Algorithmes à grain fin et schémas numériques pour des simulations exascales de plasmas turbulents

Nicolas Bouzat

L'UNIVERSITÉ DE STRASBOURG

ÉCOLE DOCTORALE MATHÉMATIQUES, SCIENCES DE L'INFORMATION ET DE L'INGÉNIEUR
UMR 7501

THÈSE

Par **Nicolas BOUZAT**

pour obtenir le grade de : **Docteur de l'Université de Strasbourg**

Spécialité : **Informatique**

Algorithmes à grain fin et schémas numériques pour des simulations exascales de plasmas turbulents

---

Soutenue le : 17 Décembre 2018

Devant la commission d'examen composée de :

M. Nicolas CROUSEILLES	Chargé de recherche, Inria Rennes	Rapporteur
M. Philippe HELLUY	Professeur, Univ. de Strasbourg	Examinateur
M. Guillaume LATU	Ingénieur-chercheur, CEA Cadarache	Co-encadrant
M. Michel MEHRENBERGER	Professeur, Aix-Marseille Univ.	Directeur
M. Raymond NAMYST	Professeur, Univ. de Bordeaux	Rapporteur
M. Jean ROMAN	Professeur, Inria et Bordeaux INP	Co-directeur
Mme. Stéphanie SALMON	Professeur, Univ. de Reims	Examinateur

- 2018 -

Algorithmes à grain fin et schémas numériques pour des simulations exascales de plasmas turbulents

Résumé

Les architectures de calcul haute performance les plus récentes intègrent de plus en plus de nœuds de calcul qui contiennent eux-mêmes plus de cœurs. Les bus mémoires et les réseaux de communication sont soumis à un niveau d'utilisation critique. La programmation parallèle sur ces nouvelles machines nécessite de porter une attention particulière à ces problématiques pour l'écriture de nouveaux algorithmes. Nous analysons dans cette thèse un code de simulation de turbulences de plasma et proposons une refonte de la parallélisation de l'opérateur de gyromoyenne plus adapté en termes de distribution de données et bénéficiant d'un schéma de recouvrement calcul – communication efficace. Les optimisations permettent un gain vis-à-vis des coûts de communication et de l'empreinte mémoire. Nous étudions également les possibilités d'évolution de ce code à travers la conception d'un prototype utilisant un modèle programmation par tâche et un schéma de communication asynchrone adapté. Cela permet d'atteindre un meilleur équilibrage de charge afin de maximiser le temps de calcul et de minimiser les communications entre processus. Un maillage réduit adaptatif en espace est proposé, diminuant le nombre de points sans pour autant perdre en précision, mais ajoutant de fait une couche supplémentaire de complexité. Ce prototype explore également une distribution de données différente ainsi qu'un maillage en géométrie complexe adapté aux nouvelles configurations des tokamaks. Les performances de différentes optimisations sont étudiées et comparées avec le code préexistant et un cas dimensionnant sur un grand nombre de cœurs est présenté.

Mots clés :

Parallélisme, schémas numériques, programmation par tâches, recouvrement calcul-communication, ordonnancement, maillage réduit

Discipline :

Informatique

IPL C2S@Exa
Équipe projet Inria TONUS et HiePACS
CEA Cadarache - IRFM - Équipe Gysela

Fine grain algorithms and numerical schemes for exascale simulations of turbulent plasmas

Abstract

Recent high performance computing architectures come with more and more cores on a greater number of computational nodes. Memory buses and communication networks are facing critical levels of use. Programming parallel codes for those architectures requires to put the emphasize on those matters while writing tailored algorithms. In this thesis, a plasma turbulence simulation code is analyzed and its parallelization is overhauled. The gyroaverage operator benefits from a new algorithm that is better suited with regard to its data distribution and that uses a computation – communication overlapping scheme. Those optimizations lead to an improvement by reducing both execution times and memory footprint. We also study new designs for the code by developing a prototype based on task programming model and an asynchronous communication scheme. It allows us to reach a better load balancing and thus to achieve better execution times by minimizing communication overheads. A new reduced mesh is introduced, shrinking the overall mesh size while keeping the same numerical accuracy but at the expense of more complex operators. This prototype also uses a new data distribution and twists the mesh to adapt to the complex geometries of modern tokamak reactors. Performance of the different optimizations is studied and compared to that of the current code. A case scaling on a large number of cores is given.

