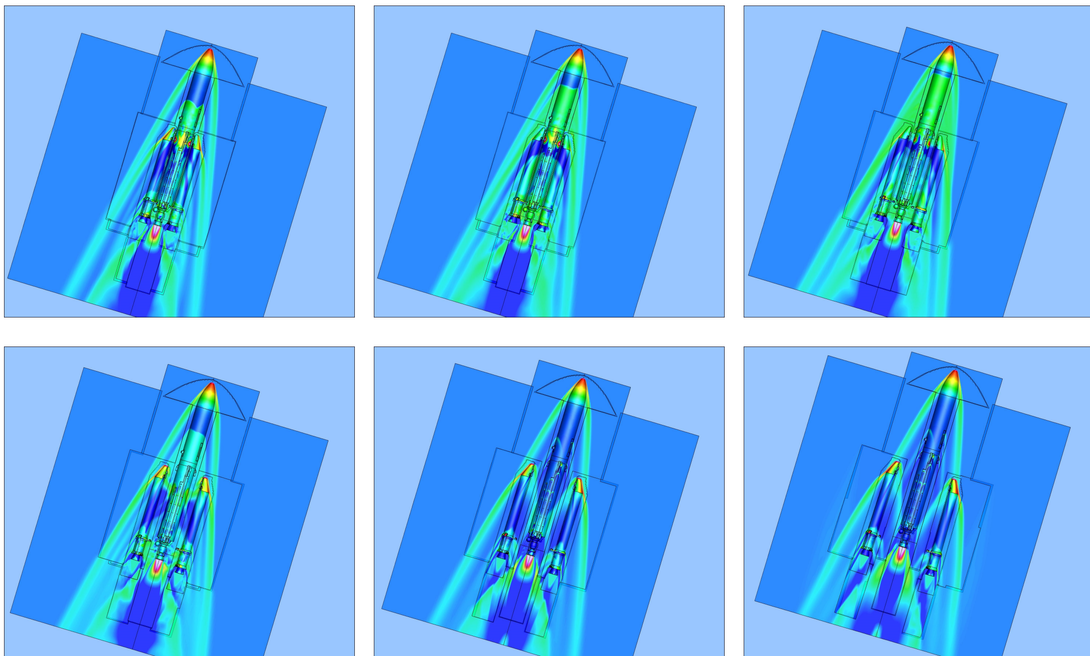


FIGURE 2.34 – Illustration du calcul de séparation des EAP.

La séparation des EAP intervient à “H0+2min21s”. Lorsque les EAP sont quasiment vides, ils sont séparés de l'étage principal afin d'alléger le lanceur. La séparation se fait via un découpage pyrotechnique des attaches qui lient les EAP à l'étage principal. A ce moment-là, 8 fusées d'éloignement sont allumées de manière à assurer l'intégrité de l'étage principal.

Le calcul présenté dans cette section se déroule à ce moment précis : depuis l'allumage des fusées d'éloignement jusqu'à ce que les EAP soient suffisamment éloignés de l'étage principal.

Plusieurs maillages sont utilisés dans ce calcul et sont liés à 3 différents corps : l'étage prin-

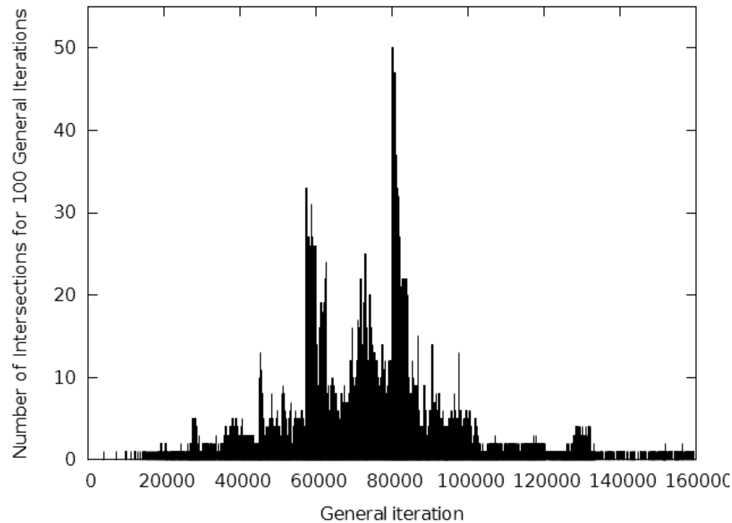


FIGURE 2.35 – Evolution de l'importance des intersections au cours du calcul.

cipal, le premier EAP et le second. Les maillages sont fins autour des étages et plus grossiers ailleurs. La Figure 2.33 présente la géométrie et les maillages.

Le calcul se déroule en 2 phases : tout d'abord, il faut faire converger une solution stationnaire. Ensuite, l'allumage des fusées d'éloignement est simulé via une condition limite particulière qui évolue de manière préprogrammée en fonction du temps. Seules les fusées d'éloignement et la balistique ont une influence sur les déplacements de maillages.

La Figure 2.34 illustre le calcul. Sur la première image, les fusées d'éloignement sont allumées. Sur les 3 premières images, on peut voir un décollement de la couche limite dû à cet allumage. Sur la quatrième image, les fusées d'éloignement sont éteintes et les déplacements qui suivent proviennent uniquement de la balistique. Sur la dernière image, les EAP sont un peu plus éloignés.

Conditions du calcul

Étant donné que l'on est dans le cas d'un calcul de démonstration particulièrement coûteux et que l'on se concentre principalement sur la capacité d'effectuer le calcul d'intersection de manière asynchrone, on présente un unique calcul.

Ce calcul a été effectué sur un cluster composé de nœuds disposant chacun de 2 processeurs Haswell de 12 cœurs (24 cœurs par nœud au total). On utilise un processus d'intersection pour 24 processus aérodynamiques. Le *binding* des processus peut cependant varier. Ainsi, au cours du calcul, on a testé plusieurs configurations :

— 1 processus d'intersection placé seul sur un nœud, le maître seul sur un autre nœud et 4 processus par nœud pour les processus aérodynamiques (24 processus aérodynamiques pour un total de six nœuds) ;
— 1 processus d'intersection sur le même nœud que le processus maître et toujours 4 processus par nœud pour les processus aérodynamiques.

La configuration avec plusieurs processus d'intersection n'a pas été effectuée car il y avait trop peu de calcul pour justifier l'utilisation de plusieurs processus d'intersection.

L'algorithme d'intersection permet plusieurs niveaux de robustesse. Une des différences importante provient de la sélection des cellules potentiellement concernées par le calcul d'intersection : la version la moins robuste présélectionne moins de cellules candidates mais il peut

arriver que le calcul d'intersection qui résulte de ce choix soit incorrect, ce qui provoque une divergence dans le solveur aérodynamique dès l'itération suivante. Dans le cas du calcul présenté ici, il peut arriver que le niveau le moins robuste produise une intersection incorrecte : ce n'est pas systématique, ni même courant, mais suffisamment gênant pour que l'on choisisse un niveau de robustesse plus élevé.

La cinématique extrapolée est considérée correcte si l'écart est inférieur à 10% comparé à la vraie cinématique dans la direction où cet écart est le plus important. Sur notre cas de calcul, le mécanisme d'extrapolation de la cinématique n'a provoqué une intersection rejetée que 19 fois pour 6794 intersections calculées (moins de 0.3%), ce qui est satisfaisant.

La Figure 2.35 représente le nombre d'intersections au cours du calcul. En abscisse, on dénombre les itérations, chaque graduation représentant 100 itérations générales. En ordonnée, pour 100 itérations générales du solveur aérodynamique, on compte le nombre d'itérations où il y a eu un calcul d'intersection. On constate que les intersections n'ont pas toujours la même importance : au début du calcul il y a peu d'intersections, puis de plus en plus. La raison pour laquelle le nombre d'intersections devient moins important est dû à la finesse des maillages. Au début, les EAP sont proches de l'étage principal et les maillages sont plus fins car les corps sont à proximité. Au début de la séparation, les corps s'éloignent de plus en plus vite et on est toujours avec des maillages fins, donc le nombre d'intersections augmente. Quand les EAP s'éloignent de l'étage principal, les intersections se font avec des maillages de plus en plus grossiers. De ce fait, il y a moins d'intersections.

Coûts des différentes opérations

Dans ce cas de calcul, certains coûts sont fixes. Ainsi, on peut estimer que le coût d'une itération du solveur aérodynamique est toujours au alentour de 7.5 secondes.

Le coût d'application d'une nouvelle topologie est au environ de 7.0 secondes : il comprend la recréation des types MPI pour la communication entre les processus aérodynamiques et le processus maître, l'envoi de la nouvelle topologie et le calcul des distances des cellules aux parois.

#NCPUs	Robustesse	Temps moyen
12	Minimale	10.5 s
12	Maximale	75 s
24	Maximale	27 s

FIGURE 2.36 – Coût du calcul des intersections en fonction du nombre de cœurs alloués et du niveau de robustesse utilisé.

Concernant le coût des intersections, il varie en fonction du nombre de cœurs de calcul alloués au processus de calcul d'intersection et de la robustesse choisie. La Figure 2.36 montre ces différents coûts en fonction de ces différents paramètres. Le temps moyen est le temps passé à calculer une nouvelle topologie par un processus d'intersection.

Analyse à différents moments du calcul

Comme on l'a vu sur la Figure 2.35, l'importance des intersections varie au cours du calcul. Cette section présente plusieurs étapes du calcul. Dans les cas présentés, on utilise le niveau de robustesse maximale ainsi que 24 cœurs pour le processus d'intersection, soit la totalité d'un nœud. 6 nœuds sont dédiées aux calculs aérodynamiques.

La Figure 2.37 présente un premier cas où la fréquence du calcul des intersections est faible. En ordonnée, on a les différents types de processus : M pour le maître, I pour le processus

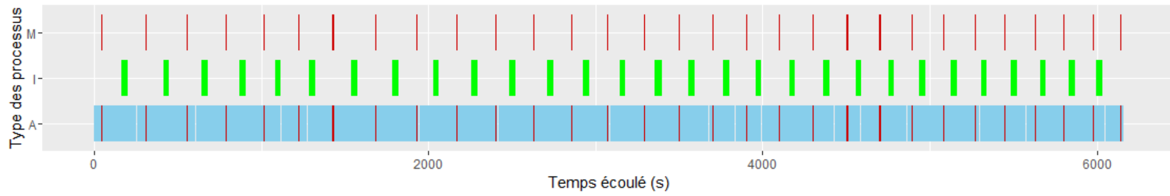


FIGURE 2.37 – État des processus (fréquence de calcul des intersections faible).

d'intersection et A représente l'ensemble des processus calculant l'aérodynamique. En abscisse, il s'agit du temps écoulé. Les différentes phases représentées sont :

— en **rouge**, l'application des intersections qui concernent le maître et les processus aérodynamiques, qui communiquent pendant cette étape ;
— en **vert**, le calcul des intersections ;
— en **bleu**, les calculs du solveur aérodynamique.

Dans notre cas de calcul, le coût des calculs aérodynamiques est prépondérant (6 nœuds y sont dédiés). On souhaite donc que le calcul des intersections ait le moins d'impact possible sur le coût total de la simulation. Ce coût se traduit par les couleurs **verte** et **rouge**. Notre objectif est que les processus dédiés à l'aérodynamique effectuent en majorité des calculs aérodynamiques (en **bleu**).

Sur cette première séquence, on constate que le calcul des intersections est totalement recouvert. Si le temps passé à appliquer les intersections n'est pas négligeable, il reste faible comparé au temps passé dans le solveur aérodynamique.

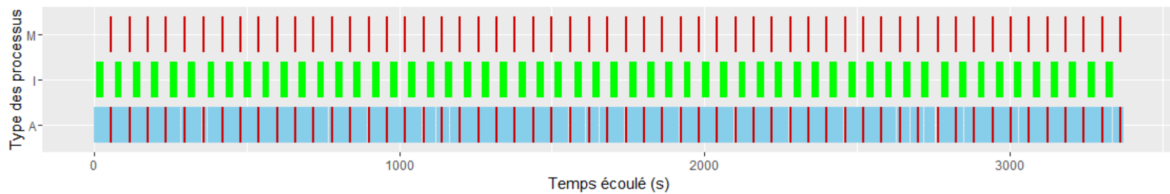


FIGURE 2.38 – État des processus (fréquence de calcul des intersections moyenne).

La Figure 2.38 présente un cas où le calcul des intersections devient plus courant ; on constate qu'elles sont toujours masquées, mais cependant le coût d'application des intersections est plus important.

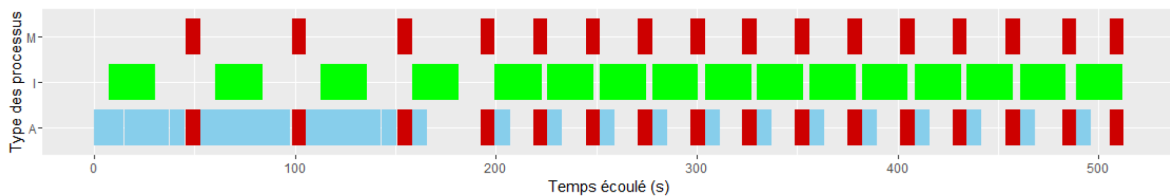


FIGURE 2.39 – État des processus (calcul des intersection systématique).

Dans le cas présenté sur la Figure 2.39, on a un calcul des intersections pour chaque itération du solveur. C'est le cas le moins favorable . Cette fois, on constate que le coût de la simulation est dominé par le calcul des intersections et les processus aérodynamiques (plus nombreux, donc plus consommateurs en ressource de calcul) passent beaucoup de temps *idle*. Dans cette configuration

il pourrait être bénéfique d'utiliser plusieurs processus de calcul d'intersection ; cependant, au cours de la simulation cette situation n'est pas courante comme on a pu le voir sur la Figure 2.35.

Discussion

Pour ce cas de calcul, le mécanisme d'extrapolation permet de masquer la plupart du temps le calcul des intersections. Les choix faits pour l'extrapolation de la cinématique sont de qualité suffisante et permettent de garantir la qualité des résultats sans trop de rejet. L'application des intersections est cependant plus coûteuse : transférer la topologie du processus maître aux processus chargés des calculs aérodynamiques est l'étape la plus coûteuse en temps et celle-ci n'est pas recouverte. Effectuer cette opération de manière asynchrone n'est pas impossible dans l'absolu, mais les limitations liées à l'organisation actuelle avec un maître qui doit disposer de l'ensemble du domaine rendent cette amélioration difficile (principalement en raison des problèmes de capacité mémoire).

Il arrive dans cette simulation que le calcul des intersections prenne le pas sur les calculs du solveur aérodynamique. Il serait judicieux de pouvoir adapter à la volée le nombre de processus lié aux intersections pour faire face à cette situation. Actuellement, celà n'est possible qu'en arrêtant le calcul et en faisant varier le nombre de processus lors de sa reprise.

Pour aller plus loin, il faudrait pouvoir exécuter de manière concurrente les opérations liées au solveur aérodynamique et les opérations liées aux calculs d'intersection et regrouper les calculs en fonction de la localité des données. Le développement d'une version en tâches pourrait répondre à ces problématiques. En utilisant le mécanisme d'extrapolation de la cinématique tel qu'il existe et en ordonnançant à la fois des tâches de calcul d'intersection et de calcul aérodynamique sur les mêmes processus, il pourrait être possible de calculer les intersections toujours en avance, et cette fois-ci de manière locale. L'application des intersections auraient alors un coût quasiment nul, mais cependant, il faudrait prévoir de stocker une partie de la topologie en double (celle qui est concernée par les calculs d'intersection).

Le développement de cette version MPI+OpenMP a permis la résolution de cas de calcul plus importants. Il s'agit désormais de la version de référence utilisée en production. Le schéma d'intégration temporelle adaptatif bénéficie bien de la parallélisation en mémoire distribuée.

Cette version a notamment été utilisée pour les calculs de la thèse de Grégoire Pont [Pon15] où des modèles de turbulence auto-adaptatifs ont été évalués. Les calculs de séparation d'étage sont devenus plus accessibles, alors qu'ils s'agissaient de calculs qui n'étaient effectués qu'exceptionnellement en raison du temps nécessaire à leur réalisation. Un calcul comme celui de l'onde de souffle présenté prend 15 fois moins de temps que précédemment. Un calcul de séparation des EAP tel que celui présenté, même s'il reste coûteux, est désormais suffisamment abordable pour être traité effectivement en production.

Chapitre 3

Conception et mise en œuvre d'une version parallèle du solveur aérodynamique avec une expression en graphe de tâches et s'exécutant sur un runtime

Sommaire

3.1 Motivations **47**

3.2 Modèle d'expression du parallélisme en graphe de tâches et support d'exécution **48**

3.2.1 Expression du parallélisme et modèles de programmation utilisant un graphe de tâches 49

3.2.2 Runtime pour les architectures modernes 51

3.2.3 Description et fonctionnalités du runtime StarPU 52

3.3 Version parallèle de FLUSEPA en mémoire distribuée utilisant StarPU **55**

3.3.1 Réalisation d'une version symbolique pour le solveur aérodynamique en mémoire partagée 56

3.3.2 Préliminaires pour la mise en place d'une version en tâches en mémoire partagée 64

3.3.3 Une version en tâches en mémoire partagée de FLUSEPA 66

3.3.4 Optimisation et validation expérimentale de la version en tâches en mémoire partagée 71

3.3.5 Mise en place d'une version distribuée 80

3.1 Motivations

Comme nous avons pu le voir dans le chapitre précédent le code FLUSEPA pose des problématiques intéressantes pour une parallélisation efficace. Dans une première version, nous avons réalisé une version MPMD qui place les deux opérations principales, à savoir le solveur aérodynamique et le calcul d'intersection de maillages sur deux processus différents, ce qui implique donc

des phases de synchronisation afin de transférer les données. Une parallélisation qui effectuerait ces deux étapes l'une à la suite de l'autre serait pénalisée par le fait que les équilibrages de charge pour les calculs d'intersection et pour le solveur aérodynamique n'ont rien à voir. De plus, cette solution ne tirerait pas parti de l'indépendance entre les deux types de calcul. L'idéal serait de pouvoir effectuer ces opérations de manière plus asynchrone en utilisant les mêmes processus afin de pouvoir tirer parti de la localité des données et de ne pas avoir de synchronisation forte. L'autre point intéressant d'un point de vue algorithmique pour le développement d'une version parallèle plus efficace est le solveur avec son schéma d'intégration temporelle adaptatif. L'utilisation de méthodes classiques de parallélisation a des limitations de par la présence de synchronisations importantes. Cependant, les dépendances réelles sont plutôt locales et il faudrait tirer parti de ces dépendances plus fines et permettre ainsi une réduction des synchronisations. Dans ce chapitre, nous décrivons une parallélisation basée sur une expression via un graphe de tâches du solveur aérodynamique et qui s'exécutera sur le support d'exécution (*runtime*) StarPU.

3.2 Modèle d'expression du parallélisme en graphe de tâches et support d'exécution

Les calculateurs ont subi une évolution forte depuis la dernière décennie. Certaines de ces évolutions ont imposé des choix aux programmeurs : rendre les applications dépendantes des nouvelles fonctionnalités matérielles (et ainsi rompre la compatibilité avec les architectures précédentes), les ignorer (et ne pas bénéficier du gain potentiel) ou alors gérer les choses aux mieux au prix d'une complexité accrue du code. Ce problème est connu sous le nom de *portabilité des performances* et c'est l'objectif majeur de l'utilisation des supports d'exécution ou runtime.

Le premier but d'un runtime est de fournir des *abstractions*. Les runtimes offrent une interface de programmation uniforme pour un sous-ensemble de dispositifs matériels (OpenGL [WNDS99] et DirectX [Lun08] sont des exemples de runtime dédiés à l'utilisation de cartes graphiques accélératrices) ou pour des fonctionnalités logicielles bas-niveau (par exemple, les implémentations de thread POSIX [But97]). Ils sont pensés comme des bibliothèques utilisateur qui enrichissent les fonctionnalités basiques fournies par les appels système. Les applications peuvent ensuite utiliser ces API d'une manière portable. Les détails bas-niveau et matériels sont cachés et traités au niveau des runtimes eux-mêmes. L'adaptation du runtime à de nouvelles architectures se fait via des pilotes. Les abstractions fournies par le runtime permettent ensuite la portabilité. Les abstractions seules ne sont pas suffisantes pour fournir la portabilité des performances car elles ne traitent pas les détails bas-niveau pour améliorer les performances.

Du coup, le second rôle des runtimes est d'*optimiser* les appels aux abstractions afin qu'elles utilisent les ressources de la manière la plus efficace possible. Le choix de cette ressource est basé sur des algorithmes d'ordonnancement et des heuristiques afin d'optimiser une métrique donnée. Ce mécanisme permet aux applications de bénéficier des dernières avancées matérielles sans rompre leur compatibilité. Ainsi, la mise en œuvre d'optimisations conjuguée à l'utilisation d'abstractions permettent aux runtimes d'offrir la portabilité des performances.

Dans le cas particulier du calcul parallèle, d'autres approches ont été utilisées. Beaucoup d'applications scientifiques et de bibliothèques numériques (notamment celles concernant les solveurs linéaires) intègrent leur propre mécanisme d'ordonnancement ou font confiance à un ordonnancement statique, soit pour des raisons historiques, soit pour éviter le potentiel surcoût de l'utilisation d'un runtime. Cependant, maintenant que le nombre de cœurs de calcul augmente fortement, que la hiérarchie des caches et de la mémoire se complexifient toujours plus, il devient de plus en plus difficile de se passer d'un runtime capable d'ordonnancer correctement des tâches

et de virtualiser l'architecture. Le runtime devra alors considérer des tâches de calcul et leurs dépendances en entrée et devra ordonnancer dynamiquement ces tâches sur les unités de calcul disponibles afin de minimiser le temps d'exécution.

Avec l'arrivée des accélérateurs dans le monde du calcul scientifique, ces runtimes font face à de nouveaux challenges. Ils doivent désormais être capables de décider aussi quels types de ressources utiliser : selon le type de tâche, on peut avoir un comportement très différent selon que la tâche soit exécutée sur un accélérateur (cœur spécifique) ou sur un processeur plus classique (cœur générique). Quand l'accélérateur dispose de sa propre mémoire, les données d'entrée de la tâche doivent être copiées de la mémoire centrale à la mémoire de l'accélérateur avant que la tâche puisse s'exécuter. Le résultat final doit ensuite être copié en mémoire centrale une fois la tâche terminée. Le coût d'une copie entre la mémoire centrale et la mémoire de l'accélérateur n'est pas négligeable. Ce coût de transfert, de même que les dépendances entre les tâches, doivent donc être pris en compte par les algorithmes d'ordonnancement. Il est aussi possible de faire en sorte que les transferts aient lieu en avance et de manière asynchrone pour recouvrir le temps de transfert par des calculs.

3.2.1 Expression du parallélisme et modèles de programmation utilisant un graphe de tâches

Les runtimes modernes ont donc pour but de virtualiser le niveau matériel de l'architecture et de permettre la portabilité des performance d'un code écrit pour les utiliser. Dans la plupart des cas, cela est obtenu via une abstraction se basant sur l'utilisation d'un graphe orienté acyclique (Directed Acyclic Graph, DAG) de tâches. Dans ce DAG, les sommets représentent les tâches à exécuter tandis que les arêtes représentent les dépendances.

Chaque runtime dispose de sa propre API qui inclut une ou plusieurs manières de décrire les dépendances dans le DAG. Cependant, on peut considérer qu'il y a deux manières principales de décrire les dépendances. La plus naturelle est de le faire de manière explicite entre les tâches. Malgré la simplicité de ce concept, cette approche est compliquée à mettre en œuvre car certains algorithmes imposent des dépendances difficiles à exprimer. Il est aussi possible d'avoir des dépendances calculées de manière implicite par le runtime en utilisant la consistance séquentielle [AK02]. Dans cette dernière approche, les tâches sont soumises dans un certain ordre et la manière dont elles accèdent aux données est aussi décrite. L'utilisation de dépendances implicites laisse au runtime le calcul des dépendances par l'utilisation d'une analyse qui assure que la parallélisation ne viole pas les dépendances imposées par la consistance séquentielle. Cette approche est similaire à celle présente au sein des processeurs superscalaires. L'analyse est effectuée sur les tâches et sur les données d'entrée et de sortie qui lui sont associées. Prenons l'exemple d'une tâche 1 qui agit sur les données x et y, respectivement en lecture-écriture et en lecture. Si une tâche 2 agit sur la donnée x en lecture, l'analyse superscalaire détectera qu'une dépendance est nécessaire pour assurer la consistance séquentielle.

Un autre paradigme pour la gestion des dépendances est la soumission récursive. Il peut être pratique pour le programmeur que la terminaison d'une tâche déclenche d'autres tâches. Les runtimes offrent souvent cette possibilité via l'utilisation de *callbacks*, un mécanisme qui consiste à effectuer une action spécifique à la fin d'une tâche. Selon le contexte, le programmeur peut vouloir utiliser plusieurs de ces paradigmes pour décrire son problème sous la forme d'un DAG.

Se contenter d'un seul de ces paradigmes peut conduire à des codes plus simples et plus faciles à maintenir. De plus, l'utilisation d'un modèle peut permettre des optimisations. Le fait de n'utiliser que la consistance séquentielle et de calculer les dépendances de manière implicite revient à utiliser le modèle STF (*Sequential Task Flow* [AK02]). Il s'agit de soumettre une série

de tâches via l'utilisation d'une fonction non-bloquante qui délègue l'exécution de la tâche au runtime. Après la soumission, le runtime ajoute la tâche au DAG ainsi que les dépendances afférentes qui auront été automatiquement calculées. L'exécution effective de la tâche ne peut avoir lieu que lorsque toutes les dépendances sont satisfaites. Comme mentionné précédemment, ce paradigme est parfois considéré comme superscalaire car il imite les fonctionnalités des processeurs superscalaires pour lesquels les instructions proviennent d'un unique flot, mais où celles-ci peuvent être exécutées dans un ordre différent de celui de leur soumission, et possiblement en parallèle.

Une autre approche, particulièrement intéressante lorsqu'il s'agit d'utiliser de très grands clusters de calcul est le modèle PTG (*Parametrized Task Graph* [CL95]). Dans ce modèle, les tâches ne sont pas énumérées mais paramétrisées et les dépendances entre les tâches sont explicites. Cette propriété peut être utilisée pour représenter le DAG de manière compacte et ainsi limiter le coût de création du DAG pour des problèmes de grande taille. De plus, avec le modèle PTG, le DAG est parcouru localement en accord avec l'exécution, alors que le DAG doit être totalement calculé avec l'approche STF. Cette caractéristique peut être primordiale dans un contexte utilisant une mémoire distribuée, puisque seule une portion du DAG est présente sur chaque nœud. Cette approche présente par contre des limitations lorsque le flot de tâches dépend d'un calcul effectué précédemment, alors que ça ne pose aucun problème avec le modèle STF.

Concernant l'utilisation d'une mémoire distribuée avec le modèle STF, deux options sont possibles : soit l'ensemble du DAG peut être soumis sur l'ensemble des nœuds, soit les communications peuvent être décrites explicitement via des tâches spécifiques.

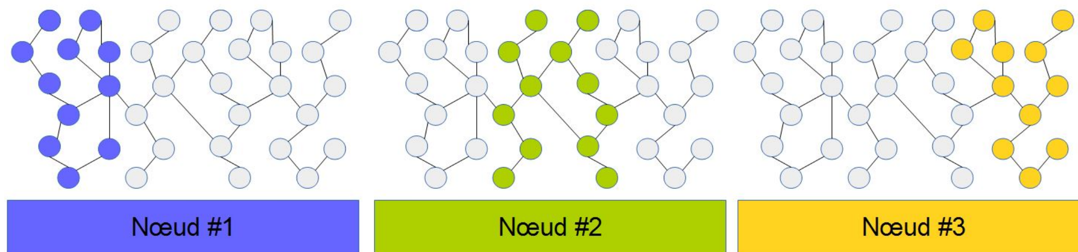


FIGURE 3.1 – Connaissance globale du DAG.

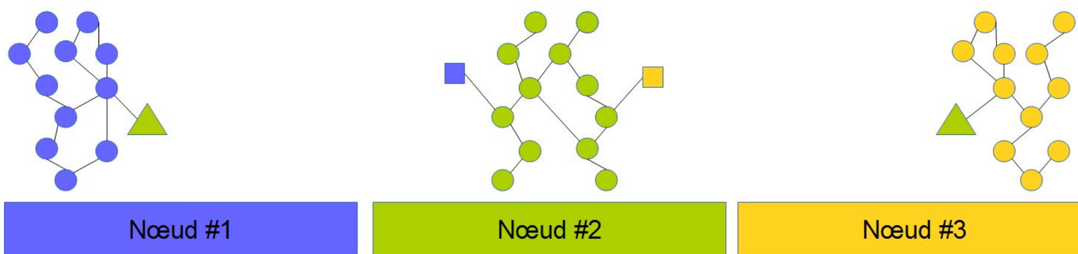


FIGURE 3.2 – Communications explicites.

La première approche présente l'avantage d'une vision globale : chaque nœud de calcul dispose de l'ensemble du DAG et certaines tâches sont marquées pour être exécutées sur un nœud spécifiquement. Il est alors très simple de remettre en cause l'équilibrage de charge. Il suffit d'attribuer un nœud différent à certaines tâches et le runtime pourra ensuite effectuer les transferts de données nécessaires. La Figure 3.1 illustre cette approche : 3 nœuds sont utilisés et l'ensemble

du DAG est soumis sur ces 3 nœuds. Chaque nœud n'exécute qu'une partie du DAG, indiquée par la couleur des nœuds. La nécessité de représenter l'ensemble du problème sur chacun des nœuds peut représenter un surcoût très important en fonction du problème traité.

La seconde nécessite uniquement la portion du DAG qui sera exécutée sur chaque nœud ; ainsi lorsque davantage de nœuds sont utilisés, le surcoût lié à l'utilisation du runtime évolue dans des proportions raisonnables. Cependant, il est plus difficile pour le programmeur d'exprimer les interactions entre les nœuds et il peut être plus difficile de corriger un déséquilibre. Il est toutefois possible de corriger ce déséquilibre au niveau applicatif (par exemple pour un problème présentant plusieurs itérations, en intervenant entre les itérations). La figure 3.2 reprend le même DAG que sur la figure précédente. Cette fois chaque nœud ne possède que sa portion du DAG et des tâches correspondant aux communications ont été ajoutées : les triangles représentent une tâche d'envoi de données et les carrés une tâche de réception.