Keywords :

Distributed computing, numerical schemes, task-based programming, computation-communication overlap, tasks and communication ordering, reduced grid

Discipline :

Computer science

IPL C2S@Exa
Équipe projet Inria TONUS et HiePACS
CEA Cadarache - IRFM - Équipe Gysela

Table des matières

Introduction

Le domaine d'expertise de cette thèse est celui du calcul haute performance (HPC) appliqué à la fusion nucléaire par confinement magnétique. En particulier, mes travaux portent sur la simulation des turbulences du plasma évoluant à l'intérieur des réacteurs de type Tokamak. Les simulations issues de la modélisation permettent d'approfondir notre connaissance des phénomènes se déroulant dans ces dispositifs expérimentaux, mais nécessitent une puissance de calcul significative du fait de leur complexité. Certains progrès sur les plasmas magnétiquement confinés [32] n'auraient pas été possibles sans la génération actuelle de supercalculateurs dite *petascale*. Les objectifs de cette thèse sont la recherche de nouveaux schémas numériques et l'optimisation de la parallélisation d'un code de simulation pour s'adapter aux évolutions matérielles de la future génération de supercalculateurs dite pré-*exascale* [27, 26]. Ces calculateurs disposent de beaucoup plus de nœuds de calcul que leurs prédécesseurs et chaque nœud dispose également de plus de cœurs. Dans la programmation parallèle, l'accent est actuellement porté sur la scalabilité et la localité mémoire, ainsi que sur l'équilibrage de charge au sein d'un nœud et sur la maîtrise des schémas de communication. Les applications doivent mettre en place des distributions de données évitant les redondances et maximisant la localité [6] afin de réduire la quantité de communications et d'éviter les communications collectives. Les communications inévitables devront, autant que faire se peut, être effectuées en simultané avec des calculs (recouvrement calcul – communication) [66]. Cela peut-être facilité par l'utilisation de grains de parallélisation variables (accessible avec un modèle de programmation par tâches) et des primitives de communication non bloquantes et si possible asynchrones [60]. La programmation par tâches constitue également un bon outil pour faciliter l'équilibrage de charge de calcul entre de nombreux cœurs [17].

Le code Gysela [34], développé au CEA Cadarache en collaboration avec Inria et d'autres partenaires, a servi de socle à ces travaux de thèse. Gysela a pour objet la simulation des turbulences du plasma dans des tokamaks à section circulaire en utilisant un modèle gyrocinétique semi–Lagrangien 5D. Il dispose d'une parallélisation efficace sur les supercalculateurs *petascales* grâce à un paradigme de programmation mixte MPI/OpenMP. Néanmoins, dû aux limitations d'extensibilité mémoire et au manque d'efficacité de certains algorithmes lors du passage à l'échelle (notamment lié aux communications qui représentent près de la moitié du temps d'exécution dans certaines configurations), le code Gysela n'est pas capable de s'exécuter en production sur plus de 32 000 cœurs sur des domaines de tailles pertinentes. Le but de ce travail de thèse est de proposer et d'implémenter des solutions afin de résorber les différents goulets d'étranglement en introduisant un parallélisme à grain plus fin et de nouveaux schémas numériques moins coûteux en mémoire et en communication. De plus, avec l'introduction d'une physique liée à une nouvelle catégorie de particules ajoutée dans le code (électrons cinétiques), la taille des maillages

augmente fortement, renforçant les problématiques liées à la distribution des données. Ces évolutions sont donc effectuées dans l'optique de permettre au code GYSELA d'utiliser au mieux les machines pré-*exascales* pour de nouveaux problèmes plus complexes. Ainsi, les travaux présentés ici se focalisent sur différents aspects du code : l'optimisation de l'opérateur de gyromoyenne dans le code existant, l'étude d'un nouveau maillage en géométrie réelle et l'implémentation d'un prototype permettant d'évaluer de nouvelles solutions algorithmiques et optimisations.

Le premier chapitre permet d'introduire les différents concepts et problématiques abordés dans cette thèse, notamment les deux modèles physiques et mathématiques utilisés pour l'implémentation des codes de simulation du plasma étudiés. Les évolutions architecturales des calculateurs de génération pré-*exascale* sont ensuite présentées ainsi que les problèmes qu'elles soulèvent. Puis, un aperçu de l'état de l'art des techniques et méthodes de programmation distribuée adaptées à ces machines est dépeint. Pour clore ce chapitre, nous effectuerons un bilan des performances du code GYSELA afin de définir précieusement où doivent être portés les efforts de parallélisation et d'optimisation et de dégager les objectifs de cette thèse.