Entre ces deux approches, la première a l'avantage de la simplicité de programmation, la seconde à l'avantage de pouvoir passer à l'échelle. La prise en compte du coût de création du DAG est donc un élément de choix déterminant entre ses deux approches.

3.2.2 Runtime pour les architectures modernes

Le développement de runtimes pour les architectures multi-cœurs et hétérogènes ont vu le jour ces dernières années. Comme décrit plus haut, la plupart de ces runtimes utilise une description en tâches pour exprimer la concurrence et les dépendances via un graphe de tâches qui représente l'application à exécuter. Les différences principales entre les développements réalisés proviennent du modèle de programmation associé, de la manière d'assurer les transferts de données entre les unités de calcul, et enfin de la manière de prendre en charge l'ordonnancement des tâches.

Les bibliothèques d'algèbre linéaire ont été les premières à utiliser le paradigme du DAG et les runtimes [AAD+11a, AAD+11b, AAD+10, BBD+11, BLKD08, KLDB10, QOQOC+08, QOIQOvdG09]. L'utilisation de runtime est centrale pour des bibliothèques telles que PLASMA [.09b], MAGMA [.09a], DPLASMA [BBD+10], CHAMELEON [AAD+10] ou FLAME [VCvdG+09]. L'algèbre linéaire dense a été un domaine pionnier dans l'utilisation des runtimes car les algorithmes considérés permettent la génération de DAG très grands et avec des tâches ayant des granularités variables en terme de calcul (BLAS de niveaux 1, 2 ou 3). Désormais, d'autres types d'applications utilisent les runtimes, par exemple en algèbre linéaire creuse [Lac15, Lop15], en calcul d'interaction à N corps (FMM) [LY12, ABC+14], pour des problèmes d'éléments finis [GGB14], de différences finis [Bau14] ou pour des méthodes de Galerkin discontinues [Boi14].

Certains runtimes ont été développés spécifiquement pour certaines bibliothèques d'algèbre linéaire. L'un d'entre eux est TBLAS [SYD09], qui fournit une interface simple pour développer des applications d'algèbre linéaire dense et qui permet d'automatiser les transferts de données. TBLAS impose au programmeur d'affecter les données aux différentes unités de calcul, mais il supporte l'utilisation de taille de bloc variable. Le runtime QUARK [KD09] a été spécifiquement conçu pour ordonnancer des noyaux de calcul d'algèbre linéaire pour des machines multi-cœurs. Son algorithme d'ordonnancement est basé sur le vol de travail et il dispose d'une forte capacité à passer à l'échelle comparé à d'autres runtimes dédiés à ce type de problème. Enfin, le runtime SuperMatrix [CZB+08] reprend la même idée avec une représentation hiérarchique de la matrice qui est utilisée.

Concernant l'utilisation de ressources hétérogènes, Qilin [LHK09] fournit une interface pour soumettre des noyaux qui travaillent sur des tableaux qui sont ensuite réparti entre les différentes unités de calcul d'une machine hétérogène. Plus encore, Qilin est capable de compiler dynami-

quement des codes à la fois pour CPUs (en se basant sur Intel TBB [Rei07]) et pour GPU (via CUDA). Pour ce qui est des problèmes d'équilibrage de charge et des optimisations concernant les communications, une autre bibliothèque connue est Charm++ [KK93]. Charm++ a été amélioré pour founir un support pour les accélérateurs tels que le Cell ou les GPUs [KK11].

La plupart des runtimes récents et disponibles aujourd'hui ne visent pas spécifiquement un type d'application et fournissent une API générique. Des runtime tels que KAAPI/X-KAAPI [GLMFR13], APC+ [HSc12], Legion [But12] ou Realm [TBA14] offrent un support pour les plateformes hétérogènes disposant de CPUs et de GPUs. La gestion de la cohérence mémoire est basée sur un méchanisme proche de celui d'une DSM (mémoire distribuée partagée) : chaque bloc de donnée est ainsi associé à un tableau qui permet de savoir quel est l'état de la donnée pour chaque unité de calcul. Le projet StarSs [BHL+09, ABI+09] consiste en l'association d'un langage d'annotations (pragmas) et de runtimes ciblant des architectures différentes. PaR-SEC [BBD+12, BBD+13] (anciennement DAGuE) est un runtime qui est capable d'ordonnancer dynamiquement des tâches au sein d'un nœud en mémoire partagée en utilisant une politique basée sur un vol de tâche prenant en compte l'architecture du nœud. Il a d'abord servi aux applications d'algèbre linéaire, puis a été étendu pour être plus générique. StarPU [ATNW11] fournit une interface générique pour développer des applications à base de graphe de tâches. Il supporte les architectures multi-cœurs équipées d'accélérateurs et dispose d'extensions lui permettant de fonctionner en mémoire distribuée. Ce runtime est capable de gérer les transferts de données de manière transparente et dispose de nombreuses fonctionnalités. Il est décrit plus en détail dans la section suivante.

La plupart des runtimes cités ci-dessus ont conduit à l'adoption de fonctionnalités similaires dans le dernier standard OpenMP (4.0). L'annotation *task* a été étendue avec la clause *depend* qui permet au runtime OpenMP de détecter automatiquement les dépendances entre les tâches et de les ordonnancer. Ce standard dispose aussi d'annotations permettant de déporter les calculs sur des accélérateurs. Cependant, il ne fournit pas encore les moyens d'utiliser les unités de calcul de manière aussi flexible que les runtimes les plus avancés.

Le travail présenté dans ce chapitre se base sur StarPU qui est décrit plus en profondeur dans la section suivante.

3.2.3 Description et fonctionnalités du runtime StarPU

StarPU [5] est un runtime développé par l'équipe STORM [6] (anciennement RUNTIME) à Inria Bordeaux-Sud-Ouest. StarPU propose une interface pour implémenter et paralléliser des applications ou des algorithmes qui peuvent être décrits par un graphe de tâches. StarPU permet notamment d'exploiter le paradigme STF. Il est aussi possible de soumettre des tâches dans des *callbacks* ou d'exprimer les dépendances de manière explicite, mais nous nous concentrerons dans cette thèse sur l'utilisation du paradigme STF, car c'est celui qui est le plus couramment utilisé avec StarPU.

Les tâches sont soumises de manière explicite au runtime avec les données sur lesquelles elles travaillent. Le mode d'accès à chaque donnée est précisé. Via une analyse des données, StarPU peut détecter automatiquement les dépendances entre les tâches et construire le DAG correspondant. Une fois qu'une tâche a été soumise, ses dépendances sont "surveillées". La tâche est ordonnancée dès que ses dépendances sont satisfaites. StarPU prend aussi soin de déplacer les données vers l'unité de calcul visée.

5. http ://starpu.gforge.inria.fr
6. http ://www.inria.fr/equipes/storm

Les tâches sont soumises depuis l'application en utilisant un *thread maître* et via l'appel à des fonctions. L'exécution des tâches est ensuite effectuée sur des *workers* qui peuvent être de types différents (CPUs, GPUs). Un worker CPU est lié uniquement à un CPU tandis qu'un worker GPU utilise à la fois un CPU et un GPU. Pour une tâche donnée, le mode d'accès aux données doit être précisé avant la soumission, car c'est une information primordiale pour le calcul des dépendances entre tâches. En pratique, une tâche est l'association d'un *codelet* et de données. Le codelet spécifie entre autre les unités de calcul capables d'exécuter la tâche, les implémentations possibles (au moins une par type d'unité de calcul capable d'exécuter la tâche), le nombre de données et leurs modes d'accès. Ces données sont déclarées à StarPU via l'utilisation d'une fonction spécifique qui la décrit plus précisément (emplacement initial, taille, ...). Une fois cette fonction appelée, StarPU fournit un *handle* qui sera le moyen pour le programmeur d'accéder à la donnée.

Parmi les nombreuses fonctionnalités de StarPU, nous utiliserons principalement celles décrites ci-dessous. Une instance de StarPU est exécutée par processus.

3.2.3.1 Ordonnanceur

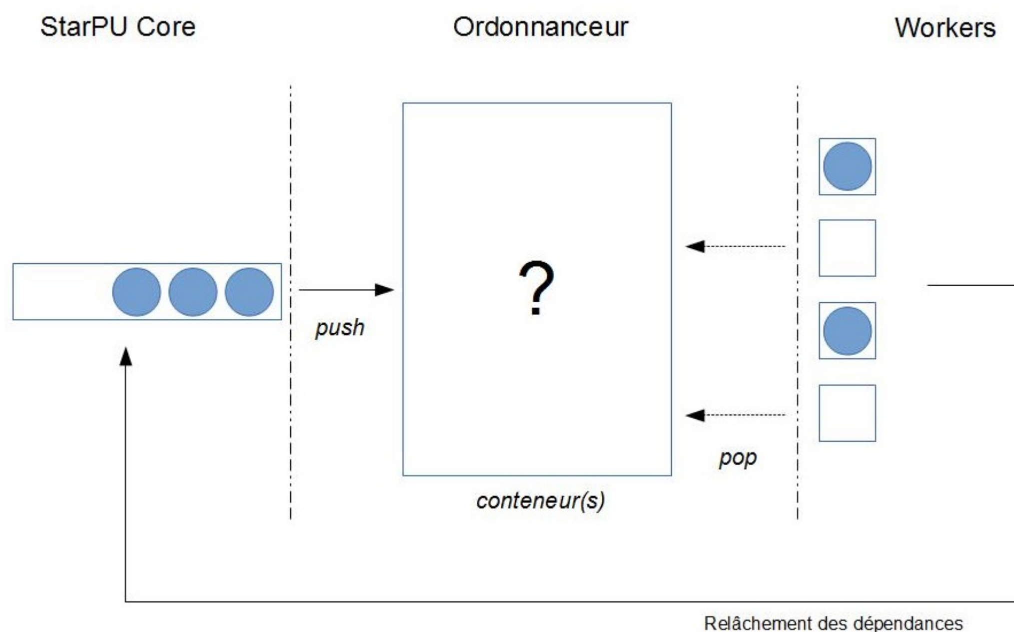


FIGURE 3.3 – Ordonnanceur dans StarPU.

StarPU fournit une API pour développer des politiques d'ordonnancement. Il suffit d'implémenter les opérations "*push*" et "*pop*" et d'avoir un ou plusieurs *conteneurs* pour stocker les tâches prêtes.

La Figure 3.3 illustre ce fonctionnement : les tâches sont représentés par des cercles bleus. Quand les dépendances d'une tâche sont résolues, la tâche est dite prête et l'opération "*push*" est effectuée par le runtime, ce qui place alors la tâche dans un conteneur. De leurs côtés, lorsqu'ils ne sont pas en train d'exécuter une tâche, les workers appellent la fonction "*pop*" et prennent une tâche prête qui provient d'un conteneur. Une fois son exécution terminée, l'information est transmise à StarPU.

En définissant des conteneurs et la manière selon laquelle ils sont accédés (via *push* et *pop*), le développeur peut avoir un réel impact sur la politique d'ordonnancement. Des indices peuvent être passés à l'ordonnanceur pour choisir un conteneur spécifique dans lequel placer la tâche. Par exemple, des priorités pour les tâches peuvent être indiquées. Dans le cas où une machine hétérogène est utilisée, un modèle de performance peut aussi être fourni : il donnera ainsi une information de temps de calcul estimé pour une unité de calcul si elle venait à exécuter une tâche donnée. Cette fonctionnalité permet d'ordonnancer des tâches sur CPUs et/ou sur GPUs.

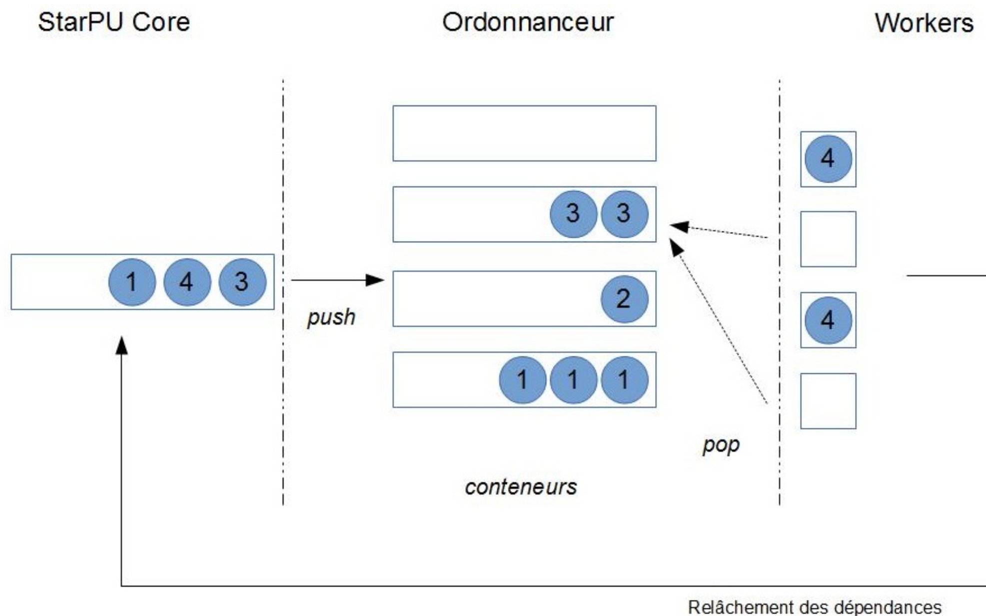


Figure 3.4 – Ordonnanceur avec une politique "prio" dans StarPU.

Dans notre étude, nous utilisons la politique "prio" qui consiste en un ordonnanceur disposant de plusieurs conteneurs, un par priorité. Les tâches sont placées dans le conteneur correspondant à leur priorité. Les workers récupèrent leurs tâches en sollicitant les conteneurs par ordre de priorité décroissante. La Figure 3.4 illustre ce mécanisme pour un ordonnanceur "prio" gérant des priorités allant de 1 à 4.

3.2.3.2 Contexte

La notion de contexte d'ordonnancement a récemment été introduite dans StarPU [HGNW13] dans le but de permettre la composition de codes parallèles. Lorsqu'un code utilise des bibliothèques parallèles de manière concurrente, l'utilisation des ressources peut être sous-optimale. Les bibliothèques parallèles étant développées dans le but de tirer parti de l'ensemble d'une machine, l'utilisation concurrente de plusieurs d'entre elles peut mener à un nombre de threads supérieur au nombre d'unités de calcul. Utilisés au sein de StarPU, les contextes permettent de cloisonner l'exécution de certaines tâches à un sous-ensemble de ressources. Un contexte détient une ou plusieurs unités de calcul et StarPU fournit une interface pour leur redimensionnement, ce qui peut se faire de manière explicite par le programmeur ou de manière dynamique via l'utilisation d'un superviseur.

Il est aussi possible d'utiliser les contextes pour tirer parti d'un autre runtime, par exemple OpenMP. Ces contextes un peu particulier (qu'on appelle *contexte-worker*) permettent l'exécu-

tion de tâches parallèles écrites pour le runtime cible [CGH+15]. Il est donc possible de soumettre des tâches écrites en OpenMP grâce à ce genre de contexte pour les voir s'exécuter de manière parallèle sur un ensemble de ressources prédéfini.

3.2.3.3 Utilisation conjointe de StarPU et de MPI

StarPU fournit deux manières intégrées d'utiliser MPI via l'extension StarPU-MPI. Il est aussi possible d'utiliser StarPU et MPI séparément.

La première manière d'utiliser conjointement StarPU via MPI consiste en l'utilisation de communications explicites intégrées à une DSM. Le développeur fait appel à *starpu_mpi_isend_detached*/*starpu_mpi_irecv_detached* et le runtime se chargera d'effectuer les opérations de manière consistante avec les autres tâches. Dans un scénario où une tâche modifie une donnée avant qu'elle ne soit envoyée, la tâche devra être terminée avant que la donnée puisse être effectivement envoyée. Avec cette solution, chaque nœud dispose d'un DAG local et les interactions avec les autres nœuds se font via ces communications explicites.

La seconde solution consiste à utiliser la fonction *starpu_mpi_task_insert*. Dans ce cas, chaque nœud doit alors insérer toutes les tâches et les transferts de données sont instanciés automatiquement par le runtime.

3.2.3.4 Profiling de code

StarPU permet d'obtenir de manière post-mortem diverses informations sur un calcul qui a été effectué. Le DAG qui a été construit peut être retrouvé, des traces peuvent être obtenues, ainsi que diverses informations comme l'activité des workers. Selon la granularité des tâches, ce profiling peut avoir un effet plus ou moins néfaste sur les performances. Pour cette raison, les tests effectifs de mesure de performance sont faits en général sans l'utilisation de cette fonctionnalité.

Les traces obtenues de StarPU sont au format Pajé [dOSdKM10]. Il est possible de les analyser via les environnements ViTE[7] ou R [IG96]. L'avantage d'utiliser ViTE réside dans la possibilité d'interagir directement sur la trace, de zoomer précisément sur certaines parties... R permet de faire ressortir des informations précises plus simplement (modification des couleurs en fonction de certains critères attachés à la tâche), mais l'interaction est limitée. Pour présenter les résultats une fois analysés, l'utilisation de R est à privilégier.

Les DAG que StarPU fournit sont au format *dot* [KN+91]. Ils sont donc exploitables via l'outil Graphviz [EGK+01]. Il est aussi possible d'utiliser Tulip [AAB+12]. Tulip est un framework permettant la visualisation de grands graphes. Il est possible d'écrire des algorithmes qui manipulent et modifient le graphe, notamment la couleur des nœuds. On peut ainsi repérer des points intéressants du DAG via un algorithme de coloriage des nœuds.

Il est aussi possible de récupérer des informations en cours de calcul sur l'activité des workers via l'utilisation de fonctions pour instrumenter certaines parties du code. Cette possibilité est moins intrusive et il donc possible de l'utiliser lors des tests de performance.

3.3 Mise en place d'une version parallèle de FLUSEPA en mémoire distribuée et utilisant StarPU

Plusieurs paramètres sont à prendre en compte lorsqu'on considère le développement d'une "version en tâches" d'un code de calcul. L'un des objectifs est de disposer du maximum possible de

7. http ://vite.gforge.inria.fr/

concurrence : il faut que la description via un graphe de tâches du problème permette l'exécution de plusieurs tâches de manière simultanée. Les données sur lesquelles le code travaille doivent permettre cette exécution concurrente.

Il faut aussi faire en sorte de pouvoir intervenir facilement sur la granularité des tâches : en effet, l'utilisation d'un runtime ne se fait pas sans surcoût et il faut pouvoir dès le départ avoir une certaine latitude sur le nombre et le coût en terme d'opérations des tâches. Une représentation trop proche de la définition du problème et n'offrant pas cette flexibilité peut ne pas être viable.

Autant que possible, il faut aussi limiter les points de synchronisation globale : un des intérêts majeur de la programmation en tâches est la définition précise des dépendances entre les différentes tâches, ce qui permet plusieurs ordres d'exécution valides des tâches. L'ajout d'un point de synchronisation limite cet possibilité en forçant l'ensemble des tâches d'une section précédant une synchronisation à être terminées avant la possibilité d'entamer la section suivante.

Ces différentes possibilités pour obtenir un cadre favorable à une parallélisation efficace peuvent parfois être contradictoires et il peut y avoir un équilibre à trouver entre elles.

Étant donné que l'on part d'un code déjà existant, il y a plusieurs autres aspects à prendre en compte. Tout d'abord, il faut définir un sous-ensemble significatif du code qui sera un bon candidat. La décomposition en tâches du problème peut imposer la réécriture de certaines portions du code et il faut s'assurer que les efforts faits dans ce sens soient pertinents. Un aspect essentiel est la définition claire de la manière d'écrire les opérations numériques de base pour le code. Dans notre cas, différents modèles physiques peuvent être utilisés et leur ajout doit rester simple pour un développeur qui n'est pas un expert de la programmation en tâches sur un runtime spécifique. Cet aspect est important et impacte la manière dont les tâches seront décrites. Il est aussi souhaitable de garder une certaine compatibilité entre la version de départ du code et la version en tâches, au moins dans un premier temps. Pouvoir développer la nouvelle version de manière incrémentale est un atout important.

3.3.1 Réalisation d'une version symbolique pour le solveur aérodynamique en mémoire partagée

Lorsqu'il s'agit de développer une application en tâches, deux méthodes se dégagent. La première consiste à partir d'une feuille blanche, de repérer les grandes lignes de l'application et de faire une application minimaliste qui pourra ensuite être enrichie. La seconde consiste à partir d'un code déjà existant et à faire les transformations nécessaires pour aboutir à un code en tâches.

Étant donné que notre objectif est de faire une version en tâches d'un code industriel, nous opterons par la suite pour la deuxième solution. Mais dans un premier temps, une version dite symbolique du solveur aérodynamique a été réalisée. Cette version ne réalise aucun calcul, mais permet d'obtenir un DAG très proche de celui qui sera réellement utilisé par la suite. Seule une itération de la partie solveur a été considérée et les cellules virtuelles (parois et conditions limites) ont été ignorées. Avant de commencer à travailler sur une vraie version en tâches, il était nécessaire d'évaluer les efforts à fournir sur différents points, notamment sur l'organisation des données en mémoire et sur la refactorisation éventuelle de certains noyaux de calcul.

3.3.1.1 Définition des entités de calcul

Dans le solveur, la source de parallélisme est essentiellement spatiale. La méthode de calcul manipule principalement des cellules et des faces ; il est donc logique de partir sur une représentation en tâches qui tire parti de cet aspect. Cependant, les valeurs pour les différentes variables

associées aux cellules et aux faces n'évoluent pas de manière indépendante : leurs valeurs dépendent des cellules et faces voisines. La quantité d'opérations au niveau d'une cellule et de ses faces est trop faible pour baser directement notre description en tâches à cette granularité là. Il est donc nécessaire de les regrouper pour avoir une granularité suffisante. L'idée de départ est donc de faire un découpage spatial en domaines géométriques pour créer ce que l'on appellera des *entités de calculs* (EC). Ensuite, comme les interactions se font entre cellules et faces, il est nécessaire de les distinguer pour mettre en évidence des tâches qui permettront un certain asynchronisme. Les regroupements de cellules et de faces seront les *composantes* dans une EC. Pour un EC de numéro i, on a une composante correspondant aux cellules C_i et une composante correspondant aux faces F_i.

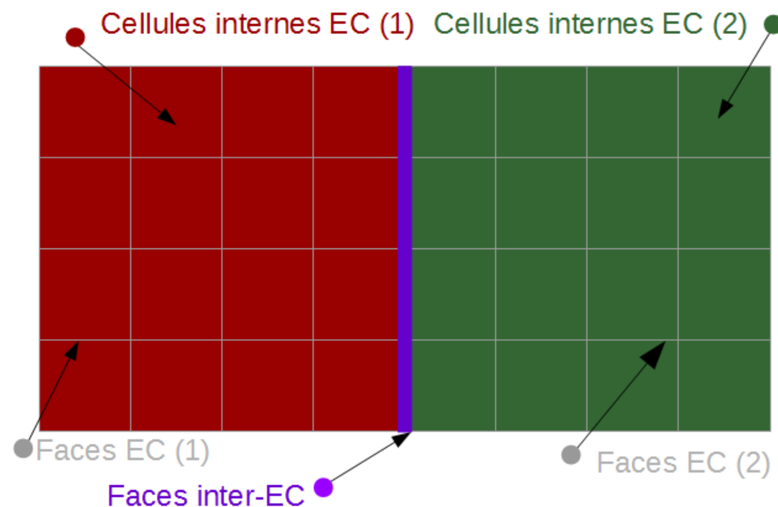


FIGURE 3.5 – Entités de calcul (EC) et composantes considérées par la version symbolique.

On se retrouve donc avec des EC (sous-domaines) dans lesquelles on distingue des composantes que l'on illustre sur la Figure 3.5. Ensuite pour chaque type de donnée manipulé par chaque composante (par exemple des “variables intensives”, des “variables extensives” pour les cellules), on définit un handle StarPU.

3.3.1.2 Génération du DAG

Le schéma d'intégration temporelle adaptatif a des particularités à prendre en compte pour la création du DAG. Pour certaines cellules, le nombre d'opérations à effectuer est beaucoup plus faible que pour d'autres. Or, ce nombre d'opérations n'est connu qu'après la classification des cellules dans les niveaux temporels. Cette répartition n'est pas prévisible de manière précise et elle est remise en cause à chaque itération générale (cf Figure 2.23). Pour pallier à ce problème, une possibilité est de générer le DAG correspondant au solveur après avoir connaissance des niveaux temporels des différentes cellules. Nous avons opté pour cette solution.

3.3.1.3 Fonctions de génération des tâches

Désormais, on sait sur quels types de donnée les tâches vont travailler et comment on souhaite les insérer. Comme on l'a vu précédemment (section 2.2.1.1), il existe plusieurs motifs d'accès aux données, motifs sur lesquels on va se baser pour générer les tâches. On peut ainsi modifier les variables de calcul :

— aux faces ;
— aux cellules ;
— aux faces en utilisant les cellules ;
— aux cellules en utilisant les faces.

Fonction de génération	#T	Composantes				
		C_i	C_j	F_i	F_j	F_{ij}
foreach_c	2	1	2			
foreach_f	3			1	2	3
foreach_ci_fi	2	1	2	1	2	
foreach_ci_cj_fij	1	1	1			1

FIGURE 3.6 – Détails des accès provoqués par les différentes fonctions de génération de tâches.

Ces différentes motifs peuvent directement servir de base pour générer les tâches. On introduit donc des fonctions de génération intitulées "foreach" et que l'on décrit sur la Figure 3.6. Les 4 fonctions de génération présentes dans la version symbolique y sont décrites selon la convention suivante. On considère 2 ECs i et j qui sont en relation entre elles. On a alors 5 composantes à considérer : les cellules de EC i (composante C_i), les cellules de EC j (composante C_j), les faces de EC i (composante F_i), les faces de EC j (composante F_j) et les faces situées entre les cellules de EC i et les cellules de EC j (composante F_{ij}). Pour chaque ligne correspondant à une fonction de génération, le numéro k d'une case composante indique que cette composante est accédée lors de la création de la tâche k. La colonne #T donne le nombre de tâches créées par la fonction de génération.

Ainsi, la ligne *foreach_ci_fi* se lit de la manière suivante : il y a 2 tâches créées par cette fonction de génération ; la première accède à C_i et F_i, la seconde à C_j et F_j.

Si un niveau temporel n'est pas présent au sein d'une EC, aucune tâche n'est ajoutée pour cette EC lorsqu'on traite ce niveau temporel.

3.3.1.4 Mise en œuvre pratique de la version symbolique

Étant donné que notre DAG dépend très fortement du cas de calcul, il était nécessaire d'obtenir des données significatives, principalement concernant deux points. Le premier est la topologie entre les différentes EC : elle doit être suffisamment réaliste. L'autre point à prendre en compte est la présence ou non de mailles des différents niveaux temporels aux sein des ECs. Nous avons rajouté une sortie au code FLUSEPA original permettant de récupérer la topologie du maillage ainsi que le niveau temporel de chaque cellule.

Cette sortie a ensuite été prise en entrée par la version symbolique. Une décomposition de domaine est calculée pour générer les ECs. Ensuite les tâches sont insérées via les fonctions de génération et le DAG final est construit.

Ainsi, nous avons pu mettre en œuvre cette version symbolique sur des cas concrets et analyser le résultat obtenu. Il est possible maintenant de modifier l'implémentation des fonctions de génération pour tester différentes manières d'insérer les tâches. Comme on n'utilise pas encore les noyaux de calcul effectifs, différentes configurations peuvent être testées facilement pour évaluer la meilleure manière de générer le DAG. Les DAG obtenus ont pu être traités via le logiciel Tulip pour obtenir des informations précises et faire des comparaisons avec des configurations réalistes. Cela nous a conduit à introduire plusieurs améliorations.

3.3.1.5 Distinction entre les cellules de bord et les cellules intérieures au sein d'une EC

Les tests menés à l'aide de la version symbolique nous ont conduit à faire le choix de séparer les cellules de bord (regroupées au sein de la composante C^B) des cellules intérieures (regroupées au sein de la composante C^I) d'une EC (cf Figure 3.7). L'ensemble des faces donnant sur des cellules d'une même EC (quelles soient cellules de bord ou intérieures) est toujours regroupé au sein de la composante F. Pour une EC i, on se retrouve donc avec les composantes suivantes : C_i^B, C_i^I, F_i et pour chaque EC j avec laquelle l'EC i est en relation, une composante F_{ij}.

La Figure 3.8 présente plusieurs ECs et les relations entre leurs composantes de type cellule. Dans ce graphe, les arêtes représentent des ensembles de faces. Un cercle large représente la composante C^I, un petit cercle représente la composante C^B. En mauve, on a les ensembles des faces entre les différentes ECs : chaque arête mauve correspond à une composante F_{ij}.

Les ensembles de faces liant les composantes C^I et C^B sont en jaune. Les arêtes en boucle représentent les ensembles de faces donnant sur des composantes cellules de même type. Pour une EC i, les arêtes en boucle et les arêtes en jaune représentent des ensembles de faces qui sont membre de la même composante F_i.

Si les ECs sont de tailles suffisantes, la proportion de calcul représentée par le bord est bien inférieure à celle représentée par l'intérieur. Cette distinction entraîne un changement dans les tâches que vont générer les fonctions de génération car plus de tâches seront générées. Ces changements sont visibles sur la Figure 3.9. La fonction *foreach_ci_fi* va générer maintenant 4 tâches au lieu de 2.