Le chapitre 2 présentera l'opérateur de gyromoyenne [63] et fera un état des lieux de son implémentation et de ses performances actuelles avant d'entrer, dans le chapitre 3, dans les détails de la nouvelle implémentation proposée. L'opérateur de gyromoyenne, basée sur les travaux de Fabien Rozar et Christophe Steiner, nécessite actuellement de coûteuses transpositions de l'ensemble des données. L'adaptation de l'opérateur pour la distribution de données de GYSELA et son optimisation se fait en ajoutant une zone tampon, appelée *halo*, autour du sous-domaine local (la taille du *stencil* de l'opérateur est fixe). De plus, cet opérateur ne combine que deux dimensions et permet donc une parallélisation facile dans les autres dimensions. Calculs et communications peuvent alors être subdivisés en blocs et un algorithme de recouvrement calcul – communication est mis en place et implémenté au moyen de *threads* OpenMP. Ces différentes optimisations permettent une amélioration significative du temps d'exécution de l'opérateur. Le chapitre 4 traite des évolutions de l'opérateur à la suite de nouvelles contraintes sur la modélisation et propose une évolution en profondeur de la structure du code. Un nouveau maillage de l'espace est proposé. Le maillage du plan poloïdal (transverse aux lignes de champs) y est réduit afin de limiter la sur-résolution au centre du plan et ainsi diminuer l'empreinte mémoire des données. De plus, il permet un mapping mettant en correspondance un espace logique avec la géométrie réelle permettant de simuler des formes de tokamak plus proches des nouvelles machines en production depuis quelques années.

Un prototype dénommé GYSELA++ est présenté au chapitre 5. Il doit permettre de tester le nouveau maillage et d'évaluer de nouvelles solutions pour pallier les goulets d'étranglement de GYSELA (quantité de communication, attente de synchronisation, empreinte mémoire). Il introduit une nouvelle distribution de données, un modèle de communication non bloquant asynchrone et utilise un modèle de programmation par tâches afin de faciliter l'équilibrage de la charge de calcul et le recouvrement calcul – communication. Il utilise également le nouveau maillage qui vise à faciliter la mise en place de modèles physiques plus complexes tels que celui du plasma de bord, ainsi qu'une nouvelle structure de données formée de tuiles facilitant la parallélisation des calculs et améliorant la localité des données.

Les choix de mise en œuvre et les résultats relatifs aux différents opérateurs du code

sont développés dans le chapitre 6. L'opérateur d'advection a été intégralement réécrit. La section 6.1 détaille sa mise en œuvre par des algorithmes à base de tâches et la gestion des communications découlant de la distribution de données. Dans GYSELA, les données subissent un changement de distribution (transposition) de sorte qu'à chaque appel à un sous-opérateur d'advection, les dimensions concernées soient localement intégralement disponibles. À l'inverse, dans GYSELA++ les données sont distribuées en 5D, générant des schémas de communication plus complexes. La section 6.2 se concentre sur l'opérateur de Poisson. Ce dernier est issu d'une version précédente de GYSELA et n'a pas fait l'objet d'améliorations autres qu'une adaptation au maillage de GYSELA++ et une mise en œuvre sous forme de tâches.

Un dernier chapitre 7 propose une comparaison entre GYSELA++ et le code GYSELA de référence en évaluant les évolutions en termes de temps d'exécution en section 7.1. Une étude de scalabilité sur un maillage conséquent et un grand nombre de cœurs est également décrite. La section 7.2 discute d'un certain nombre d'améliorations possibles concernant l'entrelacement des calculs, l'ordonnancement des tâches, la *bufferisation* des communications et la gestion mémoire des halos. Nous concluons par un résumé des résultats obtenus lors de cette thèse et sur les perspectives ouvertes par ces travaux, notamment concernant l'évolution du code GYSELA.

Les travaux de cette thèse ont été effectués dans le cadre d'une collaboration Inria – CEA, entre les équipes-projet HiePACS (Bordeaux) et TONUS (Strasbourg) et l'Institut de Recherche sur la Fusion par confinement Magnétique (Cadarache). Ils ont reçu un financement dans le cadre de l'initiative IPL C2S@Exa [1], qui est un projet financé par Inria. Ce programme a rassemblé un certain nombre d'acteurs intéressés par les problématiques liées aux futures architectures exascales.