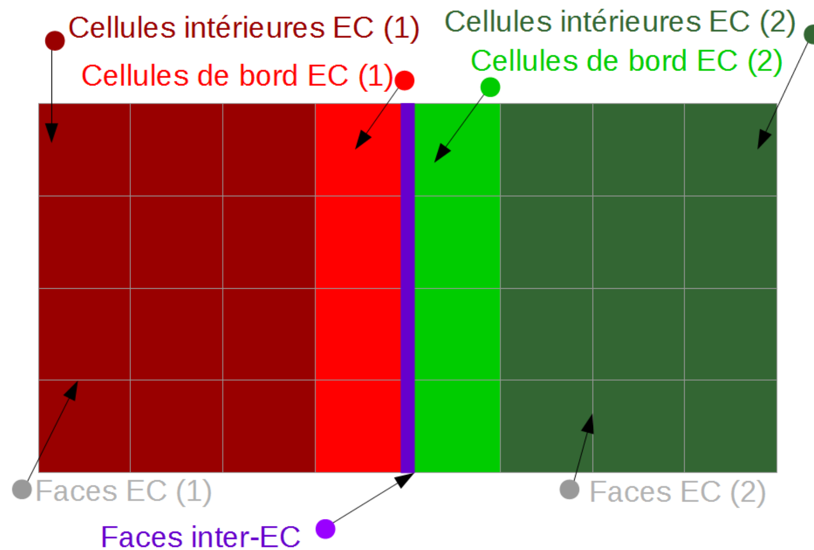


FIGURE 3.7 – Entités de calcul et composantes considérées par la version symbolique avec distinction entre les cellules de bord et les cellules intérieures.

L'objectif de cette distinction est de pouvoir parcourir le graphe plus librement. Un scénario simple d'insertion de tâches est détaillé sur la Figure 3.10 pour deux ECs qui sont en relation. On compare la version qui ne distingue pas les bords des ECs (Figure 3.5) à la version qui les distingue (Figure 3.7). Les insertions de tâches correspondant à la version qui distingue les bords des ECs sont à droite.

Un cercle représente une tâche, sa couleur représente l'EC concernée par la tâche. Le chiffre

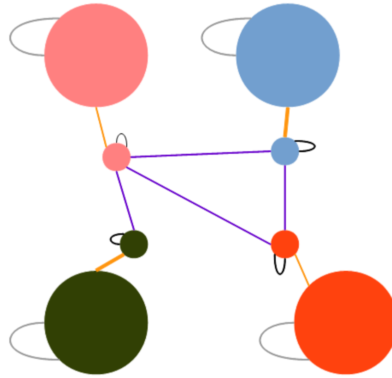


FIGURE 3.8 – Lien entre les différentes composantes.

Fonction de génération	#T	Composantes						
		C_i^B	C_i^I	C_j^B	C_j^I	F_i	F_j	F_{ij}
foreach_c	4	1	2	3	4			
foreach_f	3					1	2	3
foreach_ci_fi	4	1	2	3	4	1,2	3,4	
foreach_ci_cj_fij	1	1		1				1

FIGURE 3.9 – Détails des accès provoqués par les différentes fonctions de génération de tâches.

en bas à gauche d'une tâche représente un poids pour la tâche. On estime sur cet exemple que le poids associé au bord représente 1/10ème du poids total. Quand il y a un chiffre à l'intérieur d'un cercle, il correspond au numéro de tâche insérée comme indiqué dans les Figures 3.6 (à gauche) et 3.9 (à droite).

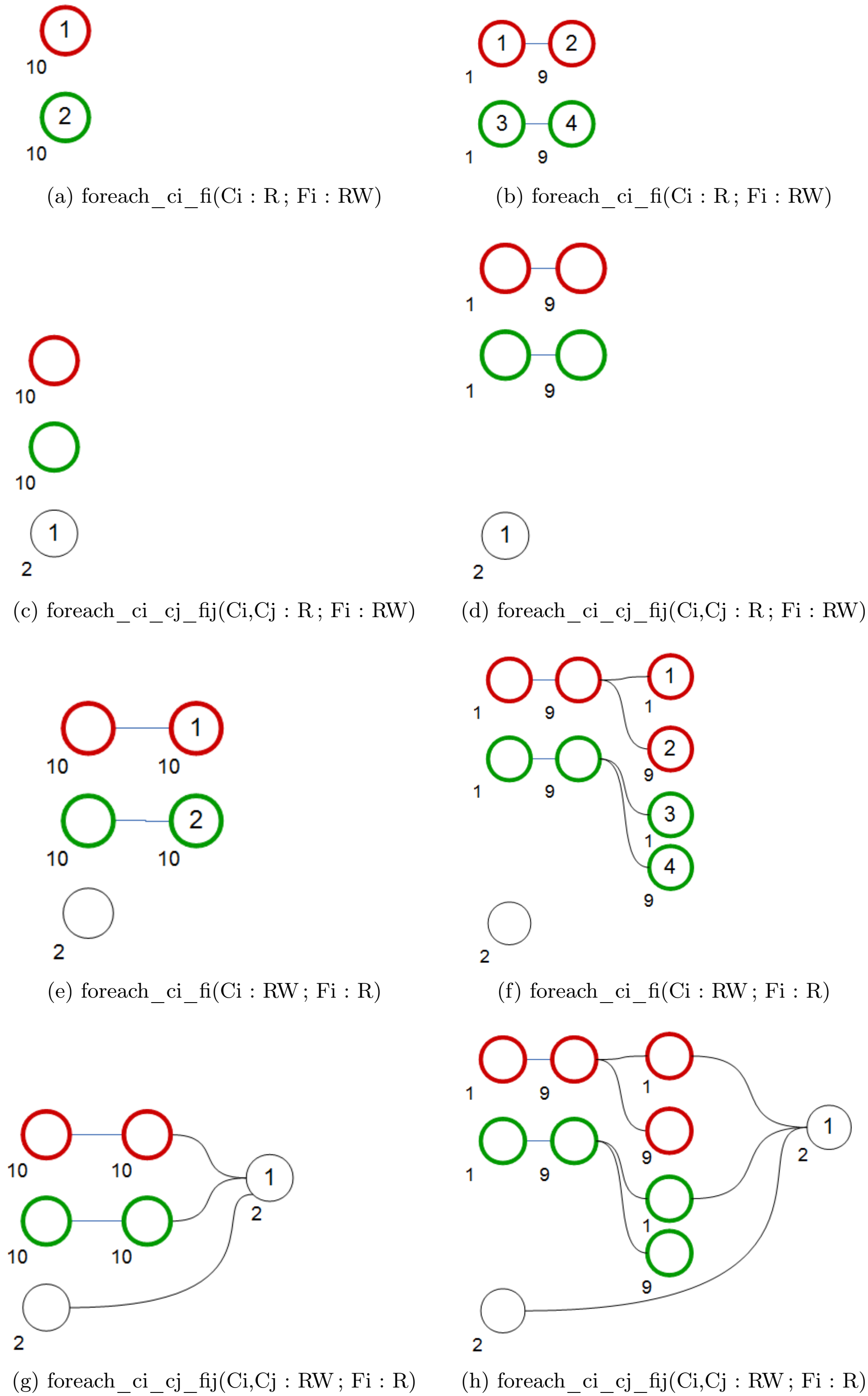
On effectue successivement :

1. un foreach_ci_fi avec les cellules en lecture (R) et les faces en lecture-écriture (RW) ;
2. un foreach_ci_cj_fij avec les cellules en lecture (R) et les faces en lecture-écriture (RW) ;
3. un foreach_ci_fi avec les cellules en lecture-écriture (RW) et les faces en lecture (R) ;
4. un foreach_ci_cj_fij avec les cellules en lecture-écriture (RW) et les faces en lecture (R).

Sur cette figure, on peut voir plusieurs choses intéressantes. Tout d'abord on peut voir que distinguer les bords à un effet qui semble néfaste après la première fonction de génération (première étape de la figure). On obtient en effet deux chaînes de deux tâches à la place de deux tâches seules. La seconde insertion est identique dans les deux cas : une nouvelle tâche indépendante de celles déjà présentes apparaît. Pour la troisième insertion, une chaîne apparaît pour la version de gauche. Pour la version de droite, les nouvelles tâches insérées sont indépendantes entre elles. Cette différence avec la première insertion, alors que la même fonction de génération est utilisée, provient du mode d'accès aux données. La quatrième fonction de génération insère une dernière tâche.

Il est alors intéressant de comparer le coût nécessaire pour atteindre cette dernière tâche. Dans le cas sans distinction des bords, il faut avoir exécuté l'ensemble de toutes les tâches insérées pour pouvoir effectuer la dernière tâche, soit un coût total de 42. Dans le cas avec distinction des bords des ECs, on arrive à un coût total de 25 : deux tâches ayant un coût unitaire de 9 ne sont pas nécessaires.

Si on compare les coûts parallèles (en considérant qu'on a au moins 3 unités de calculs), on a à gauche un coût de 22 pour terminer la dernière tâche insérée. Avec la version de droite qui distingue les bords, on arrive à un coût de 13. Cette première amélioration est donc payante : elle offre une certaine flexibilité en permettant un parcours plus libre et plus efficace du DAG.



(a) foreach_ci_fi(Ci : R ; Fi : RW)

(b) foreach_ci_fi(Ci : R ; Fi : RW)

(c) foreach_ci_cj_fij(Ci,Cj : R ; Fi : RW)

(d) foreach_ci_cj_fij(Ci,Cj : R ; Fi : RW)

(e) foreach_ci_fi(Ci : RW ; Fi : R)

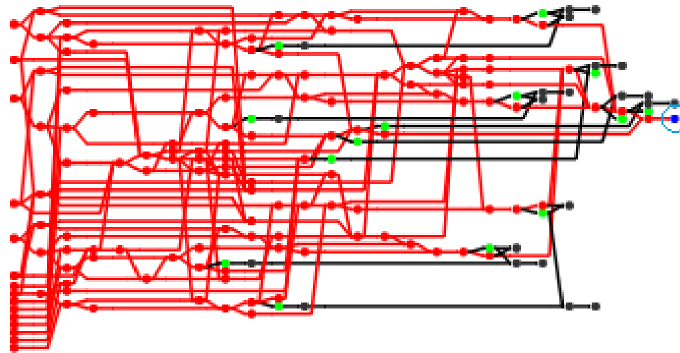
(f) foreach_ci_fi(Ci : RW ; Fi : R)

(g) foreach_ci_cj_fij(Ci,Cj : RW ; Fi : R)

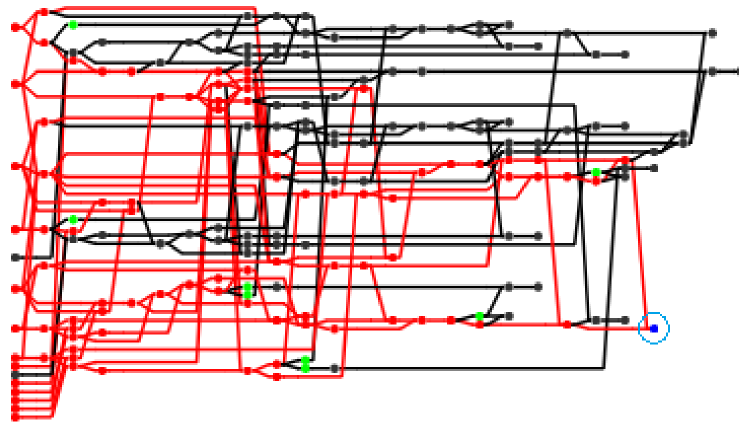
(h) foreach_ci_cj_fij(Ci,Cj : RW ; Fi : R)

FIGURE 3.10 – Scénario d'insertion des tâches : comparaison entre une version qui ne distingue pas les bords des ECs (à gauche) et une version qui les distingue (à droite).

3.3.1.6 Choix des modes d'accès pour les noyaux de calcul



(a) 1 tâche par couple (i,j)



(b) 2 tâches par couple (i,j)

FIGURE 3.11 – Comparaison de deux stratégies pour foreach_ci_cj_fij pour le même scénario d'insertion.

Comme on a pu le voir dans le paragraphe précédent, le mode d'accès qu'on utilise pour les différents paramètres des fonctions de génération de tâches a une influence non négligeable sur le DAG généré. Savoir quels modes d'accès utiliser pour les noyaux de calcul finaux du code pour un avoir un DAG "le plus intéressant" possible est donc une question importante.

Lorsqu'on utilise la fonction de génération *foreach_ci_cj_fij* pour écrire sur les cellules avec un accès en lecture pour les faces, une question intéressante est de savoir s'il est judicieux de distinguer les modes d'accès des deux composantes cellules de la fonction de génération. On compare donc :

— une version de la fonction de génération qui ne va générer qu'une tâche pour un couple (i,j) (Ci : RW ; Cj : RW ; Fij : R) ;

— et une autre version qui va insérer deux tâches avec les modes d'accès suivants (Ci : RW ; Cj : R ; Fij : R) et (Ci : R ; Cj : RW ; Fij : R).

Dans un cas on insère donc une seule tâche qui accède à deux composantes en écriture et

dans l'autre cas, on a deux tâches distinctes qui accèdent chacune à une seule des composantes en écriture.

La Figure 3.11 présente deux DAG qui comparent ces deux stratégies pour une même succession de fonctions de génération. Une certaine tâche finale est visée, la même dans les deux cas (en bleu). On colorie en rouge toutes les arêtes et tous les nœuds qui précèdent le nœud qui correspond à cette tâche dans le DAG.

Globalement, on voit qu'il y a une plus grande proportion du graphe de tâches qui doit être parcourue dans le cas où on insère une tâche par couple (i,j). Alors qu'il y a 15 tâches initiales dans les deux cas, on constate qu'il faut qu'elles soient toutes terminées dans le premier cas. Avec la seconde solution, deux de ces tâches initiales ne sont pas nécessaires. Comme on pourra le voir à l'Annexe C, l'algorithme d'intégration temporelle adaptatif conduit à des DAG très déséquilibrés et il peut être souhaitable de pouvoir traiter certaines tâches en priorité. L'insertion de deux tâches plutôt qu'une dans notre cas permet un parcours plus libre du DAG : en proportion, moins de tâches sont nécessaires pour atteindre la tâche visée.

Discussion

Notre version symbolique a permis de dégrossir les concepts de base qui feront par la suite la version finale en tâches et ce pour un temps de développement relativement court (<2 semaines). Le concept d'Entité de Calcul (EC) a notamment été introduit et évalué grâce à cette version, la distinction entre cellules de bord et cellules intérieures au sein d'une EC et l'influence sur le DAG a aussi été décidé à partir de cette version symbolique. La nécessité de passer par des fonctions de génération a été arrêté à ce moment-là.

De même, la forme que devaient avoir les noyaux de calcul était plus claire après cette version : les types de donnée utilisés (faces et cellules) et la manière de les distinguer dans les noyaux afin de générer un DAG permettant suffisamment de concurrence ont été validés grâce à cette version. Étant donné l'important travail de refactorisation impliqué, il était important d'arrêter les choix une fois pour toutes et de ne pas se lancer prématurément dans un travail qui aurait été remis en cause plus tard.

3.3.2 Préliminaires pour la mise en place d'une version en tâches en mémoire partagée

La version symbolique décrit bien les interactions entre les composantes des entités de calcul et il faut maintenant faire en sorte de disposer d'une version réalisant correctement les calculs effectifs. La version symbolique a permis d'analyser le DAG obtenu pour une itération donnée du solveur, avec des niveaux temporels fixés par des données d'entrée. La vraie version doit bien évidemment être capable d'effectuer plusieurs itérations. Les étapes précédant le solveur à proprement dit, le calcul du pas de temps et la classification des cellules dans les niveaux temporels doivent aussi être parallélisés en tâches.

Le code FLUSEPA dispose de nombreuses options (choix du modèle, choix du limiteur...), mais d'un point de vue informatique il n'y a pas de réelle différence entre les manières dont les noyaux de calcul sont écrits. Dans cette première version qui est un démonstrateur d'une version en tâches du solveur aérodynamique, nous avons décidé de nous concentrer sur un sous-ensemble des fonctionnalités du code. Ainsi, il est possible d'effectuer un calcul complet mais le choix des options est minimal. Dans la mesure du possible, il est intéressant de pouvoir développer la version "par morceau" et garder une compatibilité entre les deux implémentations (MPI+OpenMP et en tâches) est un atout important.

Cette section décrit les étapes pour parvenir à une version fonctionnelle avant de décrire les optimisations qui ont été nécessaires pour obtenir des performances effectives.

3.3.2.1 Organisation des données en mémoire

Les données doivent être placées en mémoire de manière à pouvoir générer des handles (cf. section 3.2.3). Pour un tableau donné, les valeurs correspondant à une composante d'une même entité de calcul doivent être situées de manière contigüe en mémoire. Par rapport à la version MPI+OpenMP, les données sont simplement triées. Si on applique ce tri à la version précédente, les résultats sont inchangés bien que certaines sommes ne s'effectuent plus dans le même ordre.

La plus grande modification nécessaire pour construire une version en tâches a été de faire en sorte que les noyaux de calcul soient “dans la bonne forme” pour en faire des tâches ; les observations faites avec la version symbolique ont permis de faire à ce niveau-là des choix utiles quant à la refactorisation nécessaire du code.

3.3.2.2 Refactorisation des subroutines originales

Comme les fonctions de génération des tâches utilisent des foreach, il est nécessaire que les noyaux de calcul soient écrits de manière à pouvoir être exploités par ces foreach. La plupart des subroutines existantes de FLUSEPA ne prenaient pas en compte la distinction faite sur les paramètres par les différentes fonctions de génération. Chaque subroutine a dû donc être refactorisée en plusieurs subroutines, une pour chaque fonction de génération qui sera ensuite utilisée pour la version en tâches.

Concernant les routines qui vont ensuite être utilisées pour générer des tâches via les fonctions de génération au niveau des faces, un autre travail a été nécessaire. En effet, afin de pouvoir générer des tâches qui permettent un maximum de concurrence, il a été nécessaire de distinguer les cellules situées à gauche de celles situées à droite de la face considérée. C'est notamment le cas pour les routines qui utiliseront la fonction de génération *foreach_ ci_ cj_fij* qui a été décrite précédemment. Pour traiter ce cas de figure, il a fallu doubler les paramètres qui concernaient les cellules. Afin de ne pas multiplier le nombre de noyaux de calcul à réécrire, on utilise aussi ces routines lorsqu'on utilise la fonction de génération *foreach_ ci_fi* ; on passe simplement deux fois les mêmes paramètres (une fois pour les cellules de gauche et une fois pour les cellules de droite).

Dans ce travail de refactorisation, un ensemble d'options réduit a été sélectionné pour cette première version en tâches. Ainsi, toutes les routines de FLUSEPA ne sont pas encore prêtes pour générer des tâches, mais toute la méthodologie est maintenant très claire pour le faire par la suite.

3.3.2.3 Rationalisation de l'organisation du code source et de la procédure de compilation

Le dernier point à souligner est que le code original de FLUSEPA est écrit en FORTRAN alors que StarPU est écrit en C. Pour pouvoir appeler les fonctions FORTRAN depuis le C, il est nécessaire qu'elles aient une interface si on souhaite respecter le standard et utiliser l'ISO_C_BINDING. Le moyen le plus rapide pour atteindre cela est de faire en sorte que les routines soient placées dans un module ; il faut ensuite quand même générer une entête pour l'appel de la fonction FORTRAN depuis le C. Étant donné que dans FLUSEPA une convention de nommage est utilisée, un script pour générer automatiquement les entêtes basées sur le nom des variables a été réalisé.

Dans la refactorisation du code, une étape supplémentaire a été réalisée : les sources ont été réorganisés par type de routine et CMake est désormais utilisé pour la compilation. Le fait de passer par des modules complique la réalisation d'un Makefile "à la main" alors qu'en utilisant CMake, le calcul des dépendances se fait automatiquement et très simplement.

Cette refactorisation était donc nécessaire pour la version en tâches mais ces évolutions ont aussi été portées dans la version MPI+OpenMP industrielle du code : de cette manière, pour le sous-ensemble de FLUSEPA qui est disponible dans la version en tâches, les routines qui sont utilisées par la version MPI+OpenMP actuelle sont les mêmes que celles qui sont utilisées par la version en tâches. Une simple option dans le CMake permet de compiler la version en tâches ou l'ancienne version.

3.3.3 Une version en tâches en mémoire partagée de FLUSEPA

Au début du développement de la version en tâches de FLUSEPA, la version courante de StarPU était la version 1.1.3. Au fur et à mesure de l'avancement du développement, les choix à faire nous ont poussé à passer sur la version de développement de StarPU. Parmi les évolutions de StarPU qui ont eu un impact significatif, on peut citer le changement d'algorithme pour le calcul des dépendances implicites et la possibilité d'utiliser des contextes pour bénéficier de tâches parallèles. La prise en compte de ces évolutions pendant le développement ne s'est pas faite sans difficulté à cause des quelques bugs qui ont pu être découverts et corrigés dans StarPU (principalement concernant le calcul des dépendances implicites et l'utilisation des communications explicites dans l'utilisation conjointe de StarPU et de MPI, cf. section 3.3.5).

3.3.3.1 Les entités de calcul

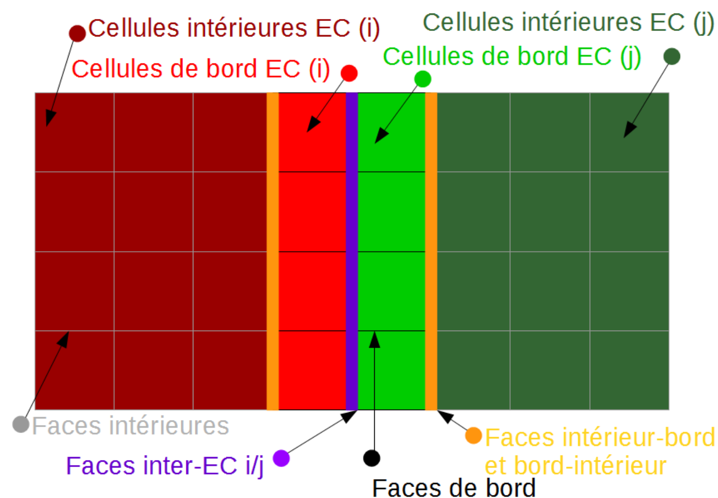


FIGURE 3.12 – Entités de calculs entre deux CEs.

La première étape concrète du développement de la version en tâches est la mise en place des entités de calcul (EC) comme défini pour la version symbolique. On a déjà décrit plus haut la réorganisation de la mémoire qui a été nécessaire ainsi que les composantes des ECs. Dans cette version en tâches, on a une composante supplémentaire pour les cellules virtuelles d'une EC. Les cellules virtuelles sont connectées via une face à une cellule réelle (cf. section 2.2.1.1) :

leur absence n'a donc pas d'impact dans la structure du DAG et c'est pour cela qu'elles n'ont pas été considérées dans la version symbolique.

Plutôt que de garder une numérotation globale comme dans la version originale, nous avons choisi une numérotation locale pour chaque type de composante dans les ECs. Si on souhaite par la suite plus de flexibilité pour la création et la destruction d'EC, disposer d'une telle numérotation locale simplifiera cette évolution.

Le fait de disposer d'une numérotation locale pour chaque composante impose une légère complexification des ECs telles qu'elles ont été décrites pour la version symbolique. La Figure 3.12 montre la nouvelle structuration. Par rapport à la version symbolique, on voit principalement une distinction des différents types de faces.

Désormais chaque EC dispose de ses propres index comme décrit dans la section 2.2.1.2. Il y a notamment plusieurs instances du tableau permettant de récupérer les relations entre faces et cellules :

— un tableau pour les faces entre cellules intérieures, nommées *faces intérieures* ;
— un tableau pour les faces entre cellules de bord, nommées *faces de bord* ;
— deux tableaux pour les faces situées entre les cellules de bord et les cellules intérieures. Il faut distinguer dans ce cas les faces qui ont une cellule de bord à gauche de celles qui ont une cellule de bord à droite. Ces tableaux sont nommés *faces bord-intérieur* et *faces intérieur-bord* ;
— pour chaque couple d'ECs i et j partageant des faces, on aura un tableau pour ces faces nommé *faces inter-EC*.

FIGURE 3.13 – Données triées et position des handles pour les cellules.

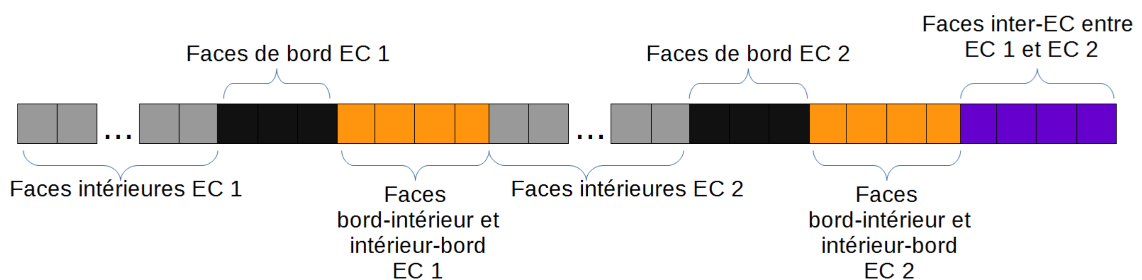


FIGURE 3.14 – Données triées et position des handles pour les faces.

Les données en mémoire sont triées de telle sorte qu'elles soient contiguës pour chaque ensemble de composantes et par EC. Cette réorganisation est nécessaire afin de pouvoir placer les handles pour StarPU. Les Figures 3.13 et 3.14 illustrent cette réorganisation pour le cas présenté sur la Figure 3.12.

```
 IHANDLE(1:24)=[C_GRAVI, C_VOL,
&               C_XMUL,C_XLBL, C_XMUT,
&               C_RO, C_GA, C_HSI, C_XN, C_T,
&               C_QR, C_QS, C_UX, C_UY, C_UZ, C_P,
&               C_GRAXN, C_GRATO, C_GRAQR, C_GRAQS,
&               C_GRAUX, C_GRAUY, C_GRAUZ, C_GRAPR]
 IHANDLE(25:35)=[F_XFACE, F_DRUG, F_DRVG, F_DRWG,
&                F_VDMXNS, F_VDMENS, F_VDMQRS, F_VDMQSS,
&                F_VDMUXS, F_VDMUYS, F_VDMUZS]

 IPARAM(1:2)=[CST1_H,CST0_H]

 CALL CE_FOREACH_CI_FI(OPAQ_CE, NUM_CE,
&                  C_LOC(PARAM), OPAQ_RUNTIME,
&                  CO_VISQFA,
&                  BO_LR, ta_one, NSIC,
&                  1, C_LOC(IPARAM),
&                  24,  C_LOC(IHANDLE), ! CELLS
&                  11, C_LOC(IHANDLE(25))) ! FACES
```

FIGURE 3.15 – Utilisation d'une fonction de génération de tâche.

3.3.3.2 Génération des tâches

Par rapport aux fonctions de génération de la version symbolique, il y a quelques opérations supplémentaires à effectuer. Une fonction de génération de tâches doit effectuer les opérations suivantes :

- récupérer l'opération à effectuer (via l'utilisation d'un codelet - cf. section 3.2.3) ;
- récupérer les index et les bornes (que l'on a présentés dans la section 2.2.1.2), car certaines opérations ne doivent être effectuées que pour un certain niveau temporel et pour certains types de données ;
- pour chaque composante des ECs concernées par la fonction de génération, récupérer les handles liés aux variables utilisées. Celles-ci sont indiquées via l'utilisation de constantes.

A cause de ces opérations supplémentaires, il existe davantage de fonctions de génération que pour la version symbolique. On a par exemple deux fonctions qui correspondent à *foreach_ ci_fi* : une va récupérer l'index qui permet de modifier les valeurs correspondant aux cellules et une autre va récupérer l'index permettant de modifier les valeurs correspondant aux faces. La Figure 3.15 montre l'appel à une fonction de génération qui insère les tâches correspondant à l'opération *VISQFA*, pour le niveau temporel *NSIC*. C'est une opération qui nécessite 24 variables liées aux cellules en lecture et 11 variables liées aux faces en lecture-écriture.

Implémentation des tâches

Afin de pouvoir utiliser les fonctions de génération, les implémentations de tâche doivent suivre un certain canevas. Dans une implémentation de tâche, la première opération est la récupération des pointeurs correspondant aux handles qui ont été passés en paramètre. Chaque fonction de génération passe les handles selon leur type dans un ordre prédéfini. Une fois ces pointeurs récupérés, il est possible d'appeler la sous-routine telle qu'elle existait auparavant. La

```
void visqfa_cpu(void * buffers[], void * cl_args)
{
  int nb_buf=0;

  //INDEX AND BOUNDS (7)
  RETRIEVE_BND_CI_FI_2C();

  //PARAM 1
  int * iscl = (int *) STARPU_VARIABLE_GET_PTR(buffers[nb_buf++]);

  //CELLS (48)
  //LEFT 24
  double * gravi_l = (double *) STARPU_VECTOR_GET_PTR(buffers[nb_buf++]);
  double * dimin_l = (double *) STARPU_VECTOR_GET_PTR(buffers[nb_buf++]);
  ...
  //RIGHT 24
  double * gravi_r = (double *) STARPU_VECTOR_GET_PTR(buffers[nb_buf++]);
  double * dimin_r = (double *) STARPU_VECTOR_GET_PTR(buffers[nb_buf++]);
  ...
  //FACE 11
  double * xface = (double *) STARPU_VECTOR_GET_PTR(buffers[nb_buf++]);
  double * drug = (double *) STARPU_VECTOR_GET_PTR(buffers[nb_buf++]);
  ...
  struct opaque opaque;
  starpu_codelet_unpack_args(cl_args, &opaque);
  int * kther = &opaque.kther;
  ...
  VISQFA(...);
}
```

FIGURE 3.16 – Exemple d'implémentation de tâche.

Figure 3.16 correspond à un extrait de code C d'une implémentation de tâche.

La fonction de génération va d'abord passer les bornes et les index : ils sont récupérés via la macro RETRIEVE_BND_CI_FI_2C. Ensuite, les paramètres sont passés dans un certain ordre : d'abord 24 paramètres correspondant aux cellules à gauche, 24 paramètres correspondant aux cellules à droite, 11 paramètres correspondant aux faces. Puis les constantes correspondant aux modèles physiques utilisés et à d'autres paramètres de calcul sont récupérés via un objet opaque ; enfin, la fonction FORTRAN originale est appelée (VISQFA).