1. `https://www-sop.inria.fr/c2s_at_exa/`

Chapitre 1

État de l'art et positionnement

Sommaire

La fusion nucléaire est considérée par certains comme une énergie d'avenir. Plus propre et plus sûre que la fission nucléaire, la quantité d'énergie qu'elle pourrait fournir par rapport à la fission pour des réacteurs de taille équivalente est bien supérieure tout en consommant moins de combustible. De nombreux défis techniques restent néanmoins à relever afin d'obtenir un dispositif d'exploitation de fusion nucléaire énergétiquement rentable. Pour ce faire, construire des réacteurs et effectuer des expérimentations est essentiel, mais extrêmement coûteux. De plus, cette approche ne pourra donner de résultats sans une compréhension fine de la physique des plasmas générés par les réacteurs. Le calcul haute performance est un domaine à l'intersection de l'informatique, des mathématiques numériques et de nombreux domaines applicatifs. Les recherches qui y sont menées utilisent des supercalculateurs afin de résoudre des problèmes complexes qui ne pourraient l'être par des moyens plus conventionnels. Ces calculateurs possèdent des architectures complexes qui nécessitent une expertise spécifique afin d'en tirer le plus de puissance calculatoire possible.

Les domaines de la fusion nucléaire et du calcul haute performance sont étroitement liés. En effet, il est nécessaire pour les physiciens d'avoir recours à la simulation pour comprendre le comportement des plasmas dans les réacteurs à fusion nucléaire. Le système

à modéliser est en effet complexe et de taille conséquente. Il doit de plus garantir une précision élevée afin de suivre un comportement physiquement correct. C'est pourquoi des calculateurs parmi les plus puissants au monde sont utilisés dans le cadre de la recherche sur la fusion nucléaire. De nombreux codes de simulation sont développés et optimisés pour ces supercalculateurs afin de faire évoluer l'architecture des réacteurs et d'arriver à un dispositif de fusion nucléaire commercialement exploitable.

Nous détaillerons dans un premier temps la réaction de fusion nucléaire ainsi qu'une des pratiques employées afin de la mettre en œuvre : le tokamak (section 1.1.1). Nous verrons par la suite différents modèles physiques et mathématiques utilisés pour modéliser les plasmas générés dans les réacteurs de type tokamak (sections 1.1.2 à 1.1.5). L'architecture typique des supercalculateurs actuels sera présentée en section 1.2.1 ainsi qu'un certain nombre de techniques de programmation parallèles (sections 1.2.2 à 1.2.4). Nous nous intéresserons ensuite au code de simulation des turbulences du plasma Gysela dont le fonctionnement sera détaillé dans les grandes lignes. Pour finir, les travaux présentés dans ce manuscrit seront exposés et mis en regard de l'état de l'art actuel (section 1.4).

1.1 Cadre physique et modélisation

La fusion nucléaire est un domaine faisant l'objet de nombreux programmes de recherche. En effet, une réaction de fusion nucléaire contrôlée offrirait une source d'énergie au rendement inégalé, pérenne et sûre. De nombreux réacteurs expérimentaux de type tokamak (JET [1], EAST [2], ITER [3]) sont en cours d'exploitation ou de construction afin d'arriver à maîtriser cette réaction. Similairement, des codes de simulations (Gene [33], Gysela [35], GT5D [43] ...) sont mis en œuvre afin d'améliorer les connaissances théoriques sur les plasmas générés par ces réacteurs et de pouvoir améliorer leur design. Plusieurs théories sont utilisées pour modéliser ces plasmas à différentes échelles d'espace et de temps.

1.1.1 Fusion nucléaire et tokamaks

La fusion nucléaire est une énergie plus propre que la fission nucléaire. En effet, les éléments radioactifs entrant en jeu sont moins nocifs. Le tritium possède une demi-vie d'environ douze ans pour une énergie de désintégration de quelques dizaines de kiloélectronvolts contre une demi-vie de plusieurs dizaines de milliers d'années pour les déchets de plutonium et d'uranium avec une énergie de désintégration de plusieurs mégaélectronvolts. Contrairement à la fission nucléaire, bien maîtrisée, la fusion nucléaire civile n'en est encore qu'au stade expérimental. En effet, la réaction de fusion ne peut se dérouler que dans des conditions extrêmes (au cœur du soleil par exemple). C'est pourquoi elle présente également des avantages en termes de sécurité : cette réaction nécessitant un apport conséquent en énergie, en cas de problème au niveau de l'enceinte de confinement, l'énergie serait dissipée rapidement et la réaction s'arrêterait spontanément sans risque d'emballement. De plus, la fusion nucléaire est potentiellement plus efficiente que la fission et nécessite beaucoup moins de combustible à quantité d'énergie produite égale.

Les réactions de fusion nucléaire consistent à faire fusionner des atomes légers en les chauffant à très haute température. Ils forment alors un plasma ionisé ; la vitesse atteinte

1. http://www.ccfe.ac.uk
2. http://english.ipp.cas.cn/rh/east/
3. https://www.iter.org/fr

par les ions leur permet de vaincre la barrière colombienne et de fusionner. Ils forment alors un atome plus lourd avec émission d'une ou plusieurs particules (photons, neutrons, protons) suivant le type de réaction. La perte de masse entraîne un dégagement d'énergie sous forme cinétique ($E = mc^2$) répartie entre l'atome créé et les particules émises. Cette réaction est exoénergétique, c'est-à-dire qu'elle produit plus d'énergie qu'il n'a fallu en fournir au système pour qu'elle ait lieu. Ces réactions se déroulent naturellement au cœur du soleil où les conditions de température, de densité et de pression sont propices à leur apparition spontanée.

Sur Terre, la réaction la plus accessible est celle des isotopes de l'hydrogène, le deutérium $^{2}_{1}\mathrm{H}$ et le tritium $^{3}_{1}\mathrm{H}$ (voir Figure 1.1) qui fusionnent pour donner un ion d'hélium $^{5}_{2}\mathrm{He}$ instable. Celui-ci se dégrade en un ion d'hélium $^{4}_{2}\mathrm{He}$ énergétique et libère un neutron dont l'énergie sera celle exploitée pour produire de l'électricité. Cette réaction est celle privilégiée dans les réacteurs expérimentaux à l'heure actuelle, car plus les noyaux sont lourds, plus leur fusion requiert un apport d'énergie élevé. Bien que la température nécessaire pour engendrer cette réaction à pression atmosphérique soit déjà de l'ordre de la centaine de millions de degrés Kelvin, c'est-à-dire une température plusieurs fois supérieure à celle du cœur du soleil, elle serait encore plus élevée pour d'autres réactions. La fusion deutérium–tritium est donc préférée car elle est la plus facilement réalisable sur Terre. De plus, elle présente un facteur d'amplification attractif ; cela représente la proportion d'énergie créée par rapport à l'énergie apportée pour démarrer la réaction. Elle peut être obtenue ponctuellement (bombe H, fusion inertielle) en appliquant une température et une densité suffisante à une microcapsule de combustible par différents procédés (laser, striction magnétique). Le procédé est ensuite répété de façon cyclique à la manière d'un moteur thermique ; c'est le principe de la fusion inertielle.

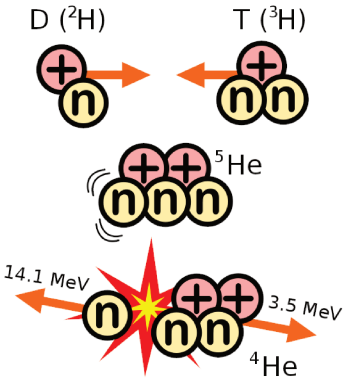
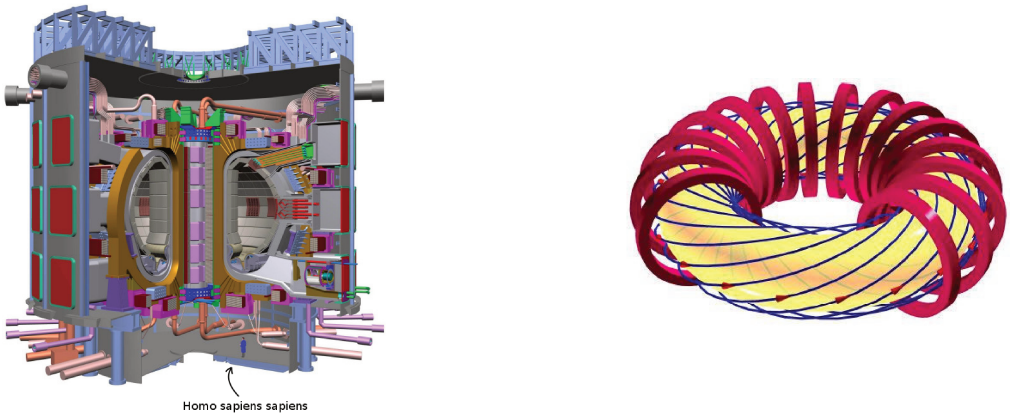