Insertion des tâches

Algorithme 9 Insertion des tâches pour le solveur aérodynamique

1: Insertion des tâches pour le calcul du pas de temps
2: Insertion des tâches pour la classification des cellules dans les niveaux temporels
3: Attente de toutes les tâches
4: Calcul des informations pour l'ordonnanceur et l'algorithme de regroupement
5: *Boucle d'intégration temporelle adaptative :*
6: **pour** sous-itération=1 **à** 2^θ **faire**
7: Calcul de τ
8: Insertion des tâches pour le prédicteur ($\{0...\tau\}$)
9: **pour** $\tau' = \tau$ **à** 0 **faire**
10: Insertion des tâches pour le correcteur (τ')
11: **fin pour**
12: **fin pour**
13: Attente de toutes les tâches
14: Envoi d'informations au processus maître
15: Mise à jour des variables intensives
16: Attente de toutes les tâches

L'algorithme d'insertion des tâches (cf. Algorithme 9) est assez proche de l'Algorithme 2. Le calcul du pas de temps et la classification des cellules dans les différents niveaux temporels est aussi faite via des tâches (lignes 1 et 2). Certaines de ces tâches n'utilisent pas les fonctions de génération. Des informations pour l'ordonnanceur et l'algorithme de regroupement des tâches sont calculées (ligne 4), ce point sera décrit plus tard. Les tâches insérées pendant l'algorithme d'intégration temporelle adaptatif utilisent toutes les fonctions de génération.

3.3.3.3 Cohabitation de deux versions

En faisant cohabiter les structures de données servant à la précédente version MPI+OpenMP et à la version en tâches, il est possible de simplifier le développement. La manière dont sont représentées les données permet d'utiliser les mêmes tableaux pour les deux versions. Il existe cependant une exception : les tableaux multidimensionnels. Lorsqu'on transforme les tableaux multidimensionnels originaux pour qu'ils soient contigus en mémoire et pour pouvoir placer les handles, on ne tient plus compte de la définition originale du tableau. Il existe donc deux versions de ces tableaux selon la version parallèle utilisée.

La fonction *starpu_data_acquire* est une fonction bloquante qui prend en paramètre un handle et un mode d'accès (lecture / écriture / lecture-écriture). Elle permet de déclarer que les données sont utilisées par l'application en dehors des tâches. Pour les libérer, il faut utiliser la fonction *starpu_data_release*. Le principe est simple : pour passer de la représentation de

la version en tâches à l'ancienne représentation, il suffit d'attendre toutes les tâches, d'appeler *starpu_data_acquire* pour toutes les données, puis d'effectuer la conversion des tableaux multidimensionnels. Pour repasser dans la représentation de la version en tâches, il suffit de faire les opérations inverses : reconvertir les tableaux multidimensionnels, libérer les données acquises et recommencer à insérer des tâches.

D'un point de vue performance, utiliser cette cohabitation est très handicapante. L'appel à *starpu_data_acquire* est bloquant. L'intérêt principal de cette fonctionnalité est la possibilité de valider simplement une ancienne portion du code que l'on voudrait intégrer dans la version en tâches. Dans un premier temps, il suffit d'utiliser la fonction qui permet de changer de représentation, d'exécuter l'ancien code et de revenir à la représentation pour la version en tâches. Cette démarche a été utilisée de manière incrémentale et les résultats ont pu être vérifiés systématiquement à chaque étape pendant le développement.

3.3.4 Optimisation et validation expérimentale de la version en tâches en mémoire partagée

Ayant maintenant une description correcte de la version en tâches pour le solveur aérodynamique, il est possible de considérer différentes optimisations.

3.3.4.1 Réduction du coût du calcul des dépendances

Comme un grand nombre de handles est transmis à chaque codelet et que les accès aux données sont toujours les mêmes, il a été décidé de désactiver la *consistance séquentielle* pour les handles qui permettent d'accéder aux variables. Quand la consistance séquentielle est désactivée pour un handle au sein de StarPU, ce handle n'est plus considéré pour le calcul des dépendances implicites. On se basera donc sur des handles symboliques qui correspondront à chaque composante des ECs : faces d'une EC, cellules de bord, cellule internes, cellules virtuelles, faces inter-EC.

La succession des opérations dans FLUSEPA fait que ce choix ne supprime quasiment pas de parallélisme et simplifie grandement le calcul des dépendances qui désormais ne se concentre plus que sur un faible nombre d'entités symboliques plutôt que sur l'ensemble complet des handles qui sont passés à la codelet.

Cette optimisation a eu un impact important lorsque StarPU 1.1.3 a été utilisé et la Figure 3.17 montre l'influence de cette modification. A noter que depuis la version suivante de StarPU (1.1.4), la complexité du calcul des dépendances implicites a été fortement réduite, ce qui a réduit l'impact de cette modification.

Il s'agit d'un test avec un pas de temps global où on augmente le nombre d'ECs. On teste dans un premier temps la version qui prend en compte tous les handles pour le calcul des dépendances (barre de gauche notée *no_symb*), puis on refait ensuite le même test avec l'utilisation des handles symboliques pour le calcul des dépendances (barre de droite notée *symb*). L'overhead (en jaune) évolue dans des proportions très importantes dans le cas *no_symb*, au point qu'il devient plus important que le temps passé à calculer (en vert) dans le cas à 128 ECs. Dans le cas *symb*, l'overhead reste négligeable ou presque quelque soit le cas d'utilisation.

3.3.4.2 Pack de tâches

Inconvénients des fonctions de génération

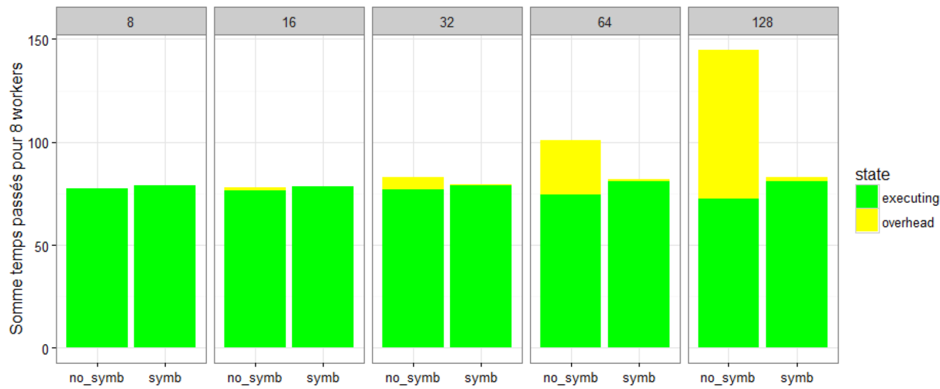


FIGURE 3.17 – Réduction du coût du calcul des dépendances via l'utilisation de handles symboliques.

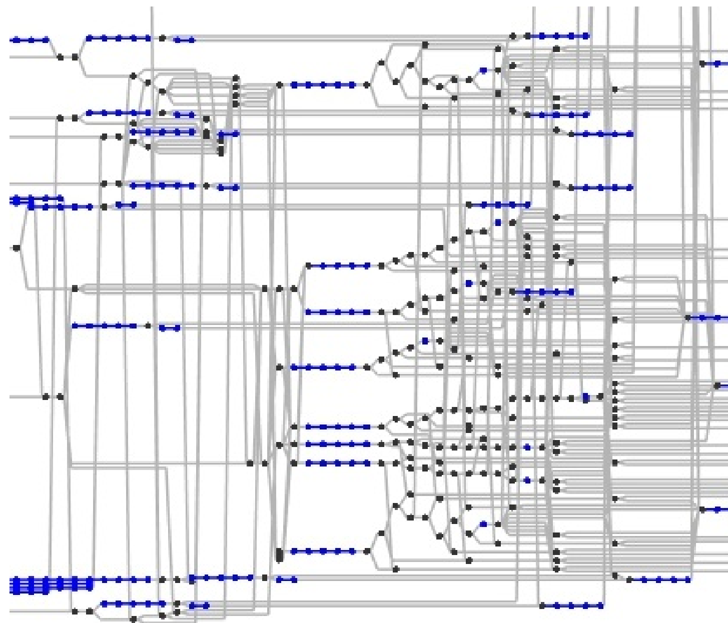


FIGURE 3.18 – Extrait de DAG où les chaînes sont mises en évidence.

La manière dont sont écrites les fonctions de génération peut générer un grand nombre de tâches. Par exemple, lorsque deux fonctions de génération du même type avec les mêmes accès s'exécutent en suivant, des *chaînes* sont générées dans le DAG. Il est cependant parfois judicieux d'avoir deux fonctions de génération similaires qui se suivent. Par exemple, on peut créer des briques élémentaires de calcul plus simples et vouloir les réutiliser ou pas selon le modèle physique qui est utilisé. Au lieu de faire des branchements au sein des noyaux de calcul, ils peuvent ainsi se faire au moment de la génération des tâches.

Une autre source de création de chaînes provient du fait qu'on a choisi de considérer souvent des noyaux de calcul simples conduisant toujours au même scénario : récupération des bornes, appel de la fonction FORTRAN existante. De fait, certaines fonctions de génération génèrent directement une chaîne.

La Figure 3.18 illustre la présence de chaînes dans le DAG. Les arêtes et les nœuds membres

de la chaîne sont détectés et coloriés en bleus. A noter que ce phénomène a été mis en évidence grâce à la version symbolique décrite en 3.3.1.

Au niveau de la réécriture des noyaux de calcul, un réel effort a aussi été réalisé afin de faire en sorte que ces noyaux n'écrivent que sur une seule composante d'une EC à chaque fois ; ce fait peut être exploité pour limiter le nombre de tâches et les dépendances associées. C'est ce que nous allons montrer maintenant.

Regroupement de tâches

L'objectif de regrouper les tâches est de simplifier le DAG sans perdre en capacité de parcours efficace. Comme nos tâches n'écrivent que sur une composante à la fois, il est possible de regrouper les tâches successives qui écrivent sur la même composante au sein de la même tâche. Ces tâches ne sont pas forcément issues des mêmes fonctions de génération. Lorsqu'on regroupe des tâches, les fonctions de génération n'insèrent plus les tâches, mais les ajoutent à un *pack de tâches*.

Le fonctionnement de l'algorithme de regroupement est relativement simple et il est modélisé par un automate à deux états : soit on est en train de regrouper des tâches qui écrivent sur les composantes liées aux faces, soit on est en train de regrouper les tâches qui écrivent sur les composantes liées aux cellules. Quand l'automate change d'état, on insère les packs de tâches que l'on vient de construire.

Les changements d'état de l'automate sont provoqués par l'utilisation des fonctions de génération. Comme on a dû faire différentes fonctions de génération selon les bornes qu'on récupère, on sait a priori sur quel type de composante on écrit.

Pour chaque EC, il existe un pack de tâches par composante : on a donc un pack pour les cellules de bord, un pack pour les cellules intérieures et un pack pour les faces.

Il existe un autre cas où il peut être intéressant de regrouper les tâches au sein d'un pack. Au sein d'une EC, quand toutes les cellules d'un niveau temporel sont situées dans une même composante de cellules intérieures, les cellules de ce niveau temporel ne seront jamais en interaction avec des cellules d'une autre EC. Donc, quand on va insérer les tâches correspondant à ce niveau temporel et à cette EC, on va insérer en fait une chaîne. Les cellules de niveau temporel τ_0 sont très peu nombreuses et sont donc peu susceptibles d'être situées aux frontières d'une EC. En plus, elles vont générer un très grand nombre de tâches : elles sont traitées à toutes les sous-itérations et une fois sur deux, on va les traiter seules (cf. Figure 2.1). On insère donc de nombreuses tâches qui forment des chaînes et qui sont en plus très peu coûteuses. Cette situation est défavorable car elle génère un overhead très important pour le runtime.

Il existera donc un deuxième cas de regroupement de tâches qu'on appellera *gros pack*. Quand les cellules du niveau temporel traité pour une EC sont toutes incluses dans la composante cellules intérieures d'une EC, toutes les autres tâches en rapport avec elles sont incluses dans le même pack jusqu'au moment où on traitera un autre niveau temporel.

La Figure 3.19 donne l'allure d'un DAG une fois qu'on a effectué toutes ces optimisations. On utilise 6 ECs et un niveau temporel maximum $\theta = 2$. Il y a donc 3 niveaux temporels et 4 sous-itérations. Les couleurs représentent les différents ECs. L'EC **rouge** contient des cellules des 3 niveaux temporels. Deux autres ECs (la **bleue** et la **mauve**) ne contiennent que des cellules de niveaux τ_1 et τ_2. Les trois dernières ECs ne contiennent que des cellules de niveau τ_2.

Les losanges colorés correspondent aux calculs qui modifient les données pour les faces d'une EC. Les losanges noirs sont pour les faces inter-EC, les petits cercles pour les cellules de bord, les grands cercles pour les cellules intérieures.

Les triangles représentent les *gros packs* : dans ce cas, chaque *gros pack* contient plus de 10

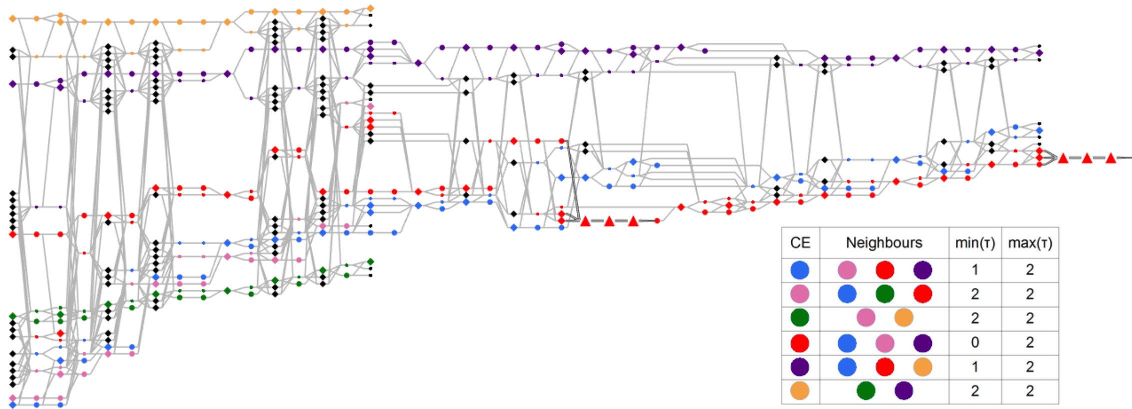


FIGURE 3.19 – DAG pour un calcul avec 6 CEs et $\theta = 2$.

sous-tâches. Pour ce DAG, on constate que les cellules de classe τ_0 ne sont présentes que dans la composante intérieure de l'EC rouge.

L'Annexe B contient des tableaux reprenant des informations sur la création du DAG pour différents cas de calcul. Le nombre de packs insérés et le nombre de tâches élémentaires ajoutées dans les différents packs sont indiqués ainsi que le temps nécessaire pour soumettre l'ensemble du DAG.

3.3.4.3 Ordonnancement des tâches

Dans la section 3.2.3.1, on a présenté de manière générale les règles d'ordonnancement de StarPU et l'ordonnanceur qu'on va utiliser principalement dans notre cas.

Si on considère l'ordonnanceur "prio", les priorités que le développeur va donner aux tâches sont d'une grande importance. En mémoire partagée, notre problème principal provient de la fin de l'itération générale : il peut arriver que les tâches qui sont prêtes ne permettent pas de donner du travail à tous les workers. En effet, les dépendances entre les tâches peuvent faire que le nombre de tâches prêtes à un instant donné soit inférieur au nombre de workers. Ils sont alors en situation potentielle de famine.

En utilisant des priorités adéquates, il est possible de limiter ce problème. Les DAG générés par l'intégration temporelle adaptative ont la particularité d'être très déséquilibrés. Un parcours qui prend en compte ce déséquilibre peut être judicieux. On souhaite implémenter une stratégie qui permette de traiter les tâches situées sur les branches les plus longues en priorité (comme par exemple celle de couleur rouge sur le DAG de la Figure 3.19.)

Dans StarPU, on peut donner à chaque tâche une priorité. Cependant les priorités ne se propagent pas : ce n'est pas parce qu'une tâche à une très grande priorité que les tâches qui la précèdent auront aussi une grande priorité. Se contenter de donner une forte priorité aux tâches traitant les faibles niveaux temporels n'est donc pas suffisant.

Pour atteindre notre but, on va donc procéder différemment. On va marquer les ECs qui contiennent des cellules de faibles niveaux temporels et leur donner la priorité maximale. Cette évolution présente déjà une amélioration, mais il est possible de faire mieux. Les dépendances entre ECs sont locales, donc on va attribuer une priorité à chaque EC en fonction de sa distance par rapport à une EC marquée. Plus cette distance est petite, plus la priorité sera importante.

Les Figures 3.20 et 3.21 montrent l'impact de l'utilisation de cette stratégie de priorité pour un cas à 128 ECs et $\theta = 4$. L'*overhead* est en jaune, le temps *idle* est en **rouge**. Les tâches

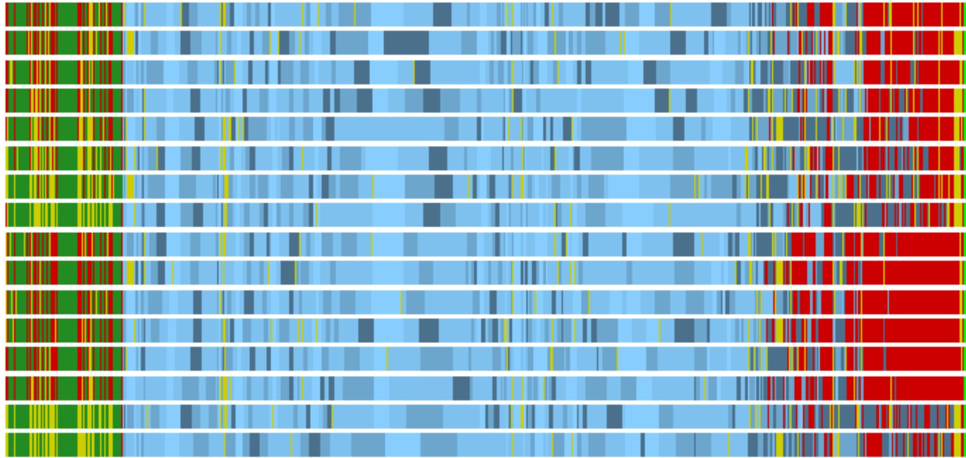


FIGURE 3.20 – Trace sans utilisation des priorités (128 ECs, $\theta = 4$: t=15.565s).

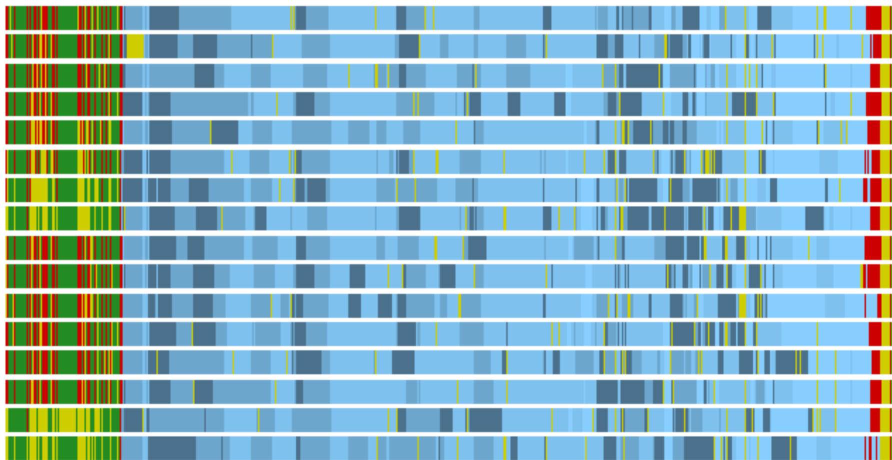


FIGURE 3.21 – Trace avec utilisation des priorités (128 ECs, $\theta = 4$: t=14.184s).

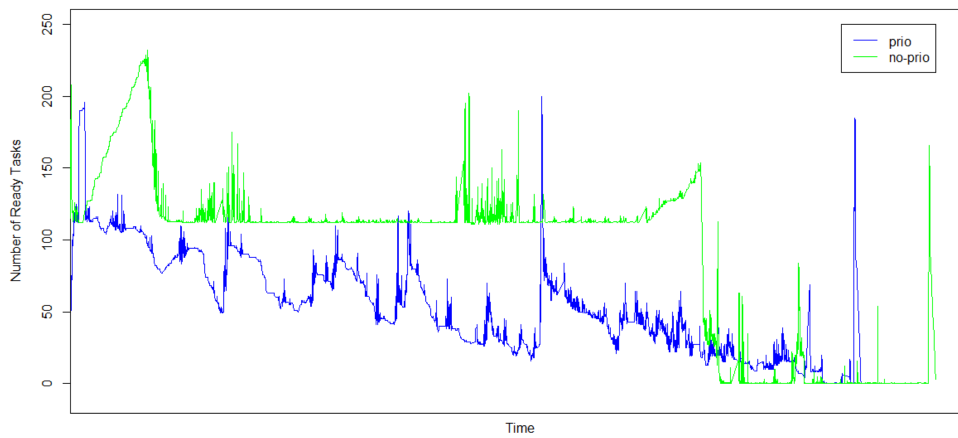


FIGURE 3.22 – Évolution du nombre de tâches prêtes en fonction de la stratégie de priorité.

correspondant aux opérations avant le solveur (lignes 1 à 3 de l'Algorithme 9) sont en **vert** (partie la plus à gauche). Les tâches correspondant au solveur sont en **bleu** (partie centrale). Les tâches correspondant à la dernière étape du calcul (ligne 15 de l'Algorithme 9) sont en **vert fluo** (partie la plus à droite). Concernant les tâches du solveur, elles sont coloriées en fonction de la priorité qu'on leur a donnée dans le calcul avec priorité. Les ECs contenant des cellules de niveaux temporels τ_1 ou τ_0 ont été marquées et ont reçu la plus haute priorité. Les autres ECs ont reçu une priorité en fonction de la distance à une EC marquée. Il y a quatre priorités : du **bleu foncé** (tâches à la priorité la plus haute) au **bleu clair** (tâches à la priorité la plus basse).

Dans les deux cas, le même DAG est construit et seule l'ordre des calculs change quand on utilise la stratégie avec priorités. Dans la trace pour laquelle on ne prend pas en compte les priorités, on observe à la fin une grande période de famine. Les workers passent du temps *idle* car ils n'ont pas de tâches prêtes à consommer. Pour le cas avec utilisation des priorités, la famine est beaucoup moins importante : le DAG est parcouru dans un ordre qui permet de garder des tâches prêtes jusqu'à la fin. On remarque qu'on arrive à terminer l'ordonnancement du calcul du solveur par des tâches de faibles priorités.

La Figure 3.22 montre l'évolution du nombre de tâches prêtes en fonction du temps. Les tâches prêtes sont celles pour lesquelles les dépendances sont réalisées et qui sont donc disponibles pour les workers. Les deux pics qu'on peut voir à droite proviennent des dernières tâches qu'on insère après le solveur (ligne 15 de l'Algorithme 9) : après avoir attendu toutes les tâches de calcul, on insère une dernière tâche pour toutes les composantes cellules des ECs car on effectue quelques opérations qui ne sont pas faites via des tâches dont une communication avec le processus maître.

On remarque que parcourir le DAG sans la stratégie d'ajout de priorités (no-prio en vert) débloque plus de tâches au départ, mais à la fin, il n'y a plus assez de tâches pour nourrir les workers. En utilisant les priorités (prio en bleu), on est vraiment capable de garder des tâches disponibles jusqu'à la fin. La stratégie avec priorités engendre un gain de temps d'exécution de près de 10% sur cet exemple : 14,184 s contre 15,565s sans priorité.

3.3.4.4 Utilisation de tâches parallèles

L'utilisation de tâches parallèles peut être intéressante car les utiliser permet de réduire le nombre de workers. Le problème de famine qui a été illustré dans la partie ordonnancement pourrait aussi être réglé au moins potentiellement en utilisant des tâches parallèles : s'il n'y a qu'un seul worker pour traiter la fin du DAG, il ne peut pas y avoir de problème de famine.

L'intérêt qu'on peut y voir est donc double car le coût de création du DAG lui-même n'est pas négligeable et parce que l'utilisation des tâches parallèles permettrait de nourrir l'ensemble des workers avec un DAG contenant moins de tâches (chaque tâche étant alors traitée elle-même par plusieurs unités de calcul). Dans notre cas, on dispose déjà d'une implémentation OpenMP des noyaux de calcul provenant de la version précédente. Il est donc judicieux de pouvoir les utiliser directement.

Utilisation de tâches parallèles OpenMP dans StarPU

L'utilisation des contextes (cf. section 3.2.3.2) et d'OpenMP n'est pas la seule possibilité pour faire des tâches parallèles dans StarPU. Son introduction est récente comparée aux autres méthodes proposées plus anciennes.

Dans StarPU, outre les contextes, il est possible de définir des tâches parallèles qui peuvent être de deux types : STARPU_SPMD ou STARPU_FORKJOIN. Ces tâches parallèles utilisent

ce qu'on appelle des *workers combinés* et nécessitent l'utilisation d'une stratégie d'ordonnancement qui prend en compte la présence de tâches parallèles. Avec ces approches, c'est l'ordonnanceur qui prend la décision du nombre d'unités de calcul utilisées par la tâche parallèle. Il est possible d'utiliser un runtime externe (par exemple OpenMP) dans le cas de tâche de type STARPU_FORKJOIN.

Il n'y a pas à notre connaissance d'autre utilisation de ces tâches parallèles, si on excepte les exemples fournis avec StarPU. Par contre, il y a actuellement d'autres travaux utilisant les tâches parallèles basés sur les contextes [CGH+15].

L'utilisation de *contexte-worker* (cf. section 3.2.3.2) est différente. Un contexte est un ensemble de ressources de calcul avec une stratégie d'ordonnancement dans lequel des tâches sont soumises. Les contextes peuvent être imbriqués. Quand un contexte n'a pas de stratégie d'ordonnancement, il est vu comme un contexte-worker : quand une tâche est placée dans un tel contexte, elle est supposéee être exécutée sur toutes les ressources de ce contexte.

Pour utiliser du code OpenMP dans un contexte-worker, il faut utiliser une tâche d'initialisation spéciale qui va lier les threads OpenMP aux unités de calcul contenues dans le contexte. Dans notre cas, on veut principalement utiliser les contexte-workers pour limiter le nombre de workers. Pour le moment, on ne cherche pas a priori à avoir des workers de tailles différentes et on ne cherche pas non plus à les redimensionner à la volée.

Application à FLUSEPA

Dans notre implémentation, on dimensionne les workers à l'initialisation, une fois pour toute et à l'aide de variables d'environnement. Il y a 3 cas possibles :

- il n'y a pas d'utilisation des tâches parallèles et donc pas d'utilisation de contexte-worker ;
- on souhaite mettre en place un seul worker pour l'ensemble du nœud de calcul : dans ce cas, on crée un unique contexte-worker et toutes les tâches sont soumises à ce contexte ;
- on souhaite disposer de plusieurs workers parallèles : dans ce cas, on crée un contexte avec une politique d'ordonnancement. Ce contexte contient alors le nombre de contexte-workers que l'on souhaite. Les tâches sont soumises dans le contexte qui contient tous les contexte-workers et elles sont distribuées entre les contexte-workers en accord avec la politique d'ordonnancement.

On introduit la notion de *configuration de calcul* comme étant la grandeur (nombre de contexte-worker $\times$ taille d'un contexte-worker) où la taille est ici le nombre de cœurs de calcul pour un contexte-worker.

La section suivante présente des expériences qui testent principalement l'utilisation des tâches parallèles OpenMP et l'influence des règles de priorité. Les autres optimisations présentées auparavant sont utilisées systématiquement car elles ont toujours un effet bénéfique.

3.3.4.5 Étude expérimentale

Dans cette étude expérimentale, on fixe $\theta = 4$ et on teste diverses configurations. Parmi les optimisations précédentes, la réduction du coût de calcul des dépendances et le regroupement de tâches seront donc systématiquement utilisés. On va se concentrer sur l'impact de l'utilisation des priorités et sur l'impact de l'utilisation des tâches parallèles.

Le cas de calcul utilisé est le même que dans la section 2.3.1 (calcul d'une onde de souffle, 11M de cellules). Une reprise du même calcul est effectuée pour tous les tests et on compare les performances pour une itération générale donnée.

Influence de la stratégie de priorité

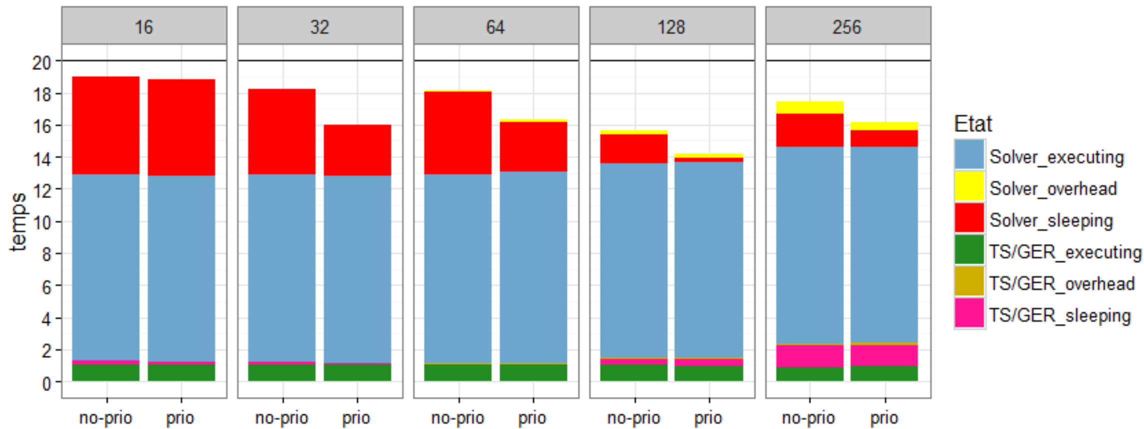


FIGURE 3.23 – Influence de la stratégie de priorité avec une configuration (16×1).