FIGURE 1.1 – Réaction de fusion entre deutérium et tritium [4].

La fusion par confinement magnétique vise à obtenir une réaction de fusion continue auto-entretenue. Il est donc nécessaire de confiner cette réaction magnétiquement sans aucun contact avec les parois. À ces températures, le combustible est sous forme de plasma ionisé : les atomes sont suffisamment excités pour que les électrons s'échappent de leur orbite et naviguent librement dans le plasma. Ainsi les particules se déplaçant dans le plasma sont des électrons libres et des ions ; ce sont des particules chargées sensibles aux champs électromagnétiques. Ainsi, le deutérium et le tritium sont portés à très haute température afin de créer un plasma ionisé. Les particules générées par la fusion contribuent

4. par I. Panoptik, CC BY 2.5, `https://commons.wikimedia.org/w/index.php?curid=2510231`

à maintenir le plasma à une température suffisante pour entretenir la réaction jusqu'à épuisement des combustibles. Une des solutions étudiées pour réaliser des plasmas de fusion magnétiquement confinés est le tokamak. Ce réacteur consiste en une enceinte torique entourée de bobines générant le champ magnétique nécessaire au confinement du plasma (voir Figure 1.2b). Le réacteur ITER (voir Figure 1.2a) sera bientôt le plus gros tokamak au monde. Il est actuellement en construction sur le site de Cadarache en France et devrait être le premier réacteur expérimental à atteindre un facteur d'amplification supérieur à 1. C'est la modélisation de ce type de réacteur que visent les travaux de cette thèse.



(a) Vue en coupe du réacteur ITER.

(b) Représentation du champ magnétique (lignes bleues), des bobines de confinement (rouge) et du plasma (jaune).

FIGURE 1.2 – Exemples de tokamaks.

En effet, de nombreux défis restent à relever afin de totalement comprendre et maîtriser le comportement du plasma dans un tokamak. Une de ses problématiques est l'apparition des turbulences qui fragilisent le confinement, entraînant une extinction de la réaction [12]. C'est pourquoi, il est nécessaire de comprendre comment ces turbulences se développent, évoluent au sein du plasma et impactent le champ électromagnétique. C'est dans ce but que sont utilisées les simulations numériques qui modélisent l'évolution du plasma et de ses turbulences. Plusieurs descriptions du système sont possibles, permettant de décrire différentes échelles de temps et d'espace à différents coûts.

1.1.2 Modèle particulaire

La modélisation la plus précise du système consiste à considérer chacune des particules évoluant dans le plasma et à simuler son évolution grâce à la dynamique newtonienne. Les particules sont représentées par une position $\mathbf{x}$, une vitesse $\mathbf{v}$, une masse m ainsi qu'une charge q. Dans un cas non relativiste, l'évolution des particules est régie par le principe fondamental de la dynamique

$$\frac{\mathrm{d}m\mathbf{v}}{\mathrm{d}t} = \sum F_{ext}.$$

Les forces exercées sur les particules se composent essentiellement de la force de Lorentz engendrée par les fluctuations du champ électromagnétique. Les autres forces telles que le poids sont négligeables. Dans ce modèle, le nombre de particules à simuler afin de

reproduire fidèlement l'évolution d'un plasma est de l'ordre de 10^{10} pour des tokamaks de plusieurs mètres d'envergure. Cela représente un coût calculatoire trop élevé pour être utilisé dans une simulation. Il faut donc effectuer des approximations pour proposer des modèles qui ont un coût calculatoire raisonnable tout en restant le plus précis possible. Une analyse macroscopique du système utilisant une représentation sous forme de fluide est possible. Ceux-ci sont caractérisés par leur densité, leur vitesse et leur énergie. Cette approche est une bonne approximation lorsque les particules sont proches d'un équilibre thermodynamique. Le modèle cinétique se base sur une description statistique du système et permet d'atteindre une bonne approximation de l'évolution des turbulences, même loin d'un état d'équilibre. C'est à cette approche que nous nous intéressons ici.