La Figure 3.23 concerne une expérience sans utilisation de tâches parallèles, en utilisant (prio) ou pas (no-prio) la stratégie de priorité décrite dans la section 3.3.4.3. La barre de référence (située en haut à 20.02 s) correspond à un calcul avec 1 EC et en utilisant une configuration de calcul (1×16), soit un contexte-worker de taille 16, ce qui correspondrait à la configuration la plus proche de l'ancienne version en mémoire partagée utilisant les 16 cœurs d'un même nœud. La première observation est que cette configuration de référence est la pire en terme de performance.

Pour un nombre d'ECs qui varie maintenant de 16 à 256, on compare le temps écoulé avec la stratégie de priorité (à droite) ou sans elle (à gauche) et on montre une synthèse de l'état des différents workers au cours d'une itération. La taille globale de la barre indique le temps (en secondes) nécessaire pour terminer une itération. Les couleurs de remplissage correspondent aux proportions passées dans chaque état (*executing*, *sleeping*, *overhead*). L'état *overhead* contient, entre autre chose, le temps nécessaire pour calculer les dépendances et insérer les tâches. L'état *sleeping* signifie que le worker est prêt, mais qu'il n'y a pas de tâches prêtes.

La partie TS/GER correspond aux lignes 1 à 3 de l'Algorithme 9 en incluant l'étape d'attente de toutes les tâches. La partie solveur correspond aux lignes 4 à 15. On observe que le temps passé à calculer effectivement (en **bleu** et en **vert**) n'évolue pas quand on augmente le nombre d'ECs ou selon que l'on utilise ou pas la stratégie de priorité, ce qui est bien. Concernant les tests allant de 16 à 128 ECs, l'état *sleeping* (en **rouge** et en **rose**) est réduit lorsqu'on augmente le nombre d'ECs et lorsqu'on utilise la stratégie de priorité. Cette stratégie est bénéfique dès qu'on utilise 32 ECs. Concernant l'état *overhead* (en **jaune** et en **marron**), il évolue de manière linéaire avec le nombre d'ECs et la stratégie de priorité n'a pas d'influence.

Le cas à 128 ECs correspond à celui qui a été présenté dans la section 3.3.4.3. Dans ce cas, on arrive presque à réduire à zéro le temps passé dans l'état *sleeping* grâce à la stratégie de priorité et c'est la situation où on obtient la meilleure performance globale.

Lorsqu'on utilise 256 ECs, on remarque que le temps passé dans l'état *sleeping* augmente, à la fois lorsqu'on est dans la partie TS/GER et dans la partie solveur. Ceci est dû au temps nécessaire pour insérer les tâches. La Figure 3.24 indique le temps d'insertion des tâches dans la partie solveur (lignes 5 à 12 de l'Algorithme 9) pour l'ensemble des 16 sous-itérations. Pour le cas à 256 ECs, il faut 13.6 secondes pour insérer toutes les tâches et le temps CPU total

#CE	Temps d'insertion des tâches	Temps CPU (partie solveur)
16	0.79 s	184.5 s
32	1.66 s	187.3 s
64	3.25 s	190.4 s
128	6.99 s	195.5 s
256	13.57 s	195.8 s

FIGURE 3.24 – Temps d'insertion des tâches et temps CPU de la partie solveur en fonction du nombre d'ECs pour les 16 sous-itérations.

dans cette section est de 195.8 s (soit 12.2 secondes par worker en moyenne). On constate donc qu'augmenter le nombre d'ECs permet d'avoir plus d'opérations concurrentes, mais avec 256 ECs, le coût d'insertion des tâches devient trop important.

Influence de l'utilisation des tâches parallèles

Dans un premier temps, on teste l'influence des tâches parallèles sans la stratégie de priorité, puis on combinera les deux optimisations.

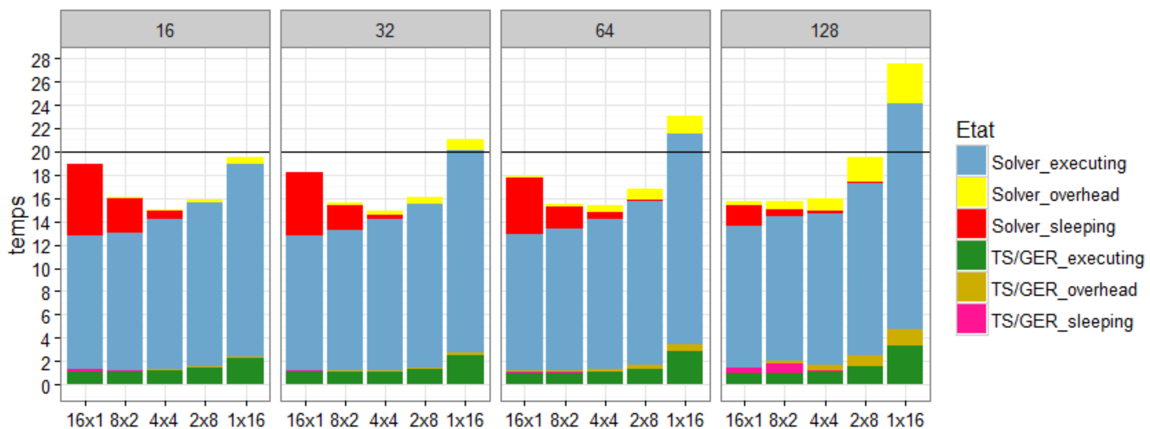


FIGURE 3.25 – Influence de l'utilisation des tâches parallèles sans stratégie de priorité.

La Figure 3.25 montre les résultats. On constate que l'utilisation de tâches parallèles permet de réduire très fortement le temps passé dans l'état *sleeping*. Ainsi dès 16 ECs et lorsqu'on utilise la configuration de calcul (2×8), le temps passés dans l'état *sleeping* devient très faible.

Cependant, on constate aussi que le temps que l'on passe à calculer effectivement augmente lorsqu'on utilise les tâches parallèles (état *executing*). Comme on a pu le voir plus tôt sur la Figure 2.21, la scalabilité de nos noyaux de calcul n'est pas parfaite. Cependant, jusqu'à la configuration (2×8), cela reste correct. Quand on utilise l'ensemble des 16 cœurs du nœud (qui sont situés sur 2 sockets différentes) à savoir la configuration (1×16), la situation est plus critique. Pour cette première batterie de tests sans utilisation de priorité, la meilleure solution est obtenue pour 32 ECs en configuration de calcul (4×4).

L'utilisation des tâches parallèles permet donc de réduire le temps passé dans l'état *sleeping*, ce qui était un effet recherché. Par contre il y a un effet négatif : à cause de la scalabilité déficiente des noyaux de calcul, le temps total passé à les exécuter augmente lorsqu'on travaille avec des

workers de taille de plus en plus grande. Lorqu'on considère le cas avec 32 ECs et que l'on passe d'une taille de worker de 1 à 2, on passe 5 % de temps en plus dans le solveur à exécuter les noyaux de calcul. De 1 à 4, l'augmentation est de 11 %, ce qui reste raisonnable. Par contre, lorsqu'on considère des tailles de worker de 8 et 16, l'augmentation est respectivement de 20 et de 45 %, ce qui est très important.

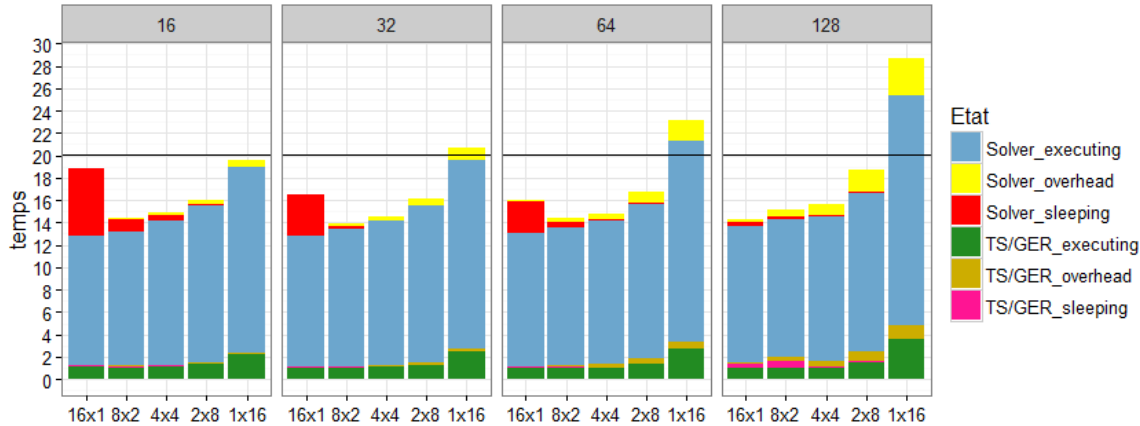


FIGURE 3.26 – Influence de l'utilisation des tâches parallèles avec utilisation des priorités.

La Figure 3.26 montre les résultats lorsqu'on combine l'utilisation des tâches parallèles avec la stratégie de priorité. On voit que l'on tire un net bénéfice à utiliser ces 2 optimisations conjointement. La configuration de calcul (8×2) devient particulièrement intéressante. On ne perd quasiment pas d'efficacité des noyaux de calcul en utilisant cette configuration et il y a suffisamment peu de workers pour pouvoir passer un temps négligeable en attente dès 32 ECs. La situation avec 32 ECs en configuration de calcul (8×2) donne les meilleurs résultats pour notre cas test. L'Annexe C contient les traces avec et sans priorité pour cette configuration de calcul, le détails de la topologie des ECs ainsi que des DAG montrant l'ordre d'exécution des tâches. Avec cette configuration, on obtient un temps de 13.9s : à ressource équivalente, on va 44 % plus vite qu'avec la version de référence (20.02s)

D'autres expériences sont détaillées dans l'Annexe B. On a vu que le coût des itérations variait en cours de calcul, les expériences sont donc faites pour deux instants distincts du calcul en faisant varier θ.

3.3.5 Mise en place d'une version distribuée

La version distribuée MPI+OpenMP se basait sur une décomposition de domaine. Pour la version en tâches en mémoire distribuée, on fait le même choix. Le problème principal auquel était confronté la première version était la rigidité de la parallélisation : malgré l'utilisation de communications asynchrones, les temps de synchronisation étaient importants. Ce problème est illustré en particulier par les traces d'exécution présentées dans la section 2.3.1 et dans l'Annexe A.

Le choix de passer sur une version à base de tâches a été motivé par le fait de pouvoir disposer d'une expression des dépendances moins stricte que dans la version MPI+OpenMP. Ainsi, la possibilité d'effectuer les calculs selon plusieurs ordres valides peut autoriser des parcours du DAG limitant le temps passé en attente pour les workers.

Lors de la mise au point de la version utilisant des tâches en mémoire partagée, nous avons eu à faire face à un problème de famine à la fin des itérations. Bien que différent du problème que l'on souhaite régler en utilisant aussi la mémoire distribuée, ce problème nous a contraint à mettre en place une stratégie de priorité des tâches. L'utilisation de cette stratégie de priorité permet de modifier l'ordre dans lequel les tâches sont exécutées, ce qui est exactement ce que l'on souhaite faire aussi dans cette nouvelle version distribuée.

FIGURE 3.27 – Trace avec utilisation des priorités, colorié en fonction des sous-itérations. $\theta = 4$, 16 sous-itérations, 128 ECs, configuration de calcul (16×1)

La trace d'exécution présentée sur la Figure 3.27 illustre un calcul effectué avec la version en tâches en mémoire partagée. Sur cette trace, le **rouge** représente à la fois l'état *overhead* et l'état *sleeping* pour les différents workers. Le **noir** est utilisé pour les tâches insérées aux lignes 1, 2 et 15 de l'Algorithme 9. Au milieu de la trace, on a donc les tâches insérées au cours de la *boucle d'intégration temporelle adaptative* (lignes 6 à 12). Ces tâches sont coloriées en fonction de la sous-itération à laquelle elles sont insérées :

- au cours de la 1ère sous-itération, elles sont en **vert foncé** ;
- au cours de la 5ème sous-itération, elles sont en **vert clair** ;
- au cours de la 9ème sous-itération, elles sont en **jaune** ;
- au cours de la 13ème sous-itération, elles sont en **rose**.

Au cours des autres sous-itérations, les calculs effectués ne sont pas suffisamment importants pour que l'on puisse les distinguer sur la trace. Le point très positif que l'on constate sur cette trace est le fait que certaines tâches insérées au cours de la 1ère sous-itération sont effectuées à la toute fin du calcul comme l'atteste la présence de vert foncé à droite de la trace. Si on compare donc cette trace d'exécution avec une trace de la version MPI+OpenMP, (par exemple celle de la Figure 2.32, qui était aussi pour un $\theta = 4$), on constate que l'ordre imposé de la version MPI+OpenMP est "cassé" par cette nouvelle version en tâches. Cette capacité à exécuter les tâches dans un ordre plus relâché, mais toujours correct, est un élément sur lequel on souhaite s'appuyer pour réduire les problèmes de synchronisation présents dans la version MPI+OpenMP.

3.3.5.1 Utilisation de communications explicites

Comme nous l'avons présenté dans la section 3.2.3.3, il existe plusieurs possibilités pour l'implémentation de la version distribuée. On peut insérer l'ensemble de toutes les tâches sur tous les

nœuds et laisser au runtime déterminer les communications. On peut aussi se baser sur des communications explicites intégrées au runtime, qui sont alors comparables à des tâches. On exclut d'emblée la possibilité d'utiliser MPI en dehors de StarPU car cette solution ne nous permettrait pas de résoudre les problèmes provenant de la version précédente ; en effet, on se retrouverait de nouveau avec les mêmes problèmes de synchronisation que ceux que l'on a rencontré avec la version MPI+OpenMP.

Dans les expériences précédentes, on a montré que les choix que nous avons faits ont un grand impact sur le coût de création du DAG. Le fait de devoir effectuer une première partie des calculs avant de pouvoir insérer toutes les tâches fait que le DAG ne peut pas être décrit explicitement dès le départ. Pour pouvoir insérer les tâches sur tous les nœuds, il faudrait donc effectuer des communications pour que tous les nœuds sachent quelles tâches insérer. Le coût de création du DAG est déjà un problème en mémoire partagée et utiliser cette stratégie là pour la version distribuée ne ferait qu'accentuer le problème.

Dans notre problème, exprimer les communications de manière explicite est relativement simple : en effet, on part d'une décomposition de domaine où la topologie est connue et les communications ne se font qu'avec les nœuds partageant une frontière. L'autre avantage est que l'on dispose d'une version MPI déjà existante et on a donc une bonne idée du schéma de communication. Nous avons donc opté pour des communications décrites de manière explicite à l'aide des fonctions *starpu_mpi_isend_detached* / *starpu_mpi_irecv_detached*.

3.3.5.2 Réalisation de la version distribuée

De la même manière que pour la version MPI+OpenMP, chaque processus se voit attribué un domaine après le partitionnement effectué par SCOTCH. A ce moment-là, les cellules de bord entre domaines sont identifiées et les faces de bord entre domaines sont dupliquées pour chaque processus. Afin de distinguer les cellules de bord d'un domaine des cellules de bord d'une EC, on note les premières *cellules de bord-MPI*.

Lors de la création des ECs, les cellules de bord-MPI sont incluses dans la composante de bord (C^B) de l'EC auxquelles elles appartiennent. Après cela, chaque processus communique avec ses voisins afin d'obtenir les informations concernant les cellules de bord communes. Pour chaque couple (EC locale, EC voisine), on met en place une composante cellule fantôme. Là aussi, comme dans la version MPI+OpenMP, les variables liées à cette composante ne seront modifiées que par des communications.

La Figure 3.28 montre ces nouveaux changements : on dispose de deux domaines de calcul. Il y a 2 ECs par domaine. Les domaines sont connectés via 2 faces. Entre l'EC verte du premier domaine et l'EC bleue du second domaine, on peut voir des faces inter-domaines. Pour ces faces inter-domaines, d'un côté il y a une cellule de bord-MPI et de l'autre côté une cellule fantôme. La cellule fantôme voit ses valeurs modifiées uniquement par des communications. Avec cette version distribuée, on a donc un type de face en plus, les faces inter-domaines, ainsi qu'un type de cellule en plus, les cellules fantômes.

Pour les communications proprement dites, on utilise pour l'envoi une tâche qui copie les cellules de bord-MPI vers une zone tampon. Ce tampon est envoyé en utilisant *starpu_isend_detached*. Pour la réception des données, la fonction *starpu_irecv_detached* est utilisée. Là aussi, on passe par une zone tampon avant que les données ne soient copiées dans la composante cellule fantôme. On utilise une zone tampon car on ne souhaite pas envoyer l'ensemble de la composante de bord d'une EC : certaines cellules de bord de cette composante ne sont pas de bord-MPI et lorsqu'on traite des cellules de faibles niveaux temporels, elles ne représentent qu'une fraction des cellules de bord-MPI.

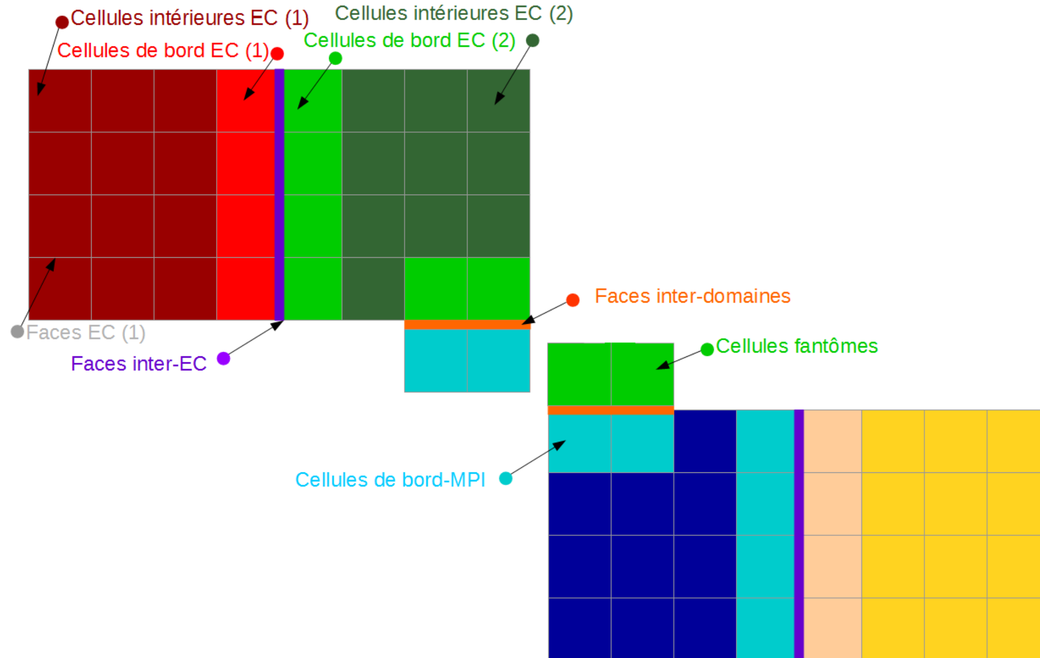


FIGURE 3.28 – 2 domaines de calcul avec 2 ECs chacun.

Avant chaque fonction de génération de tâche qui utilise les cellules en écriture, on insère nos tâches de communication, comme montré sur l'Algorithme 10. Les tâches d'envoi et de réception sont insérées en même temps.

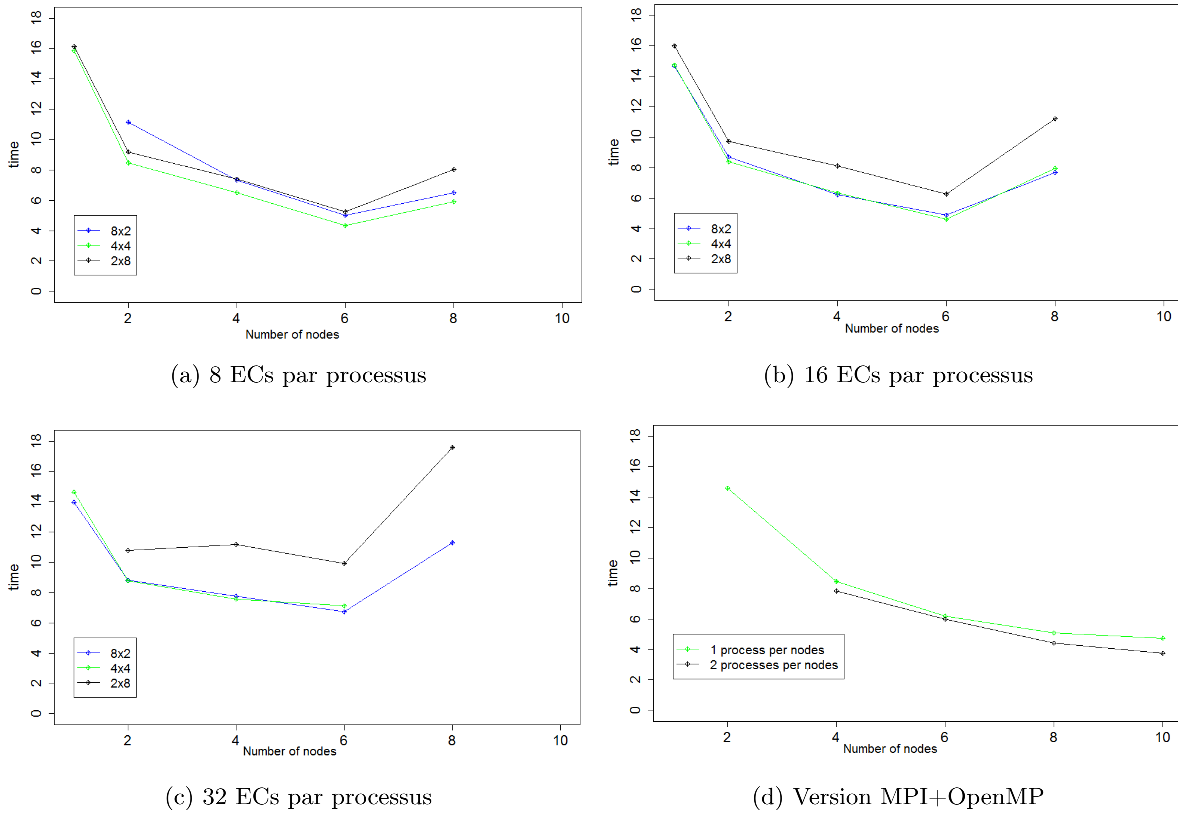
Algorithme 10 Pseudo-code avec communications

1: Insertion des tâches de communication
2: Fonction de génération de tâches prenant les cellules en lecture

Si on compare cette succession d'opérations à celle de la version précédente (Algorithme 5), elle est en fait plus simple. Cette fois, le programmeur n'a plus à distinguer les cellules de bord des cellules intérieures et il se contente d'appeler la fonction de communication juste avant les tâches qui en ont besoin. Le code est donc plus simple à écrire de ce point de vue là. Le recouvrement calcul-communication se fait via le runtime et non pas par une écriture statique prédéfinie comme dans l'Algorithme 5. Le programmeur n'a même pas à savoir quand il pourra envoyer les données : il se contente d'insérer la tâche de communication juste avant que les données soient nécessaires. La consistance séquentielle assure que les données seront envoyées après leur dernière modification et les priorités associées aux tâches peuvent être choisies de manière à maximiser le recouvrement calcul-communication.

3.3.5.3 Étude expérimentale sur le cas test de référence

Pour ce test de performance, on utilise le même cas que pour l'étude en mémoire partagée (calcul d'une onde de souffle, 11M de cellules). On considère toujours un calcul avec $\theta = 4$. On teste entre 2 et 8 processus MPI, en utilisant entre 8 et 32 ECs par processus et on évalue trois configurations de calcul (nombre de contexte-workers $\times$ taille d'un contexte-worker) à savoir 8×2, 4×4 et 2×8.



(a) 8 ECs par processus (b) 16 ECs par processus

(c) 32 ECs par processus (d) Version MPI+OpenMP

FIGURE 3.29 – Temps passé (en secondes) dans le solveur pour différentes configurations de calcul.

	TS/GER état *idle*	TS/GER état *executing*	Solveur état *idle*	Solveur état *executing*	Tsub
#1	0.338	0.197	1.440	2.356	0.608
#2	0.357	0.174	1.559	2.241	1.301
#3	0.351	0.184	1.401	2.395	0.937
#4	0.293	0.243	1.790	2.005	0.218
#5	0.310	0.216	1.599	2.205	0.599
#6	0.344	0.191	1.627	2.169	1.259

FIGURE 3.30 – Temps passé (en seconde) pour chaque processus. 6 processus sont utilisés, avec 8 ECs par processus et une configuration (4×4).

Chaque processus est en charge d'une partie du maillage et la décomposition de domaine prend en compte le coût des cellules. La Figure 3.29 présente le temps passé dans le solveur aérodynamique pour une itération et pour les différentes configurations testées.

On note que la meilleure performance (4,331s) est obtenue pour 8 ECs par processus et 6 processus configurés en 4×4. Ce cas est détaillé sur la Figure 3.30. Pour chaque processus, le temps passé dans les parties TS/GER et solveur en fonction de l'état des workers est mesurée. Le temps de soumission du DAG est indiqué dans la colonne Tsub. L'équilibrage de charge est correct avec une différence de 13% entre le processus 1 et le processus 4. Cependant, le temps

idle est important. Avec uniquement 2 ECs par worker, il n'est pas possible de bénéficier d'un recouvrement calcul-communication effectif. Avec 8 processus MPI, on ne réduit pas le temps passé dans le solveur.

La Figure 3.29d présente le temps écoulé pour la version précédente MPI+OpenMP. On considère 2 configurations, une avec un processus par nœud et l'autre avec deux processus par nœud. Malheureusement, le problème ne tient pas en mémoire avec un plus grand nombre de processus par nœud. On constate que parmi les configurations testées, la meilleur performance est obtenue pour 2 processus par nœud et avec 10 nœuds. Mais quand on considère un nombre de nœuds pour lequel la version en tâches est toujours compétitive (jusqu'à 6), la version en tâches est alors plus rapide (4,33s contre 5,98s). C'est donc un résultat très encourageant.

Étant donné qu'on reste à taille de problème fixée et qu'on augmente le nombre de nœuds de calcul, on arrive naturellement à un point où l'overhead de la parallélisation en tâches devient trop important. Lorsqu'on utilise 8 ECs par nœud et qu'on utilise 8 processus, on est déjà à 64 ECs. Plus on utilise de nœuds, plus le temps de calcul par processus diminue, alors que l'overhead reste à peu près constant. C'est donc la raison principale pour laquelle on ne peut pas utiliser plus d'ECs pour ce problème. Il est donc nécessaire d'évaluer notre version distribuée en tâches sur un cas test de taille plus importante.

3.3.5.4 Étude expérimentale sur un cas de plus grande taille

Dans cette section, nous allons étudier le comportement pour un problème plus gros et pour un plus grand nombre de nœuds de notre version en tâches ; nous allons aussi dans ce cas la comparer avec la version MPI+OpenMP.

Comme la version de notre code n'était pas prête pour traiter directement dans un cadre de production un cas de grande taille (essentiellement à cause du surcoût lié à la centralisation avec le maître), nous sommes partis d'un maillage original plus simple correspondant à un cas d'onde de souffle simplifié sans calcul d'intersection et que nous avons enrichi directement en divisant chaque cellule originale par 8. On appelle dans la suite ces cellules originales les cellules mères et les cellules qui en résultent par raffinement les cellules filles. En procédant de la sorte, il n'y a pas eu de problème de taille mémoire pour le maître car il ne manipule que le maillage original (au environ de 10M de mailles).

Pour effectuer l'équilibrage de charge, nous avons réalisé un premier calcul pour connaître les classes temporelles des différentes cellules filles pour leur donner un poids et cela s'est fait en utilisant la fonction de coût que nous avons précédemment utilisée. Chaque cellule mère reçoit ensuite pour poids la somme de ses cellules filles. Un fichier de poids a été ensuite généré et le calcul a été redémarré en utilisant ce fichier pour donner des poids à l'ensemble des cellules mères afin d'effectuer l'équilibrage (le calcul en sous domaines avec SCOTCH se fait sur le maillage original). D'un point de vue simulation physique, le maillage original étant plus simple que les maillages qui sont traités habituellement, il n'y a au plus que 4 niveaux temporels ($\theta = 3$) dans ce cas de calcul.

Pour que le calcul tienne en mémoire, il nous a fallu utiliser un nombre suffisant de nœuds (au moins 16). Cette fois-ci, un cluster disposant de 20 cœurs par nœud (avec 2 processeurs Ivybridge de 10 cœurs) a été utilisé. On teste la version MPI+OpenMP avec un ou deux processus par nœud et la version en tâches dans la configuration d'exécution 4×5 soit 4 workers StarPU de tailles 5 (5 tâches OpenMP). On effectue le calcul pour 16, 20, 24 et 28 nœuds. Pour la version MPI+OpenMP, uniquement le calcul avec un processus par nœud a pu être mené à bien avec 16 nœuds. Dans ce cas précis, en utilisant 2 processus par nœud, il n'a pas été possible de faire le calcul en tenant en mémoire.

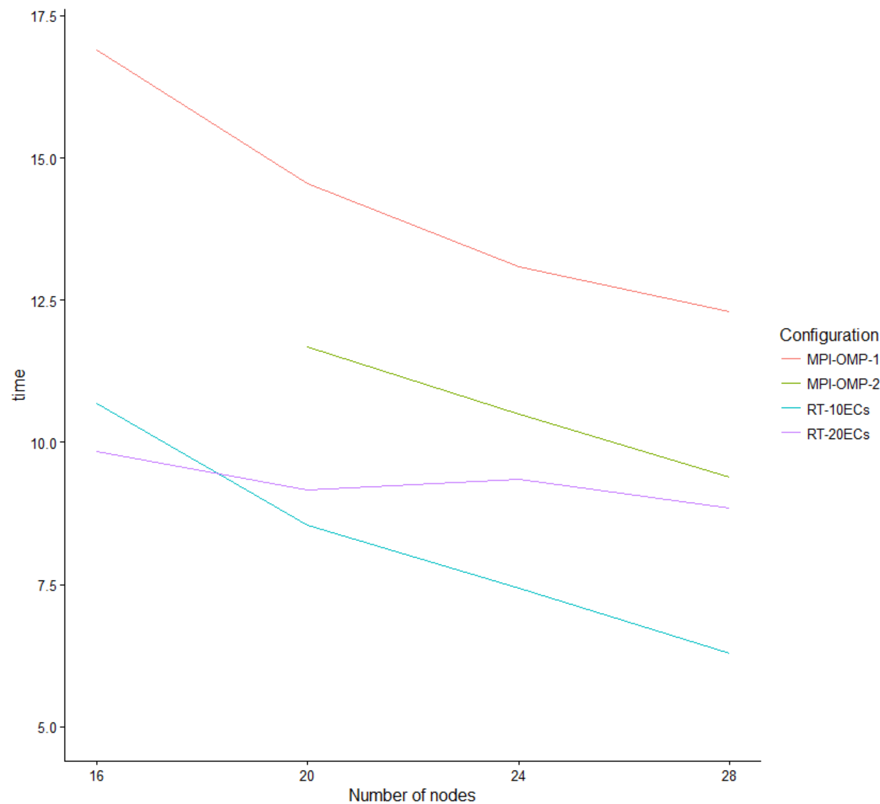


Figure 3.31 – Temps pour une itération du solveur aérodynamique.

La Figure 3.31 montre les résultats obtenus. On compare deux configurations pour la version en tâches, une utilisant 10 ECs par processus notée RT-10ECs et une autre avec 20 ECs par processus notée RT-20ECs. Pour la version MPI+OpenMP, on effectue donc le calcul avec 1 processus par nœud (noté MPI-OMP-1) ou avec 2 processus par nœud (notée MPI-OMP-2).

Pour ce qui concerne la version MPI+OpenMP, la configuration avec 2 processus par nœud est clairement la plus performante lorsqu'elle est utilisable (au delà de 16 nœuds).

La version en tâches s'exécutant sur le support d'exécution StarPU est plus performante dans tous les cas que la version MPI+OpenMP, les deux courbes correspondantes étant toujours en dessous des courbes relatives à la version MPI+OpenMP.

Avec 16 nœuds, la meilleure version en tâches (20 ECs par nœud) est 70% plus rapide que la meilleure version MPI+OpenMP (avec 1 processus par nœud). Avec 28 nœuds, ce gain est de 49%, mais cette fois-ci la meilleure version en tâches est pour 10 ECs par nœud et la meilleure version MPI+OpenMP est pour 2 processus par nœud. On constate que la version en tâches utilisant 10 ECs par nœud passe mieux à l'échelle que lorsque l'on utilise 20 ECs par nœud et cela est naturel car la granularité de calcul est de plus en plus problématique quand le nombre de nœuds augmente (le nombre de cellules par EC est à la base plus faible pour 20 ECs par nœud).

La Figure 3.32 donne le détail des différents temps passés pour les différentes configurations et pour les différents nombres de nœuds.

En ordonnée, on a le temps passé total pour une itération du solveur aérodynamique et les barres sont coloriées en fonction de la proportion passée dans les différents états pour l'ensemble des processus : en **rouge** pour le temps passé dans l'état *sleeping*, en **bleu** pour le temps passé

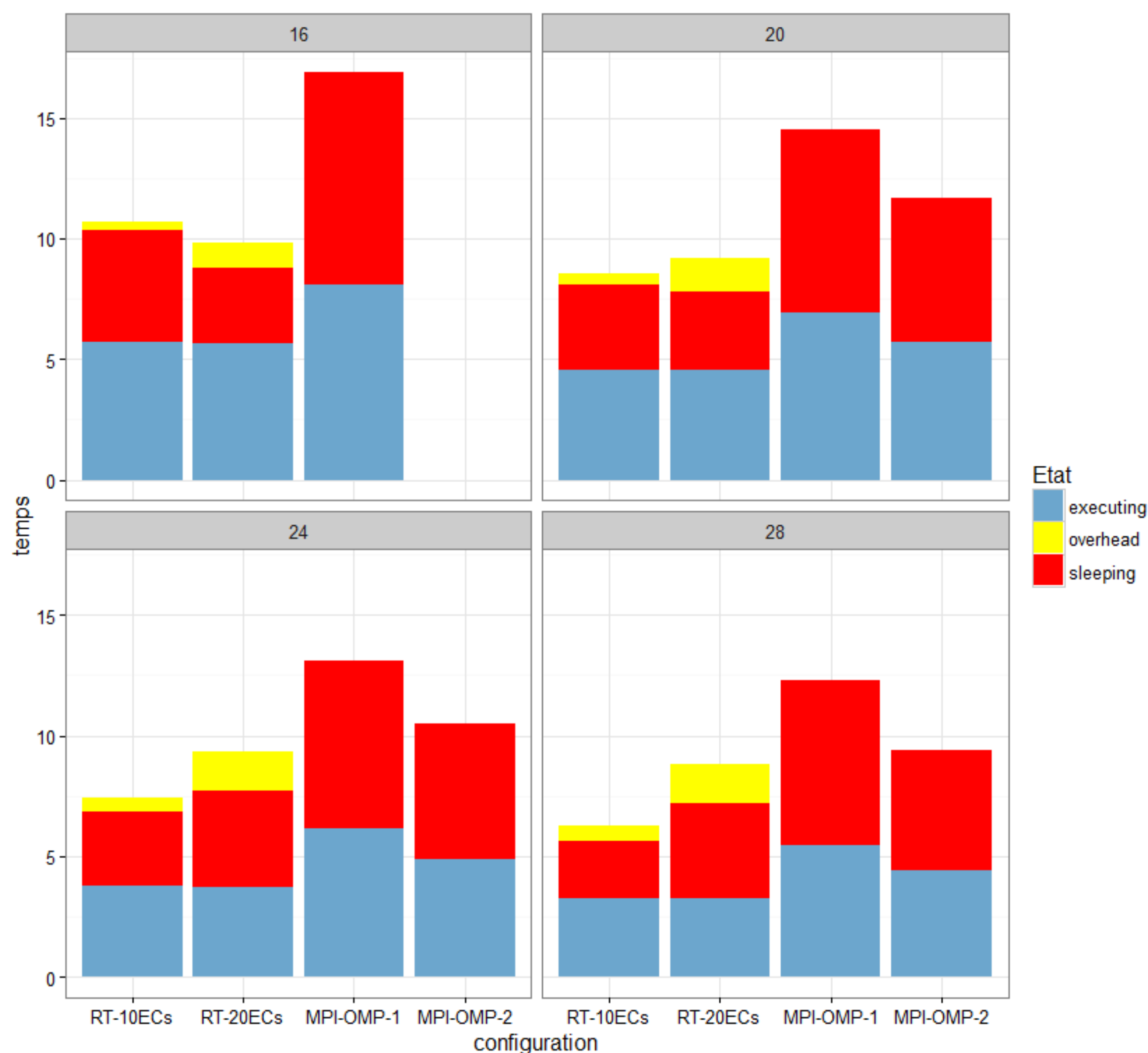


FIGURE 3.32 – Proportion du temps passé dans les différents états.

à calculer effectivement et en jaune pour le temps correspondant à l'*overhead*.

Pour la version MPI+OpenMP, on constate qu'utiliser 2 processus par nœud réduit de manière significative le temps passé à calculer. La proportion de temps passé dans l'état *sleeping* reste à peu près de 50% dans tous les cas.

Si on compare les 2 configurations de la version en tâches, on retrouve que pour 16 nœuds, la meilleure configuration est pour 20 ECs par nœud. Pour ce nombre de nœuds, augmenter le nombre d'ECs par nœud permet de réduire le temps passé dans l'état *sleeping*. On constate par contre que l'overhead est plus important qu'avec 10 ECs par nœud.

Cependant, dès que l'on utilise 20 nœuds et plus, l'avantage revient à la version 10 ECs par nœud. Pour ces cas-là, on retombe déjà progressivement dans une situation où l'overhead impliqué par l'augmentation du nombre d'ECs devient important. En effet, en ajoutant des nœuds, on augmente le nombre d'ECs global et on diminue le nombre de cellules par EC. Il serait appréciable de pouvoir limiter cet overhead afin de réellement tirer parti des possibilités offertes par la version en tâches. En principe, la manière la plus simple de disposer de plus d'asynchronisme est d'augmenter le nombre d'ECs, mais actuellement l'overhead associé et lié

au fonctionnement du support d'exécution lui-même limite ce bénéfice potentiel.

Discussion

Dans ce chapitre, nous avons décrit une démarche de conception et de mise en œuvre pour le développement d'une version en tâches du solveur aérodynamique utilisant le runtime StarPU. Les ajustements et les optimisations que permettent l'utilisation d'un runtime concernant l'exploitation de la machine (utilisation de différentes tailles de worker) ainsi que la puissance d'expressivité du parallélisme par le modèle de graphe de tâches sont clairement bénéfiques. Il est ainsi plus aisé d'exprimer les dépendances de manière plus asynchrone avec un ordre d'exécution moins contraint, ce qui permet en particulier des optimisations au niveau de l'ordonnancement des tâches.

Afin de générer les tâches, nous avons utilisé une abstraction appelé “élément de calcul” (EC). Le nombre d'ECs est actuellement un paramètre au lancement du calcul qui permet d'agir sur la granularité des tâches qui sont générées. Augmenter ce nombre d'ECs permet d'exploiter plus finement les dépendances réelles de l'algorithme, cependant l'augmentation du nombre de tâches qui en résulte génère un overhead qui peut devenir trop important pour gagner globalement en efficacité.

Pour évaluer la version en tâches, nous avons utilisé un cas industriel de calcul d'onde de souffle. En mémoire partagée, la version en tâches fait mieux que la version de production. Sur ce même cas, en mémoire distribuée, tant que l'overhead de la méthode ne devient pas trop élevé, la version en tâches est meilleure, mais elle finit par être rattrapée par la version MPI+OpenMP. Sur un cas de plus grande taille qui a été fabriqué pour l'occasion, on obtient des performances bien meilleure lorsqu'on utilise la version en tâches. Cependant, on constate qu'il reste une marge de progression : l'augmentation du nombre d'ECs permet plus d'asynchronisme mais augmente l'overhead. S'il était possible de réduire cet overhead, les gains potentiels pour la version en tâches serait encore plus importants. Les expériences conduites avec ce démonstrateur sont très encourageantes pour la poursuite du développement d'une version à base de tâches du code FLUSEPA et constituent une preuve de concept convaincante.

Chapitre 4

Conclusions et perspectives

Sommaire

4.1 Bilan de la thèse 89
4.2 Perspectives 90

4.1 Bilan de la thèse

La simulation numérique est donc d'une grande importance dans le domaine spatial. Les contraintes posées par les simulations à effectuer dans ce cadre sont assez particulières en comparaison à celles d'autres industries et les codes disponibles sur étagère ne sont pas généralement adaptés. Le code FLUSEPA, développé en interne chez Airbus Defence and Space (ADS), a été principalement conçu pour la simulation d'écoulements instationnaires ainsi que pour prendre en compte les phénomènes liés aux séparations d'étages.

Au début de cette thèse, seule une version modérément parallèle en mémoire partagée était disponible. L'objectif de ce travail a donc été double : dans un premier temps, il s'est agi de concevoir, de mettre en œuvre et de valider en vraie grandeur une première version du code utilisant des techniques éprouvées de programmation hybride (MPI+OpenMP) pour des grands clusters de calcul pouvant exploiter un parallélisme de types distribué et mémoire partagée ; dans un second temps, il s'est agi d'identifier les faiblesses de cette première version et d'évaluer l'impact que pourrait avoir l'utilisation du paradigme de programmation parallèle basé sur une description à l'aide d'un graphe de tâches s'exécutant au dessus d'un support d'exécution (runtime).

Dans le chapitre 2, on a présenté la parallélisation du code en version MPI+OpenMP. La parallélisation du solveur aérodynamique, et notamment de son schéma d'intégration temporelle adaptatif, est étudiée et est évaluée sur un cas industriel correspondant à un calcul d'onde de souffle. Les performances obtenues sont satisfaisantes, mais de nombreuses synchronisations sont présentes.

De même, la parallélisation du calcul des intersections de maillages est aussi présentée et illustrée par un calcul de séparation des EAP du lanceur Ariane 5. L'approche choisie ici a été de spécialiser les processus, certains étant destinés aux calculs aérodynamiques et d'autres aux calculs des intersections. Afin de recouvrir le temps de calcul des intersections, une extrapolation de la cinématique est effectuée et permet ainsi le calcul pour un instant futur. La nouvelle topologie est ensuite appliquée au moment opportun. Cette approche donne des résultats

intéressants. En effet, et de manière générale, le calcul des intersections est bien recouvert, mais l'application de la nouvelle topologie coûte cependant assez cher et ne passera pas à l'échelle car elle repose sur un contrôle centralisé. Cette version hybride MPI+OpenMP du code FLUSEPA a permis des gains substantiels et est désormais la version de production d'ADS.

Le chapitre 3 est relatif à la parallélisation du solveur aérodynamique en utilisant une expression du parallélisme basée sur le paradigme de graphe de tâches s'exécutant au dessus d'un support d'exécution, en l'occurrence StarPU qui a été conçu par l'équipe-projet STORM. Les évolutions actuelles des plateformes de calcul haute performance sont rapides et il est difficile d'adapter simplement et efficacement un code de nature industrielle aux nouvelles architectures qui sont de plus en plus hétérogènes (CPUs, accélérateurs de calcul, hiérarchie mémoire). L'approche graphe de tâches au-dessus d'un support d'exécution apporte une réponse effective et prometteuse concernant le problème de la portabilité des performances en virtualisant autant que faire se peut l'architecture sous-jacente et en délégant au support d'exécution l'ordonnancement et l'exécution des tâches. Dans notre cas, on exploite ce paradigme de graphe de tâches afin de mieux décrire les dépendances dans le code de calcul pour limiter les synchronisations et pouvoir exploiter au maximum les asynchronismes potentiels. Nous étudions différentes optimisations nécessaires pour obtenir une version performante permettant de rivaliser avantageusement avec la version de référence décrite au chapitre 2. Les choix qui ont été faits permettent une séparation effective entre l'écriture des noyaux de calcul et les optimisations informatiques conduisant à une mise en œuvre efficace. Les performances obtenues avec cette dernière version sont prometteuses à la fois en mémoire partagée et en mémoire distribuée. Il apparait que la description en tâches du problème permet notamment d'utiliser les ressources de manière plus judicieuse : sur un nœud, on peut ainsi utiliser les ressources de manière efficace en utilisant un seul processus, là où il est nécessaire de placer plusieurs processus par nœud avec une version MPI+OpenMP plus classique. Dans notre cas, c'est un bénéfice très important vues l'irrégularité du problème et les synchronisations que le passage à une version distribuée impliquent.

4.2 Perspectives

Ajout de fonctionnalités Les fonctionnalités actuelles et futures du code FLUSEPA semblent particulièrement bien adaptées au parallélisme à base de tâches. Dans un premier temps, il faudra réintégrer toutes les méthodes présentes dans le solveur aérodynamique original. Fondamentalement, il s'agit de refactoriser le code en suivant le même principe que celui qui a été utilisé au chapitre 3.

Dans la section 2.3.1, on a montré que l'on pouvait calculer les intersections en avance pendant les calculs aérodynamiques et que la charge liée à cette opération variait au cours du temps. La solution actuelle qui repose sur des processus spécialisés a plusieurs faiblesses. Une allocation statique des processus mal adaptée peut entraîner un temps *idle* important des processus : il serait judicieux de l'adapter en cours de calcul. Cependant, l'autre problème provient du coût d'application de la nouvelle topologie qui reste fixe et peut être élevé par rapport au coût du solveur aérodynamique.

La suite logique de notre travail serait donc de mener à bien la parallélisation du module de calcul d'intersection en utilisant aussi le paradigme de graphe de tâches. L'idée serait d'effectuer le calcul d'intersection et les calculs aérodynamiques sur les mêmes processus et de bénéficier de la localité des données. La stratégie imaginée pour tirer parti de l'indépendance des calculs peut être reprise pour faire en sorte de calculer les intersections en avance. Cette fois-ci, par

contre, on n'aurait plus le problème des synchronisations résultant de l'utilisation d'un contrôle centralisé. Dans un premier temps, on pourra garder la spécialisation des processus telle qu'elle existe pour développer une version en tâches qui soit indépendante du solveur aérodynamique. Une fois cette version fonctionnelle et validée, on pourra s'attaquer à l'ordonnancement des tâches de différentes natures au sein des mêmes processus. L'utilisation des contextes au sein de StarPU semble pertinente pour effectuer ces développements et pouvoir ajuster l'utilisation des ressources entre le calcul d'intersection et les calculs aérodynamiques.

A noter qu'il y a aussi actuellement une thèse concernant l'utilisation du raffinement de maillage au sein de FLUSEPA. Là aussi, il s'agit d'une opération qu'il faudra intégrer et qui devrait bénéficier fortement du parallélisme à base de tâches.

Au delà de l'ajout de ces fonctionnalités qui est une opération prioritaire, il reste aussi plusieurs autres pistes à explorer pour améliorer la performance intrinsèque de la version en tâches actuelle.

Adaptation des ECs Différentes optimisations sont possibles pour le solveur et une d'entre elles concerne les ECs. S'il est clair qu'il faut maîtriser le nombre d'ECs pour ne pas avoir un overhead trop important, ce n'est cependant pas la seule piste à explorer.

Actuellement, on se contente d'une décomposition de domaine pour obtenir nos ECs. Il serait possible de prendre en compte le niveau temporel des cellules pour limiter les interactions entre ECs. En limitant les frontières de faibles niveaux temporels, on peut augmenter l'efficacité de l'algorithme de regroupement de tâches. Il est aussi possible de faire en sorte de rajouter des ECs dans des zones d'intérêt pour augmenter l'asynchronisme et notamment au niveau des bords impliquant des communications MPI : un nombre plus important d'ECs dans ces zones pourrait offrir plus d'opportunités de recouvrement calcul-communication. Enfin pouvoir modifier les ECs en cours de calcul, entre chaque itération générale, pourrait être aussi très bénéfique. Comme on l'a vu, le niveau temporel des cellules évolue en cours du calcul et une décomposition statique peut perdre ses avantages initiaux. Étant donnée la nature des interactions entre les ECs, il doit être possible de faire ces modifications tout en exécutant les tâches du solveur aérodynamique.

Tâches parallèles Un compromis raisonnable est à trouver concernant l'utilisation des tâches parallèles. En effet, lorsque la taille des workers augmente, et étant donnée la scalabilité limitée actuelle des noyaux de calcul, le code perd en efficacité. A contrario, utiliser des workers composés uniquement d'un cœur de calcul peut mener à une famine à la fin des itérations. L'utilisation des priorités peut limiter de manière très satisfaisante ce problème. Cependant, lorsque l'on considère un calcul en mémoire distribuée, il peut être intéressant d'avoir une stratégie de priorité qui favorise le recouvrement calcul-communication sur les bords plutôt qu'une stratégie qui ne soit dédiée qu'à régler le problème de famine. Une solution qui pourrait tirer parti à la fois d'une meilleure efficacité des noyaux de calcul lorsqu'on utilise une taille de worker de 1 et du fait de limiter la famine serait de redimensionner les workers en cours de calcul. Ainsi, si le support d'exécution peut remonter des informations concernant l'occupation des workers à l'application, celle-ci pourrait redimensionner les workers lorsque la situation de famine se produit. Dans ce cas, pour la majorité du calcul on bénéficierait de la meilleure efficacité des noyaux et lorsque le phénomène de famine commence à se produire, on n'aurait plus qu'un worker unique avec donc toujours l'utilisation potentielle de l'ensemble des cœurs de calculs.

Outre ce redimensionnement au niveau de la taille des workers, il est possible d'utiliser des tailles de workers hétérogènes. Dans les études qui ont été faites, on a dimensionné les workers

de manière homogène. Par exemple, la configuration 4×4 correspondait à 4 workers de taille 4 ; il pourrait être judicieux d'évaluer des configurations de type $1 \times 8 + 2 \times 4$. En effet, l'efficacité parallèle des différents noyaux de calcul n'est pas la même et certains noyaux sont moins pénalisés par l'utilisation d'une grosse taille de worker parallèle. Cette efficacité dépend aussi de la taille des données traitées. Dans notre implémentation, afin de générer le maximum d'asynchronisme possible, on a différencié les cellules des bords des cellules intérieures. Naturellement, les cellules des bords représentent une fraction des cellules intérieures et les tâches qui les traitent contiennent donc moins de calcul. Il existe donc plusieurs granularités pour les tâches. Il pourrait donc être intéressant d'étudier l'impact de l'utilisation d'une configuration hétérogène des workers couplée avec un ordonnanceur capable de prendre en compte les spécificités à la fois des noyaux de calcul et de la granularité des tâches.

Amélioration de la version distribuée En mémoire partagée, les techniques développées dans cette thèse sont suffisantes pour traiter le problème de famine. Lorsqu'on effectue le calcul en mémoire distribuée, on est confronté à d'autres problèmes. Notamment, comme l'équilibrage de charge entre les différents nœuds n'est pas parfait, ceci implique des synchronisations entre itérations. Afin de limiter ce problème, il est possible de faire en sorte de pipeliner les itérations générales. Pour cela, il faut retirer les synchronisations actuellement présentes dans le code. Actuellement la synchronisation principale provient du calcul du pas de temps, ce qui permet ensuite de classer les cellules dans les différents niveaux temporels. La cellule imposant le pas de temps le plus faible est recherchée et détermine le pas de temps de l'itération générale. Une autre solution est de repartir du pas de temps précédent et d'autoriser des niveaux temporels "négatifs". Ainsi, on a toujours un pas de temps de référence, mais ce dernier n'est plus déterminé par une réduction qui implique une synchronisation. Au lieu de cela, on permettrait à des cellules d'être deux fois plus lentes (ou une autre puissance de deux) que le pas de temps de référence. Il faudrait cependant retravailler la boucle d'insertion des tâches pour prendre en compte ce cas.

Cependant, si pipeliner les itérations permettrait de réduire les synchronisations dans un premier temps, le problème est juste retardé si le déséquilibre perdure. Afin de réellement tirer parti du pipeline des itérations générales, il faudrait pouvoir remettre en cause l'équilibrage de charge entre ces itérations générales. Si on dispose de la capacité de modifier les ECs entre les itérations générales, on pourrait alors imaginer un système d'échange d'ECs entre domaines voisins. Là aussi, les dépendances étant locales, il semble possible de faire ceci de manière asynchrone.

Le choix de développer une version en tâches du code FLUSEPA ouvre donc de nombreuses nouvelles portes, ce qui n'aurait pas été possible avec une parallélisation plus classique du code. Notre étude est une preuve de concept montrant qu'il est tout à fait possible d'obtenir des performances prometteuses pour une première version à base de tâches avec des optimisations possibles, soit de nature algorithmique, soit destinées à exploiter plus efficacement les ressources de la plateforme parallèle de calcul.

Bibliographie

[.09a] MAGMA Users' Guide, version 0.2. http ://icl.cs.utk.edu/magma, November 2009.

[.09b] PLASMA Users' Guide, Parallel Linear Algebra Software for Multicore Architectures, Version 2.0. http ://icl.cs.utk.edu/plasma, November 2009.

[AAB+12] David Auber, Daniel Archambault, Romain Bourqui, Antoine Lambert, Morgan Mathiaut, Patrick Mary, Maylis Delest, Jonathan Dubois, and Guy Melançon. The Tulip 3 Framework : A Scalable Software Library for Information Visualization Applications Based on Relational Data. Research Report RR-7860, INRIA, January 2012.

[AAD+10] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Hatem Ltaief, Raymond Namyst, Samuel Thibault, and Stanimire Tomov. Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs. In Wen mei W. Hwu, editor, *GPU Computing Gems*, volume 2. Morgan Kaufmann, September 2010.

[AAD+11a] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Mathieu Faverge, Julien Langou, Hatem Ltaief, and Stanimire Tomov. LU factorization for accelerator-based systems. In Howard Jay Siegel and Amr El-Kadi, editors, *The 9th IEEE/ACS International Conference on Computer Systems and Applications, AICCSA 2011, Sharm El-Sheikh, Egypt, December 27-30, 2011*, pages 217–224. IEEE, 2011.

[AAD+11b] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Mathieu Faverge, Hatem Ltaief, Samuel Thibault, and Stanimire Tomov. QR Factorization on a Multicore Node Enhanced with Multiple GPU Accelerators. In *IPDPS*, pages 932–943. IEEE, 2011.

[ABC+14] Emmanuel Agullo, Bérenger Bramas, Olivier Coulaud, Eric Darve, Matthias Messner, and Toru Takahashi. Task-based FMM for multicore architectures. *SIAM J. Scientific Computing*, 36(1), 2014.

[ABI+09] Eduard Ayguadé, Rosa M. Badia, Francisco D. Igual, Jesús Labarta, Rafael Mayo, and Enrique S. Quintana-Ortí. An extension of the starss programming model for platforms with multiple GPUs. In *Euro-Par*, pages 851–862, 2009.

[AK02] Randy Allen and Ken Kennedy. *Optimizing compilers for modern architectures : a dependence-based approach*, volume 289. Morgan Kaufmann San Francisco, 2002.

[ATNW11] Cedric 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, Special Issue : Euro-Par 2009*, 23 :187–198, February 2011.

[Bau14] Michael Edward Bauer. *Legion : Programming Distributed Heterogeneous Architectures with Logical Regions*. PhD thesis, Stanford University, 2014.

[BBD+10] G. Bosilca, A. Bouteiller, A Danalis, M. Faverge, H. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczek, A. YarKhan, and J. Dongarra. Distributed-Memory Task Execution and Dependence Tracking within DAGuE and the DPLASMA Project. *Innovative Computing Laboratory Technical Report*, 2010.

[BBD+11] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Azzam Haidar, Thomas Hérault, Jakub Kurzak, Julien Langou, Pierre Lemarinier, Hatem Ltaief, Piotr Luszczek, Asim YarKhan, and Jack Dongarra. Flexible Development of Dense Linear Algebra Algorithms on Massively Parallel Architectures with DPLASMA. In *IPDPS Workshops*, pages 1432–1441. IEEE, 2011.

[BBD+12] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Hérault, Pierre Lemarinier, and Jack Dongarra. DAGuE : A generic distributed DAG engine for high performance computing. *Parallel Computing*, 38(1-2) :37–51, 2012.

[BBD+13] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack J. Dongarra. PaRSEC : Exploiting Heterogeneity to Enhance Scalability. *Computing in Science and Engineering*, 15(6) :36–45, 2013.

[Ben13] Farid Benyoucef. *Amélioration de la prévision des écoulements turbulents par une approche URANS avancée*. PhD thesis, Toulouse, ISAE, 2013.

[Ber87] Marsha J. Berger. On conservation at grid interfaces. *SIAM journal on numerical analysis*, 24(5) :967–984, 1987.

[BHL+09] Rosa M. Badia, José R. Herrero, Jesús Labarta, Josep M. Pérez, Enrique S. Quintana-Ortí, and Gregorio Quintana-Ortí. Parallelizing dense and banded linear algebra libraries using SMPSs. *Concurrency and Computation : Practice and Experience*, 21(18) :2438–2456, 2009.

[BLKD08] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. Parallel tiled QR factorization for multicore architectures. *Concurrency and Computation : Practice and Experience*, 20(13) :1573–1590, 2008.

[Boi14] Lionel Boillot. *Contributions to the mathematical modeling and to the parallel algorithmic for the optimization of an elastic wave propagator in anisotropic media*. Phd thesis, Université de Pau et des Pays de l'Adour, December 2014. Collaboration Inria-Total.

[Bre91] Pierre Brenner. Three-dimensional aerodynamics with moving bodies applied to solid propellant. In *AIAA paper 91-2304*, 1991.

[Bre93] Pierre Brenner. Numerical simulation of 3D and unsteady aerodynamics about bodies in relative motion applied to TSTO separation. In *AIAA paper 93-2542*, 1993.

[Bre95] Pierre Brenner. Simulation du mouvement relatif de corps soumis à un écoulement instationnaire par une méthode de chevauchement de maillages. In *AGARD Conference Proceedings 578 : Progress and Challenges in CFD Methods and Algorithms*, 1995.

[But97] David R Butenhof. *Programming with POSIX threads*. Addison-Wesley Professional, 1997.

[But12] Alfredo Buttari. Fine granularity sparse QR factorization for multicore based systems. In *Proceedings of the 10th international conference on Applied Parallel and Scientific Computing - Volume 2*, PARA'10, pages 226–236, Berlin, Heidelberg, 2012. Springer-Verlag.

[CBP07] Jean Collinet, Pierre Brenner, and Sandrine Palerm. Dynamic Stability of the Huygens Probe. In *Aerospace Science and Technology, vol. 11*, pages 202–210, 2007.

[CGH+15] Terry Cojean, Abdou Guermouche, Andra Hugo, Raymond Namyst, and Pierre-André Wacrenier. Exploiting two-level parallelism by aggregating computing resources in task-based applications over accelerator-based machines. https ://hal.inria.fr/hal-01181135, 2015.

[Cho68] Alexandre Joel Chorin. Numerical solution of the navier-stokes equations. *Mathematics of computation*, 22(104) :745–762, 1968.

[CL95] Michel Cosnard and Michel Loi. Automatic task graph generation techniques. In *System Sciences, 1995. Proceedings of the Twenty-Eighth Hawaii International Conference on*, volume 2, pages 113–122. IEEE, 1995.

[CZB+08] E. Chan, F. G. Van Zee, P. Bientinesi, E. S. Quintana-Ortí, G. Quintana-Ortí, and R. A. van de Geijn. Supermatrix : a multithreaded runtime scheduling system for algorithms-by-blocks. In *PPOPP*, pages 123–132, 2008.

[dOSdKM10] B de Oliveira Stein, J Chassin de Kergommeaux, and G Mounié. Pajé trace file format. Technical report, Technical report, ID-IMAG, Grenoble, France, 2002. http ://www-id. imag. fr/Logiciels/paje/publications, 2010.

[EGK+01] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C North, and Gordon Woodhull. Graphviz—open source graph drawing tools. In *Graph Drawing*, pages 483–484. Springer, 2001.

[Fed45] Herbert Federer. The Gauss-Green theorem. *Transactions of the American Mathematical Society*, 58(1) :44–76, 1945.

[Ger99] M Germano. From RANS to DNS : towards a bridging model. In *Direct and Large-Eddy Simulation III*, pages 225–236. Springer, 1999.

[GGB14] Damien Genet, Abdou Guermouche, and George Bosilca. Assembly operations for multicore architectures using task-based runtime systems. In *Euro-Par 2014 : Parallel Processing Workshops - Euro-Par 2014 International Workshops, Porto, Portugal, August 25-26, 2014, Revised Selected Papers, Part II*, pages 338–350, 2014.

[GLMFR13] Thierry Gautier, Fabien Le Mentec, Vincent Faucher, and Bruno Raffin. X-Kaapi : A multi paradigm runtime for multicore architectures. In *42nd International Conference on Parallel Processing, ICPP 2013, Lyon, France, October 1-4, 2013*, pages 728–735, 2013.

[GMP95] J-P Gillyboeuf, P Mansuy, and S Pavsic. Two new Chimera methods : application to missile separation. *Tiré à part- Office national d'études et de recherches aerospatiales*, 1995.

[GZI+79] Sergueï Godounov, A. Zabrodine, Mikhail Ivanov, A. Kraiko, and G. Prokopov. *Résolution numérique des problemes multidimensionnels de la dynamique des gaz.* Editions Mir, 1979.

[HAC74] C. W Hirt, A. A Amsden, and J. L Cook. An arbitrary Lagrangian-Eulerian computing method for all flow speeds. *Journal of Computational Physics*, 14(3) :227–253, 1974.

[HGNW13] Andra Hugo, Abdou Guermouche, Raymond Namyst, and Pierre-André Wacrenier. Composing multiple StarPU applications over heterogeneous machines : a supervised approach. In *Third International Workshop on Accelerators and Hybrid Exascale Systems*, Boston, USA, May 2013.

[HSc12] T. D. R. Hartley, E. Saule, and Ü. V. Çatalyürek. Improving performance of adaptive component-based dataflow middleware. *Parallel Computing*, 38(6-7) :289–309, 2012.

[IG96] Ross Ihaka and Robert Gentleman. R : a language for data analysis and graphics. *Journal of computational and graphical statistics*, 5(3) :299–314, 1996.

[JR13] James Jeffers and James Reinders. *Intel Xeon Phi coprocessor high-performance programming.* Newnes, 2013.

[KBW92] W. L. Kleb, J. T. Batina, and M. H. Williams. Temporal adaptive Euler/Navier-Stokes algorithm involving unstructured dynamic meshes. *AIAA journal*, 30(8) :1980–1985, 1992.

[KD09] Jakub Kurzak and Jack Dongarra. Fully dynamic scheduler for numerical computing on multicore processors. *LAPACK working note*, lawn220, 2009.

[KK93] Laxmikant V. Kalé and Sanjeev Krishnan. CHARM++ : A portable concurrent object oriented system based on c++. In *OOPSLA*, pages 91–108, 1993.

[KK11] D. M. Kunzman and L. V. Kalé. Programming heterogeneous clusters with accelerators using object-based programming. *Scientific Programming*, 19(1) :47–62, 2011.

[KLC94] Kai-Hsiung Kao, Meng-Sing Liou, and Chuen-Yen Chow. Grid adaptation using Chimera composite overlapping meshes. *AIAA journal*, 32(5) :942–949, 1994.

[KLDB10] Jakub Kurzak, Hatem Ltaief, Jack Dongarra, and Rosa M. Badia. Scheduling dense linear algebra operations on multicore processors. *Concurrency and Computation : Practice and Experience*, 22(1) :15–44, 2010.

[KN+91] Eleftherios Koutsofios, Stephen North, et al. Drawing graphs with dot. Technical report, Technical Report 910904-59113-08TM, AT&T Bell Laboratories, Murray Hill, NJ, 1991.

[Lac15] X. Lacoste. *Scheduling and memory optimizations for sparse direct solver on multi-core/multi-gpu cluster systems.* PhD thesis, LaBRI, Université Bordeaux, Talence, France, February 2015.

[LHK09] C.-K. Luk, S. Hong, and H. Kim. Qilin : exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *MICRO*, pages 45–55, 2009.

[LMPZ85] R. Löhner, K. Morgan, J. Peraire, and O. A. Zienkiewicz. *Finite element methods for high speed flows.* University College of Swansea Institute for Numerical Methods in Engineering, 1985.

[Lop15] Florent Lopez. *Task-based multifrontal QR solver for heterogeneous architectures*. PhD thesis, University Paul Sabatier, Toulouse, France, 2015.

[Lun08] Frank Luna. *Introduction to 3D game programming with DirectX 10*. Jones & Bartlett Publishers, 2008.

[LY12] Hatem Ltaief and Rio Yokota. Data-driven execution of fast multipole methods. *CoRR*, abs/1203.0889, 2012.

[OHL+08] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. GPU computing. *Proceedings of the IEEE*, 96(5) :879–899, 2008.

[Ors70] Steven A Orszag. Analytical theories of turbulence. *Journal of Fluid Mechanics*, 41(02) :363–386, 1970.

[OS83] Stanley Osher and Richard Sanders. Numerical approximations to nonlinear conservation laws with locally varying time and space grids. *Mathematics of Computation*, 41(164) :321–336, 1983.

[PB88] Mehtab M. Pervaiz and Judson R. Baron. Temporal and spatial adaptive algorithm for reacting flows. *Communications in Applied Numerical Methods*, 4(1) :97–111, 1988.

[Pon15] Grégoire Pont. *Self adaptive turbulence models for unsteady compressible flows*. PhD thesis, Ecole Nationale Supérieure d'Arts et Métiers, 2015.

[PR96] François Pellegrini and Jean Roman. SCOTCH : A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *High-Performance Computing and Networking*, pages 493–498. Springer, 1996.

[QOIQOvdG09] Gregorio Quintana-Ortí, Francisco D. Igual, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. Solving dense linear systems on platforms with multiple hardware accelerators. *ACM SIGPLAN Notices*, 44(4) :121–130, April 2009.

[QOQOC+08] G. Quintana-Ortí, E. S. Quintana-Ortí, E. Chan, F. G. Van Zee, and R. A. van de Geijn. Scheduling of QR factorization algorithms on SMP and multicore architectures. In *Proceedings of PDP'08*, 2008. FLAME Working Note #24.

[Rei07] James Reinders. *Intel Threading Building Blocks : Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly, 2007.

[SYD09] Fengguang Song, Asim YarKhan, and Jack Dongarra. Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems. In *Proceedings of the ACM/IEEE Conference on High Performance Computing, SC'09*, 2009.

[TBA14] Sean Treichler, Michael Bauer, and Alex Aiken. Realm : an event-based low-level runtime for distributed memory architectures. In *International Conference on Parallel Architectures and Compilation, PACT '14, Edmonton, AB, Canada, August 24-27, 2014*, pages 263–276, 2014.

[VCvdG+09] Field G. Van Zee, Ernie Chan, Robert A. van de Geijn, Enrique S. Quintana-Orti, and Gregorio Quintana-Orti. The libFLAME Library for Dense Matrix Computations. *Computing in Science and Engineering*, 11(6) :56–63, November/December 2009.

[VL03] Bram Van Leer. Towards the ultimate conservative difference scheme. IV. a new approach to numerical convection. *Journal of Computational Physics*, 23(3) :276–299, 1977-03.

[VM07] Henk Kaarle Versteeg and Weeratunge Malalasekera. *An Introduction to Computational Fluid Dynamics : The Finite Volume Method*. Pearson Education, 2007.

[WNDS99] Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL programming guide : the official guide to learning OpenGL, version 1.2*. Addison-Wesley Longman Publishing Co., Inc., 1999.

Liste des publications

[FLU1] Jean-Marie Couteyen, Jean Roman, Pierre Brenner. Towards an efficient Task-based Parallelization over a Runtime System of an Explicit Finite-Volume CFD Code with Adaptive Time Stepping. In *Proceedings of 2016 PDSEC IPDPS Workshop*, pages 1212-1221. *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS) Workshops, Chicago, IL, United States*, May 2016.

[FLU2] Jean-Marie Couteyen, Jean Roman, and Pierre Brenner. Task-based parallelization of a CFD code over a Runtime System. *27th International Conference on Parallel Computational Fluid Dynamics, Montreal, Canada*, May 2015.

[FLU3] Jean-Marie Couteyen, Jean Roman, and Pierre Brenner. MPMD parallelization of an aerodynamic code with bodies in relative motion. *2nd International Workshop on High Performance Computing Simulation in Energy/Transport Domains (HPCSET 2015),Frankfurt, Germany*, July 2015.

[FLU4] Jean-Marie Couteyen, Jean Roman, and Pierre Brenner. FLUSEPA - a Navier-Stokes Solver for Unsteady Problems with Bodies in Relative Motion : Toward a Task-Based Parallel Version over a Runtime System. *SIAM CSE, Salt Lake City, Utah, USA*, March 2015, Poster.

Annexes

Annexe A

Version MPI+OpenMP Traces pour 16 nœuds

Cette Annexe présente 2 traces MPI utilisant toutes les deux 16 nœuds de calcul pour une même simulation d'onde de souffle ($\theta = 4$ donc 16 sous-itérations et même instant physique de la simulation). La Figure A.1 montre le détail de la trace d'exécution en utilisant un processus par nœud et la Figure A.2 celle de la trace en utilisant 2 processus par nœud.

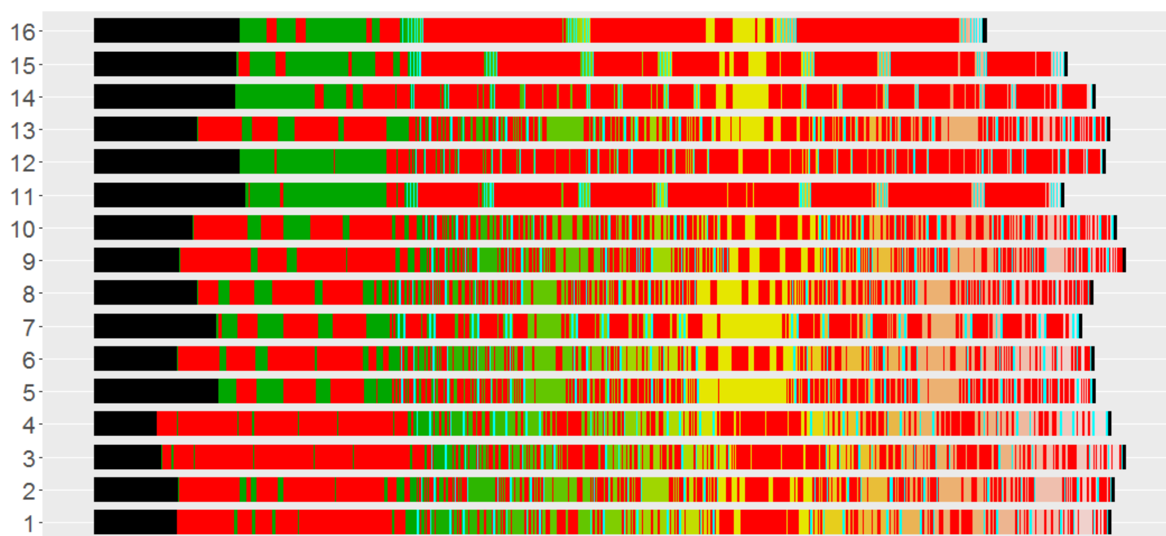


FIGURE A.1 – 16 nœuds, 1 processus par nœud (#ppn=1).

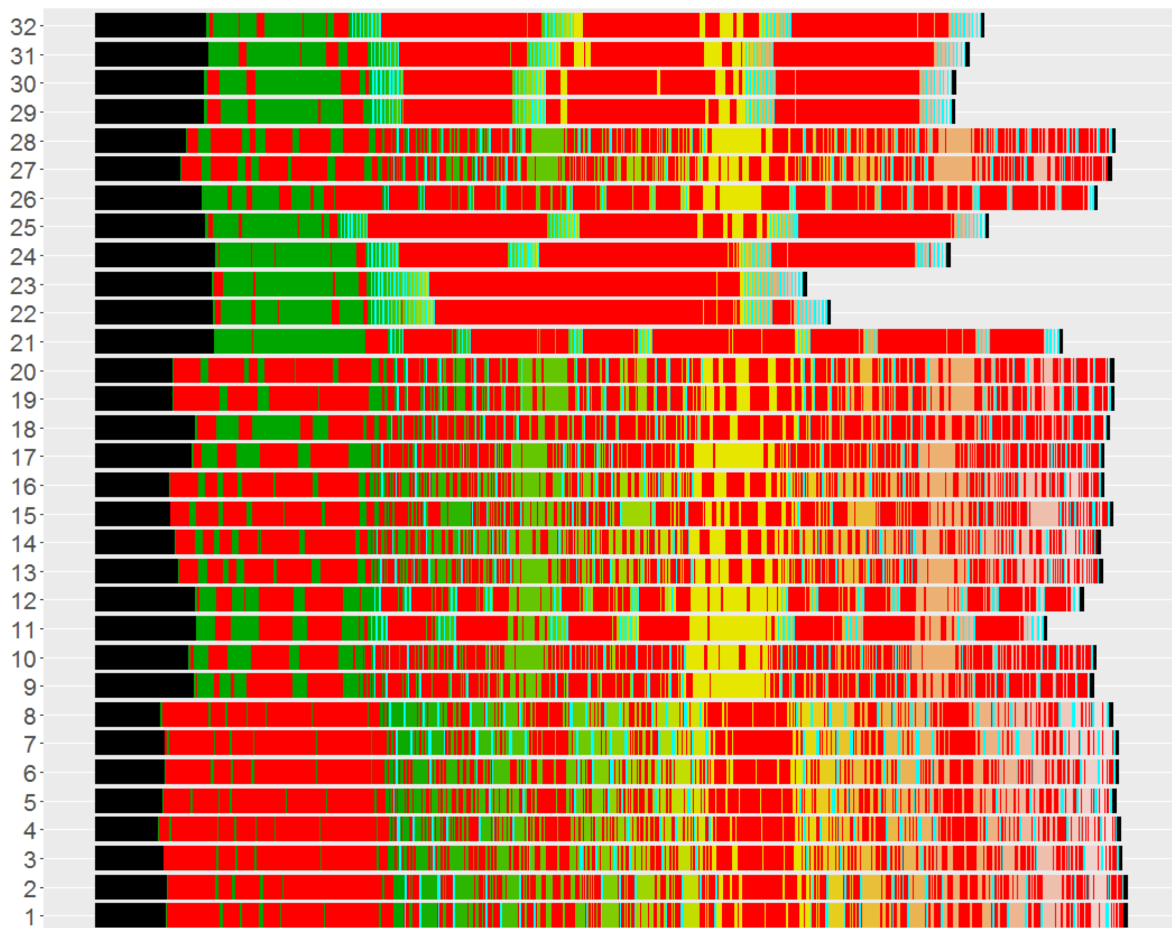


FIGURE A.2 – 16 nœuds, 2 processus par nœud (#ppn=2).

Annexe B

Performance en mémoire partagée de la version en tâches

Cette annexe présente les performances en mémoire partagée pour différents niveaux temporels. La stratégie de priorité est utilisée. On a vu dans le chapitre 2 sur la Figure 2.23 que le coût évoluait au cours du temps. L'expérience est donc faite deux fois : une fois au tout début du calcul (dans ce cas, θ ne dépasse pas 4) et une deuxième fois aux alentours de la 2300ème itération (dans ce cas, θ atteint la valeur 6). La barre de référence (en haut en noir) correspond au calcul fait avec la version MPI+OpenMP en utilisant un processus sur l'ensemble d'un nœud.

B.1 Premier relevé (début du calcul)

B.1.1 Détails pour θ=0

	τ_0
Prop. Faces	100.00 %
Prop. Calc.	100.00 %

FIGURE B.1 – Répartition des faces par niveau temporel et proportion de calcul associée ($\theta = 0$).

#NCE	Tsub	# Tâches élémentaires	# Packs insérés
16	0.10 s	6288	818
32	0.21 s	15084	2116
64	0.51 s	30680	4382
128	0.99 s	65343	9604

FIGURE B.2 – Information sur la création du DAG ($\theta = 0$).

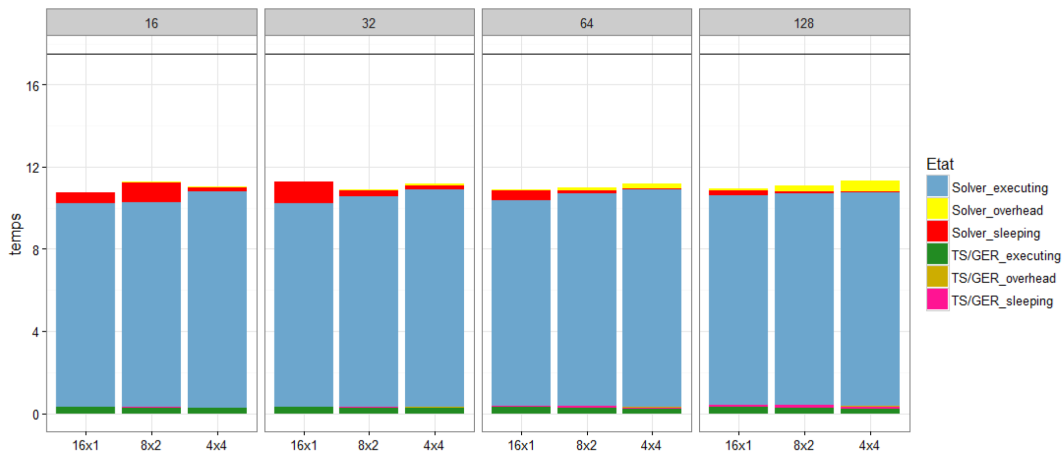


FIGURE B.3 – Performance ($\theta = 0$).

B.1.2 Détails pour θ=1

	τ_0	τ_1
Prop. Faces	0.00004 %	99.99996 %
Prop. Calc.	0.0001 %	99.9999 %

FIGURE B.4 – Répartition des faces par niveau temporel et proportion de calcul associée ($\theta = 1$).

#NCE	Tsub	# Tâches élémentaires	# Packs insérés
16	0.14 s	6436	823
32	0.32 s	15232	2121
64	0.71 s	30828	4387
128	1.72 s	65491	9609

FIGURE B.5 – Information sur la création du DAG ($\theta = 1$).

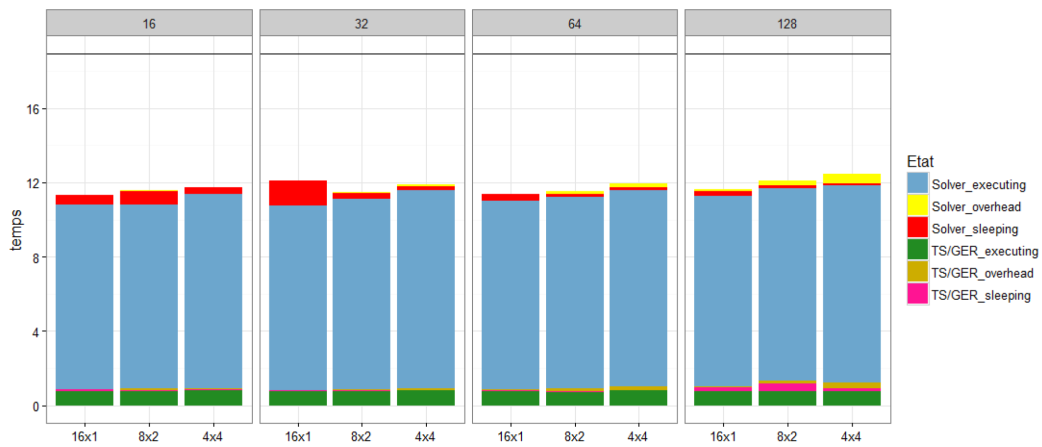


FIGURE B.6 – Performance ($\theta = 1$).

B.1.3 Détails pour θ=2

	τ_0	τ_1	τ_2
Prop. Faces	0.00004 %	0.06110 %	99.93885 %
Prop. Calc.	0.0002 %	0.1221 %	99.8777 %

FIGURE B.7 – Répartition des faces par niveau temporel et proportion de calcul associée ($\theta = 2$).

#NCE	Tsub	# Tâches élémentaires	# Packs insérés
16	0.22 s	8231	1049
32	0.50 s	17607	2390
64	1.03 s	33331	4703
128	2.51 s	69340	10109

FIGURE B.8 – Information sur la création du DAG ($\theta = 2$).

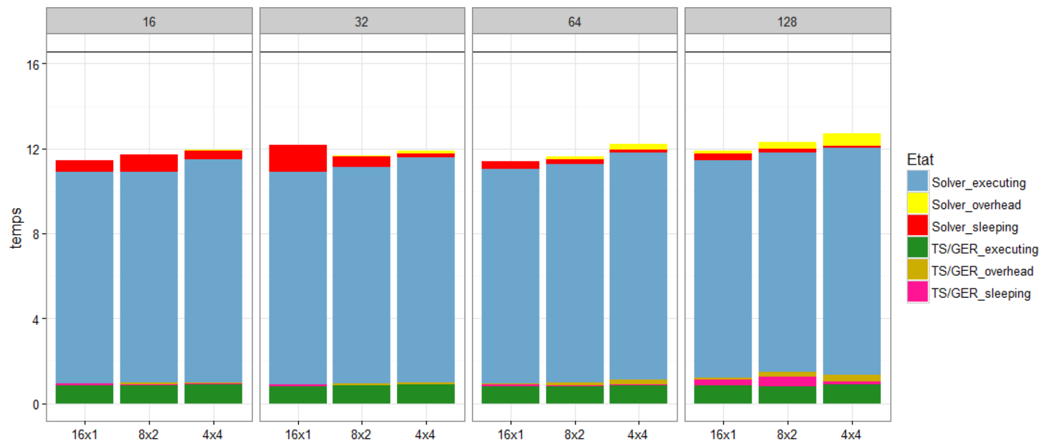


FIGURE B.9 – Performance ($\theta = 2$).

B.1.4 Détails pour θ=3

	τ_0	τ_1	τ_2	τ_3
Prop. Faces	0.00004 %	0.06106 %	1.65976 %	98.27913 %
Prop. Calc.	0.0003 %	0.2398 %	3.2594 %	96.5004 %

FIGURE B.10 – Répartition des faces par niveau temporel et proportion de calcul associée ($\theta = 3$).

#NCE	Tsub	# Tâches élémentaires	# Packs insérés
16	0.35 s	12894	1582
32	0.87 s	24944	3266
64	1.75 s	41260	5712
128	3.85 s	81747	11628

FIGURE B.11 – Information sur la création du DAG ($\theta = 3$).

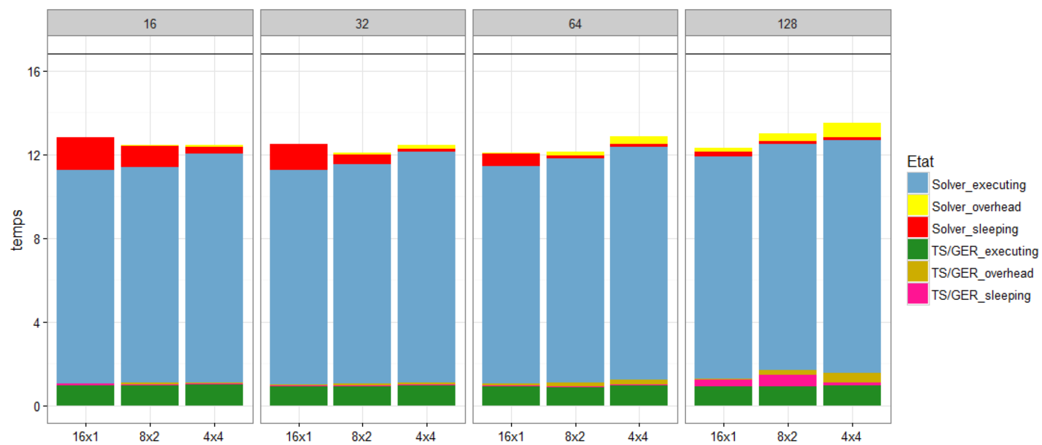


FIGURE B.12 – Performance ($\theta = 3$).

B.1.5 Détails pour θ=4

	τ_0	τ_1	τ_2	τ_3	τ_4
Prop. Faces	0.00004 %	0.06106 %	1.65942 %	3.66178 %	94.61769 %
Prop. Calc.	0.0006 %	0.4479 %	6.0858 %	6.7147 %	86.7510 %

FIGURE B.13 – Répartition des faces par niveau temporel et proportion de calcul associée ($\theta = 4$).

#NCE	Tsub	# Tâches élémentaires	# Packs insérés
16	0.70 s	23506	2826
32	1.85 s	43068	5461
64	3.46 s	64955	8696
128	7.66 s	118633	16259

FIGURE B.14 – Information sur la création du DAG ($\theta = 4$).

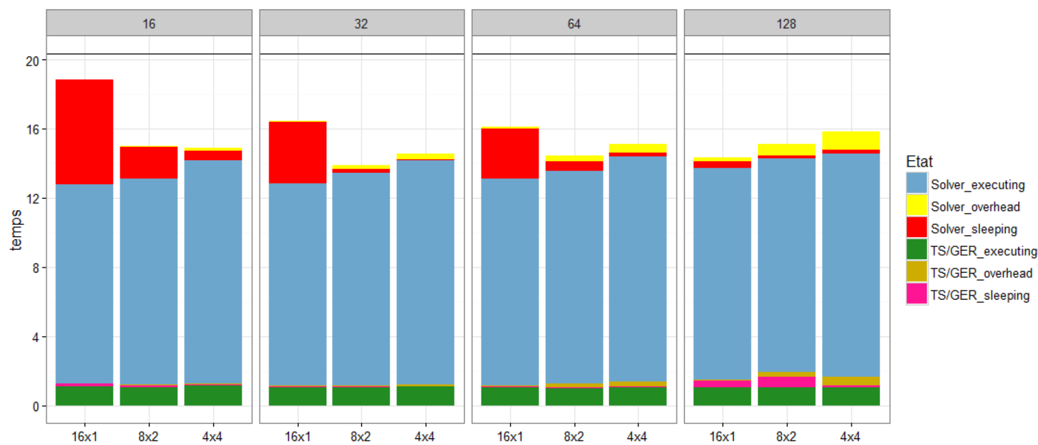


FIGURE B.15 – Performance ($\theta = 4$).

B.2 Deuxième relevé (2300ème itération)

B.2.1 Détails pour θ=0

	τ_0
Prop. Faces	100.00 %
Prop. Calc.	100.00 %

FIGURE B.16 – Répartition des faces par niveau temporel et proportion de calcul associée ($\theta = 0$).

#NCE	Tsub	# Tâches élémentaires	# Packs insérés
16	0.09 s	6288	818
32	0.21 s	15084	2116
64	0.52 s	30680	4382
128	0.90 s	65343	9604

FIGURE B.17 – Information sur la création du DAG ($\theta = 0$).

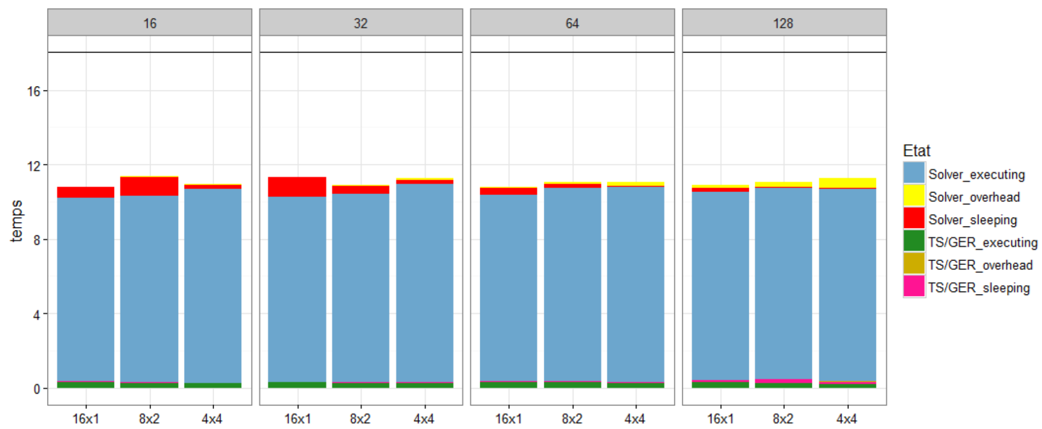


FIGURE B.18 – Performance ($\theta = 0$).

B.2.2 Détails pour θ=1

	τ_0	τ_1
Prop. Faces	0.02 %	99.98 %
Prop. Calc.	0.04 %	99.96 %

FIGURE B.19 – Répartition des faces par niveau temporel et proportion de calcul associée ($\theta = 1$).

#NCE	Tsub	# Tâches élémentaires	# Packs insérés
16	0.14 s	6944	899
32	0.34 s	15659	2176
64	0.71 s	31531	4502
128	1.70 s	66281	9706

FIGURE B.20 – Information sur la création du DAG ($\theta = 1$).

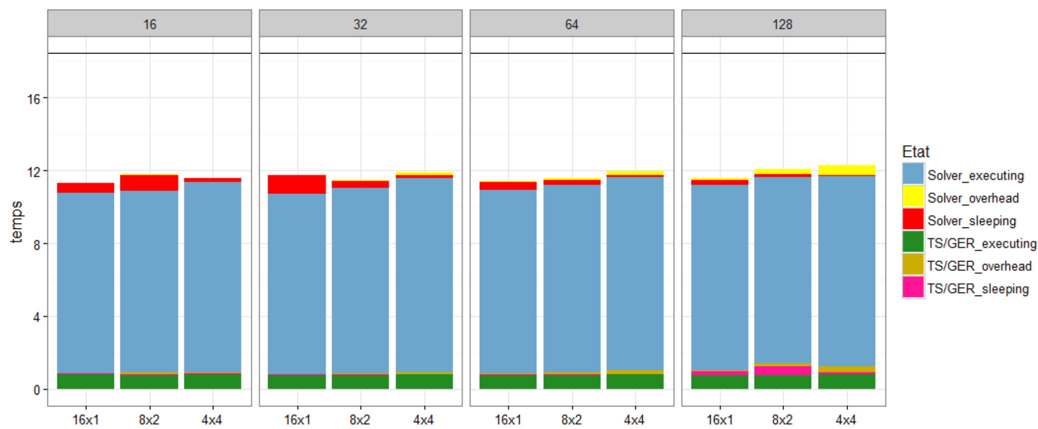


FIGURE B.21 – Performance ($\theta = 1$).

B.2.3 Détails pour θ=2

	τ_0	τ_1	τ_2
Prop. Faces	0.02 %	2.00 %	97.99 %
Prop. Calc.	0.07 %	3.91 %	96.02 %

FIGURE B.22 – Répartition des faces par niveau temporel et proportion de calcul associée ($\theta = 2$).

#NCE	Tsub	# Tâches élémentaires	# Packs insérés
16	0.25 s	10498	1348
32	0.53 s	20516	2765
64	1.04 s	37544	5264
128	2.30 s	74906	10795

FIGURE B.23 – Information sur la création du DAG ($\theta = 2$).

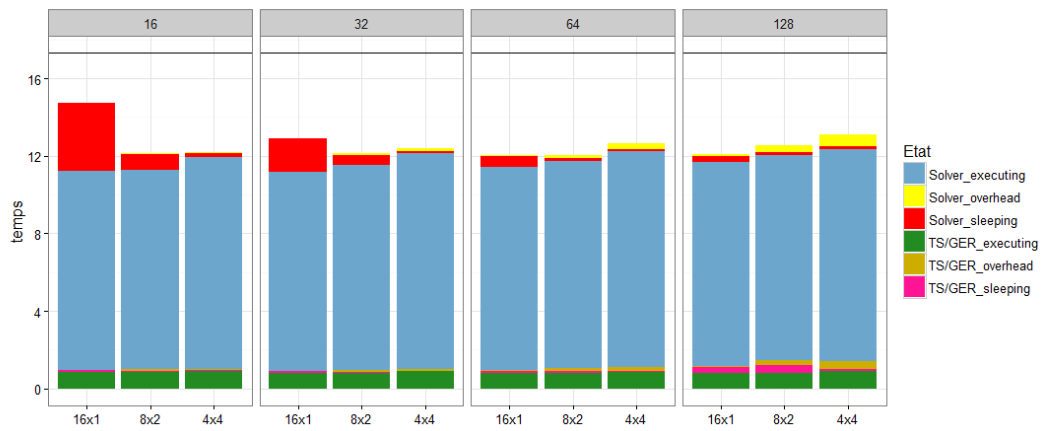


FIGURE B.24 – Performance ($\theta = 2$).

B.2.4 Détails pour θ=3

	τ_0	τ_1	τ_2	τ_3
Prop. Faces	0.02 %	2.00 %	3.04 %	94.94 %
Prop. Calc.	0.13 %	7.32 %	5.57 %	86.98 %

FIGURE B.25 – Répartition des faces par niveau temporel et proportion de calcul associée ($\theta = 3$).

#NCE	Tsub	# Tâches élémentaires	# Packs insérés
16	0.32 s	18249	2292
32	0.94 s	32366	4210
64	1.82 s	53359	7298
128	4.39 s	99288	13836

FIGURE B.26 – Information sur la création du DAG ($\theta = 3$).

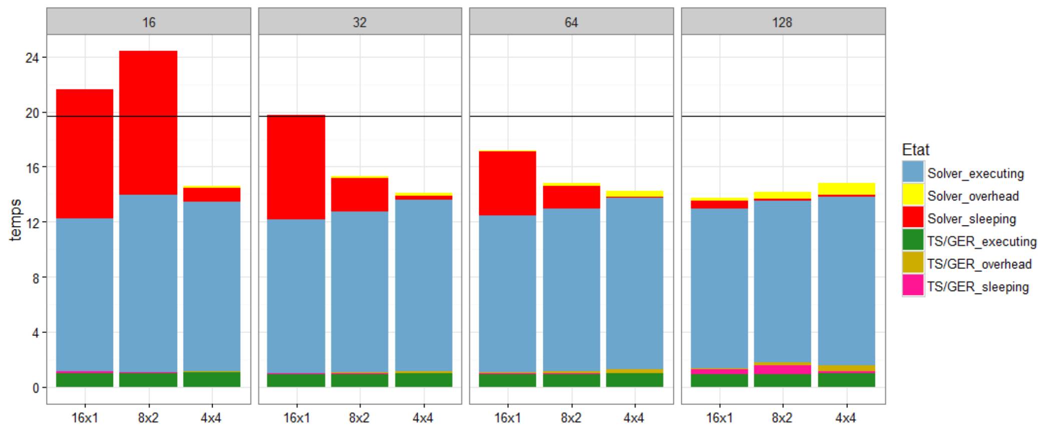


FIGURE B.27 – Performance ($\theta = 3$).

B.2.5 Détails pour θ=4

	τ_0	τ_1	τ_2	τ_3	τ_4
Prop. Faces	0.02 %	2.00 %	3.04 %	5.01 %	89.93 %
Prop. Calc.	0.22 %	12.45 %	9.48 %	7.81 %	70.04 %

FIGURE B.28 – Répartition des faces par niveau temporel et proportion de calcul associée ($\theta = 4$).

#NCE	Tsub	# Tâches élémentaires	# Packs insérés
16	0.66 s	34140	4238
32	2.09 s	58061	7361
64	3.42 s	90383	12006
128	8.11 s	157605	21174

FIGURE B.29 – Information sur la création du DAG ($\theta = 4$).

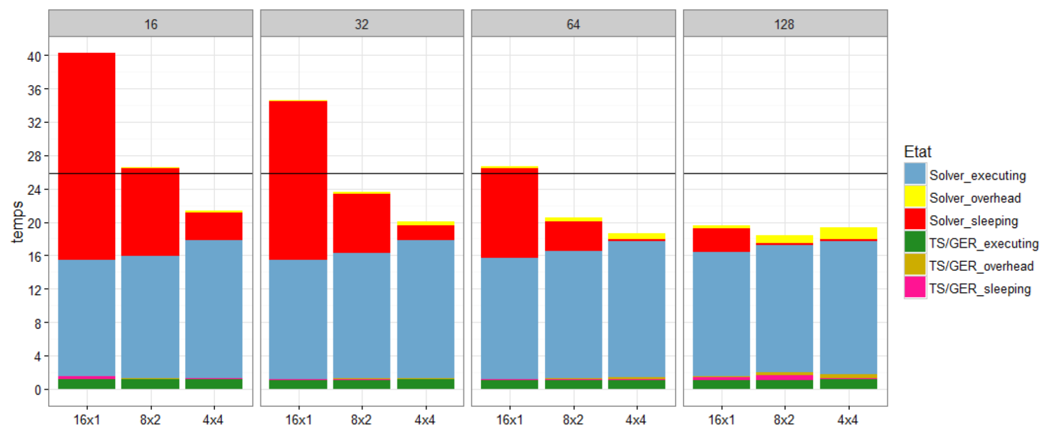


FIGURE B.30 – Performance ($\theta = 4$).

B.2.6 Détails pour θ=5

	τ_0	τ_1	τ_2	τ_3	τ_4	τ_5
Prop. Faces	0.02 %	2.00 %	3.05 %	5.01 %	13.26 %	76.67 %
Prop. Calc.	0.32 %	17.76 %	13.53 %	11.12 %	14.72 %	42.55 %

FIGURE B.31 – Répartition des faces par niveau temporel et proportion de calcul associée ($\theta = 5$).

#NCE	Tsub	# Tâches élémentaires	# Packs insérés
16	1.53 s	66225	8184
32	3.75 s	111154	13900
64	6.87 s	168285	22011
128	19.25 s	283068	37073

FIGURE B.32 – Information sur la création du DAG ($\theta = 5$).

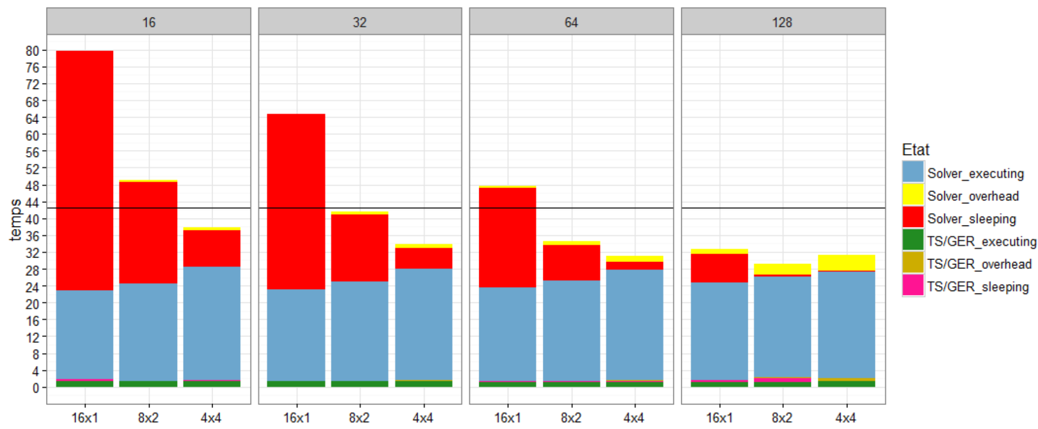


FIGURE B.33 – Performance ($\theta = 5$).

B.2.7 Détails pour θ=6

	τ_0	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6
Prop. Faces	0.02 %	2.00 %	3.06 %	5.01 %	13.28 %	22.76 %	53.87 %
Prop. Calc.	0.37 %	20.88 %	15.96 %	13.06 %	17.32 %	14.84 %	17.56 %

FIGURE B.34 – Répartition des faces par niveau temporel et proportion de calcul associée ($\theta = 6$).

#NCE	Tsub	# Tâches élémentaires	# Packs insérés
16	2.93 s	133537	16457
32	9.07 s	221246	27509
64	15.10 s	332048	43134
128	70.70 s	552012	71705

FIGURE B.35 – Information sur la création du DAG ($\theta = 6$).

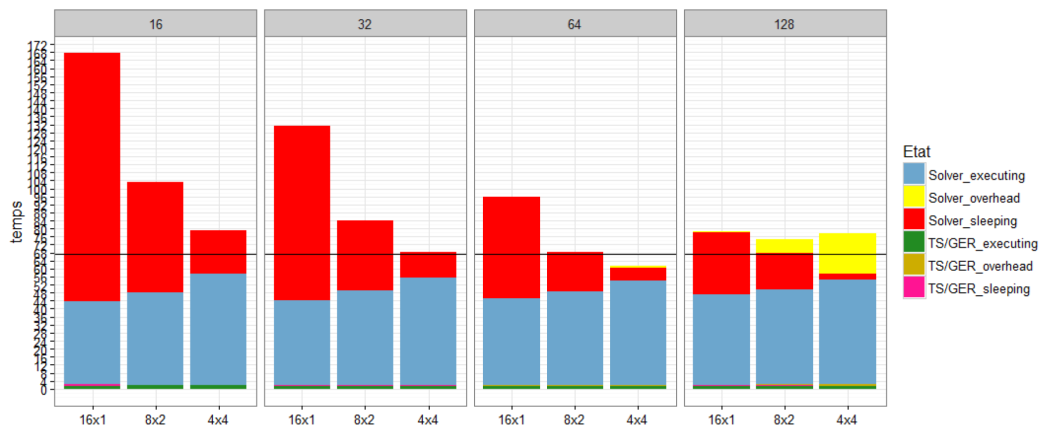


FIGURE B.36 – Performance ($\theta = 6$).

Annexe C

Performance en mémoire partagée pour la version en tâches pour 32 ECs (configuration de calcul (8×2), θ=4)

Cette annexe présente tout d'abord les traces d'exécution des deux ordonnancements possibles, sans et avec utilisation des priorités, pour le même cas de calcul sur 32 ECs ; la configuration de calcul est (8×2) et θ=4. Les tâches sont coloriées en fonction de la sous-itération qu'elles traitent et le rouge signifie que les workers sont *idle*.

La Figure C.3 donnent ensuite diverses informations concernant les ECs présentes dans le calcul : le nombre $|C^B|$ de cellules de bord, le nombre $|C^I|$ de cellules intérieures, le nombre $Nvois$ d'ECs voisines, les ECs voisines de l'EC considérée, le niveau temporel minimal $min(\tau)$ présent dans l'EC, le niveau temporel maximal $max(\tau)$ présent dans l'EC et enfin la priorité P pour le calcul utilisant l'ordonnanceur *prio*.

Enfin, on termine en donnant sur les 4 dernières figures diverses illustrations du DAG associé. Sur la Figure C.4, les tâches sont coloriées de la couleur de l'EC concernée. Il s'agit des mêmes couleurs que sur la Figure C.3. La Figure C.5 montre cette fois le DAG colorié par la couleur de la sous-itération concernée pour la tâche. Il s'agit du même code couleur que pour les traces. Les Figures C.6 et C.7 présentent le DAG colorié par ordre de terminaison des tâches en utilisant un dégradé du bleu au vert. En bleu, on a les premières tâches terminées et en vert les dernières tâches terminées. Le DAG de la Figure C.6 correspond à la trace de la Figure C.1 et le DAG de la Figure C.7 correspond à la trace de la Figure C.2. Au bas du DAG de la Figure C.7, on peut constater que certaines tâches insérées au tout début se sont terminées parmi les dernières.

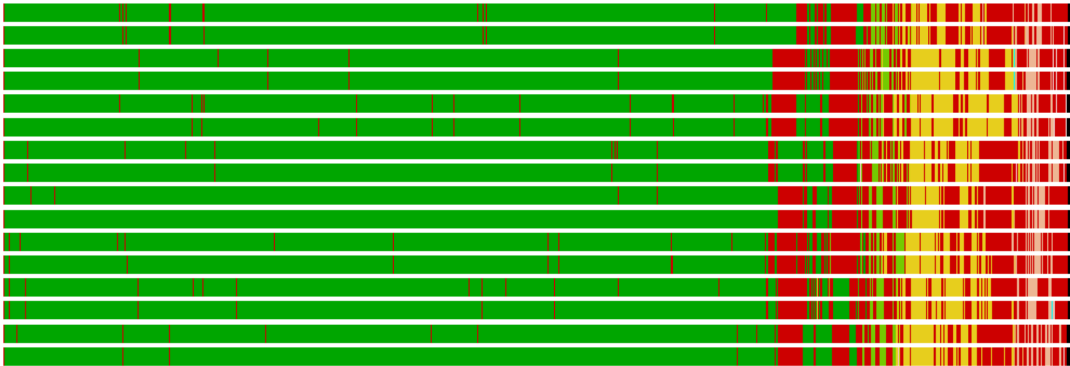


Figure C.1 – Trace sans utilisation des priorités (32 ECs, $\theta = 4$, (8×2) : t=15.55s).

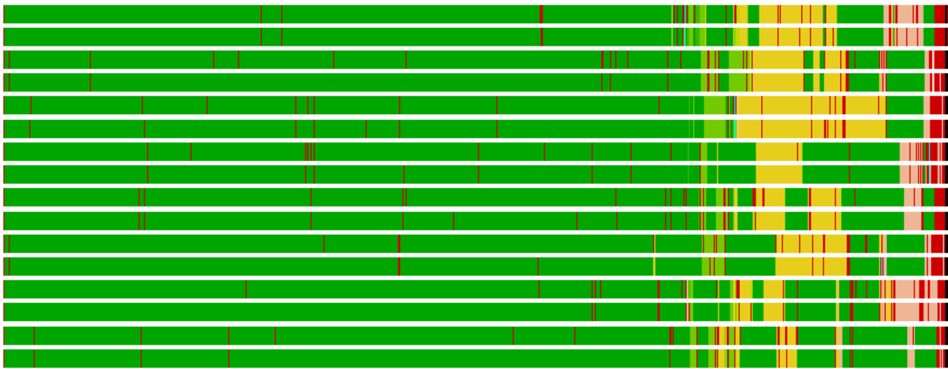


Figure C.2 – Trace avec utilisation des priorités (32 ECs, $\theta = 4$, (8×2) : t=13.89s).

EC	$\lvert C^B \rvert$	$\lvert C^I \rvert$	Nvois	Voisins	$min(\tau)$	$max(\tau)$	P
0	17731	304532	10	1 4 6 7 11 25 26 28 29 31	2	4	4
1	9624	312038	2	0 3	4	4	3
2	5105	317099	4	3 24 25 30	3	4	2
3	9635	312287	2	1 2	4	4	2
4	12925	319206	8	0 5 6 7 25 28 29 31	2	4	3
5	13023	318517	9	4 6 7 25 26 28 29 30 31	3	4	3
6	13140	337322	7	0 4 5 7 25 30 31	2	4	4
7	15020	336552	7	0 4 5 6 21 25 31	3	4	3
8	13936	315797	3	9 10 12	4	4	0
9	12579	318295	3	8 12 13	4	4	0
10	9472	319435	1	11	4	4	2
11	9472	322079	2	0 10	4	4	3
12	14305	316765	4	8 9 13 14	4	4	0
13	15796	316691	3	9 12 15	4	4	0
14	11858	325033	5	12 13 15 18 23	4	4	1
15	11802	325087	5	12 13 14 18 23	4	4	1
16	15394	314990	5	17 18 20 22 23	4	4	2
17	16765	327709	7	16 18 19 20 21 22 25	4	4	2
18	18832	313460	7	14 15 16 17 19 22 23	4	4	1
19	17281	329518	8	16 17 18 20 21 22 23 25	4	4	2
20	17450	325825	11	6 7 16 17 19 21 22 23 24 25 26	4	4	3
21	15991	324693	7	7 17 19 20 22 23 25	4	4	2
22	16644	324619	7	16 17 18 19 20 21 23	4	4	2
23	20422	322461	8	14 15 16 18 19 20 21 22	4	4	2
24	2695	323970	7	2 6 20 25 26 27 30	2	4	3
25	16715	301478	13	0 2 6 7 17 19 20 21 24 27 28 30 31	2	4	3
26	5088	314018	8	0 5 20 24 27 28 29 31	2	4	4
27	2084	338663	3	24 25 26	0	4	4
28	12486	321532	7	0 4 5 25 26 29 31	3	4	3
29	10086	315964	8	0 4 5 25 26 28 30 31	2	4	4
30	11836	343341	7	2 5 6 24 25 29 31	3	4	3
31	16473	340485	10	0 4 5 6 7 25 26 28 29 30	2	4	4

FIGURE C.3 – Informations détaillées pour les ECs.

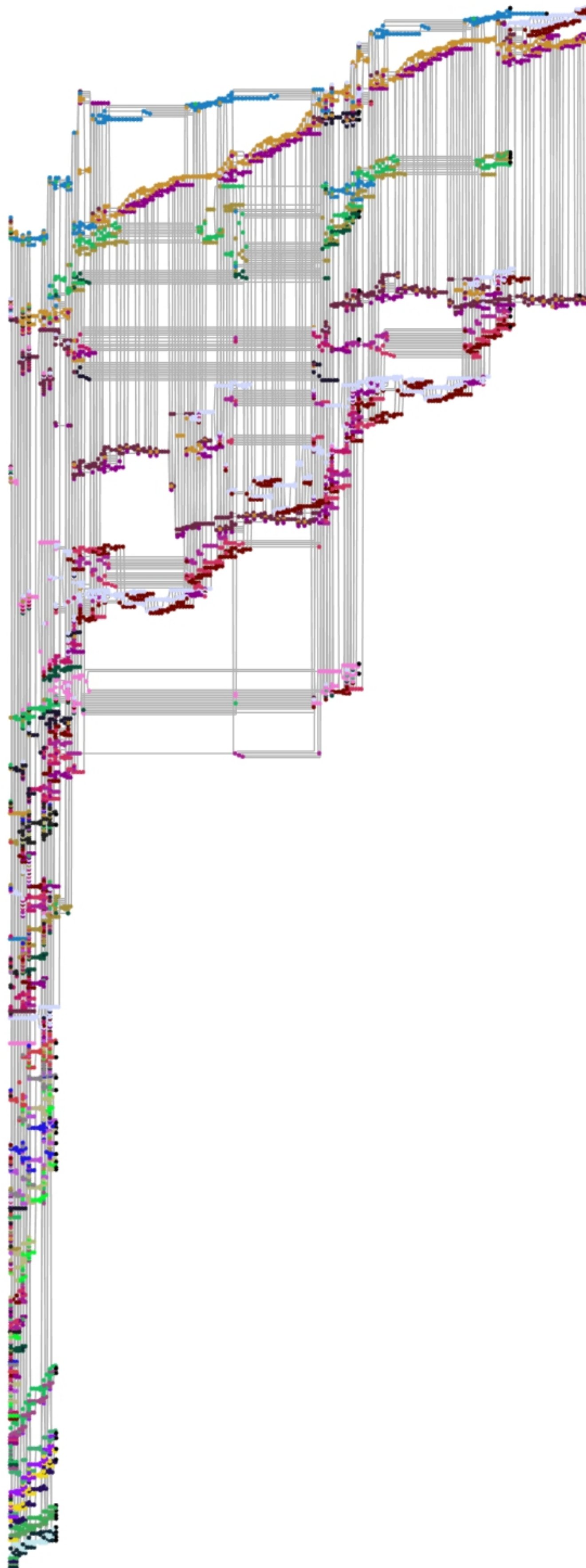


Figure C.4 – DAG colorié par EC.

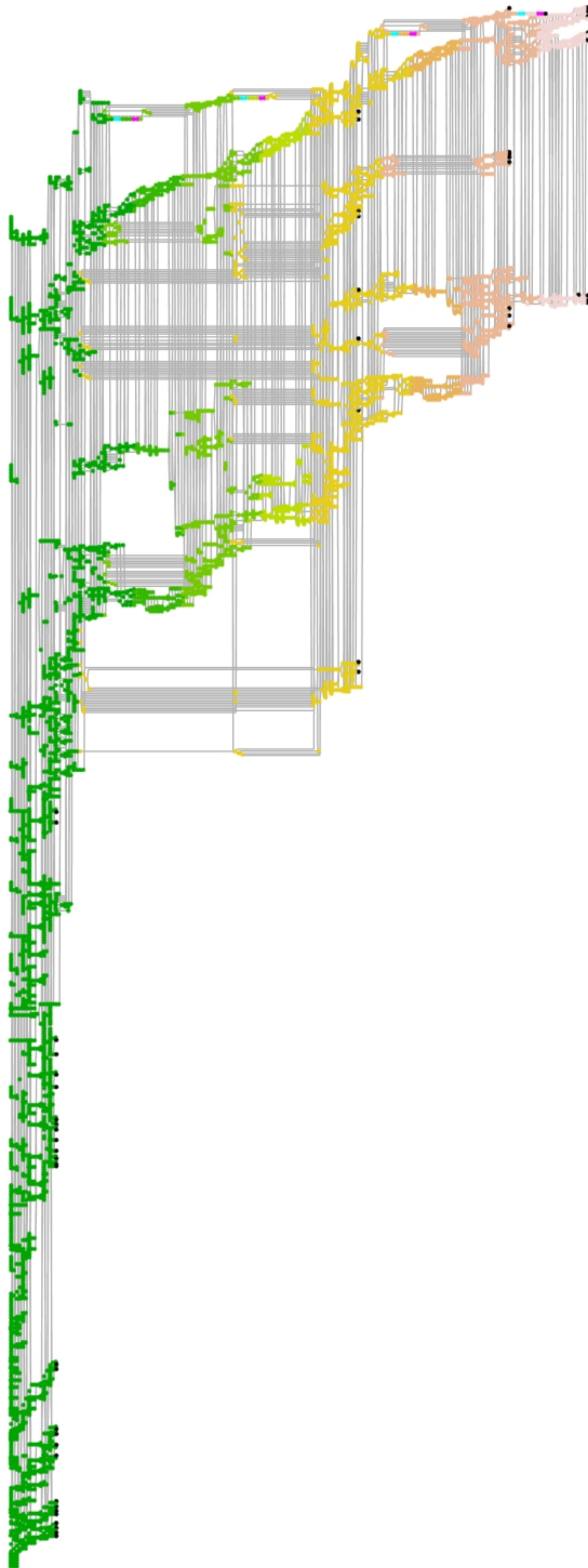


Figure C.5 – DAG colorié par numéro de sous-itération.

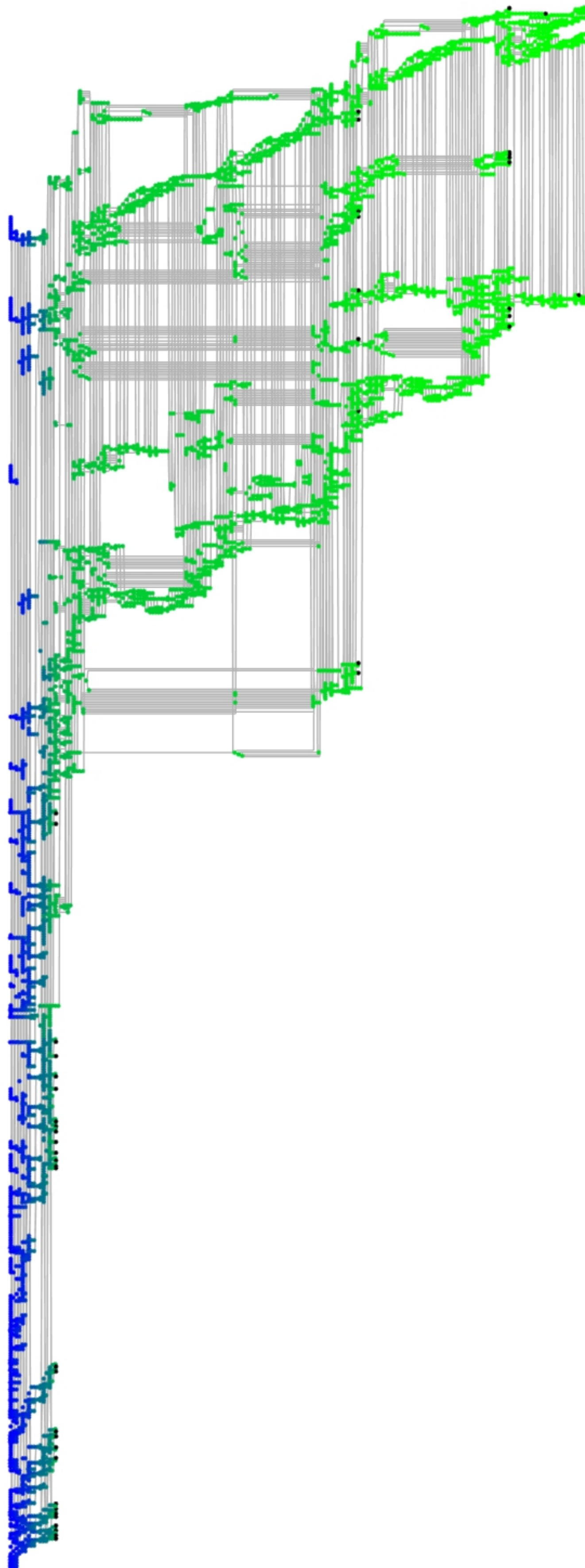


Figure C.6 – DAG colorié par ordre de lancement des tâches (sans priorité).

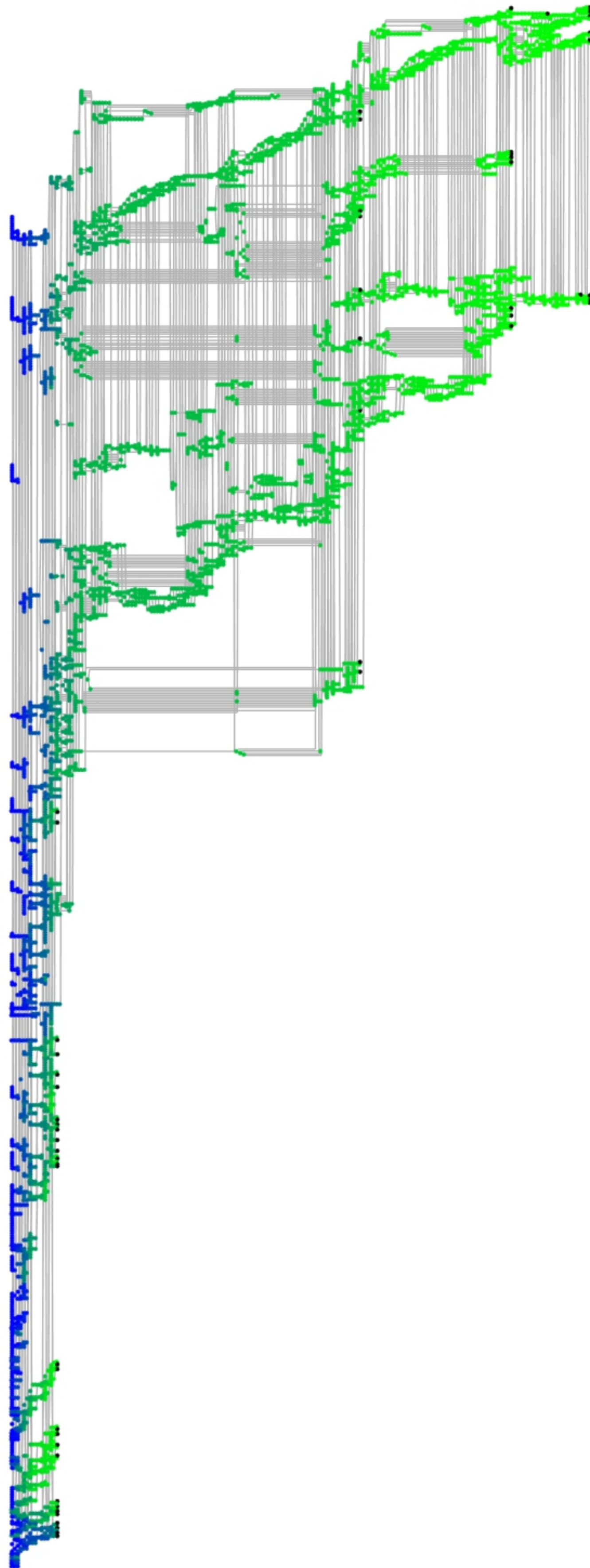


FIGURE C.7 – DAG colorié par ordre de lancement des tâches (avec priorité).