1.1.3 Modèle cinétique

Dans le modèle cinétique (représentation Eulerienne), les particules du plasma sont représentées par une fonction de distribution $f(\mathbf{x}, \mathbf{v}, t)$ pour chaque espèce de particule. $\mathbf{x}$ correspond aux trois coordonnées d'espace, $\mathbf{v}$ aux trois coordonnées de vitesse et t au temps. Pour une espèce donnée, la valeur de sa fonction de distribution f en un point $(\mathbf{x}, \mathbf{v}, t)$ donne la probabilité de trouver une particule de l'élément correspondant possédant ces coordonnées. Ainsi, la quantité $f(\mathbf{x}, \mathbf{v}, t)d\mathbf{x}d\mathbf{v}$ représente la quantité moyenne de particules localisées en $[\mathbf{x} - \frac{d\mathbf{x}}{2}, \mathbf{x} + \frac{d\mathbf{x}}{2}]$ et ayant une vitesse comprise dans $[\mathbf{v} - \frac{d\mathbf{v}}{2}, \mathbf{v} + \frac{d\mathbf{v}}{2}]$ à l'instant t. Ces particules évoluent dans l'espace des phases $(\mathbf{x}, \mathbf{v})$ soumises à un champ de force $\mathbf{F}$. Seules les forces électromagnétiques sont prises en compte, la gravité et les autres interactions étant négligeables. En notant $\mathbf{E}$ le champ électrique et $\mathbf{B}$ le champ magnétique, q et m la charge et la masse d'une particule d'une espèce donnée, le champ de force s'écrit

$$\mathbf{F} = \frac{q}{m}(\mathbf{E} + \mathbf{v} \times \mathbf{B}). \tag{1.1}$$

L'équation d'évolution des particules dans le plasma utilisée dans ce modèle est l'équation de Vlasov qui se formule

$$\frac{\partial f}{\partial t} + \mathbf{v} \cdot \nabla_{\mathbf{x}} f + \mathbf{F} \cdot \nabla_{\mathbf{v}} f = 0 \tag{1.2}$$

avec $\nabla_{\mathbf{x}} f$ (respectivement $\nabla_{\mathbf{v}} f$) le gradient de f par rapport aux coordonnées de position (respectivement coordonnées de vitesse). D'autres termes peuvent intervenir dans l'équation de Vlasov, notamment des sources de chaleur ou la prise en compte de collisions entre particules. Ils apparaissent alors dans le membre de droite et l'équation de Vlasov ainsi augmentée porte le nom d'équation de Boltzmann. Nous nous limiterons dans ces travaux à l'équation de Vlasov sans source ni collisions. Cette simplification ne remet pas en cause la pertinence de l'étude dans le cas Boltzmann.

Le déplacement des particules chargées génère lui-même un champ électromagnétique. Celui-ci peut être exprimé en fonction des caractéristiques des particules ainsi que de la fonction de distribution grâce aux équations de Maxwell. En prenant également en compte le champ magnétique constant imposé par les bobines, les équations de Vlasov et Maxwell forment un système dit auto-cohérent ; il est possible de faire évoluer la fonction de distribution des particules à partir de ces équations [20].

Le modèle cinétique est donc un modèle en six dimensions, et de ce fait présente un coût calculatoire élevé. De plus, les structures turbulentes dont l'étude est l'objet de ces simulations nécessitent un maillage très raffiné pour être correctement représenté. Finalement,

la dynamique des particules étant extrêmement rapide, le pas de temps utilisé doit être suffisamment petit pour assurer la précision requise. C'est pourquoi le modèle cinétique est délaissé au profit du modèle gyrocinétique qui permet d'abaisser ces coûts.

1.1.4 Modèle gyrocinétique

Le modèle gyrocinétique est actuellement utilisé pour de nombreux codes de simulation des turbulences du plasma (GYSELA [35], GT5D [43]) et fait l'objet de recherches actives [11, 50]. On sait que sous un fort champ magnétique, les particules se déplacent le long d'une trajectoire proche des lignes de champ en effectuant un mouvement de giration hélicoïdal tel que montré en Figure 1.3 [51]. On appelle cette trajectoire «centre-guide».

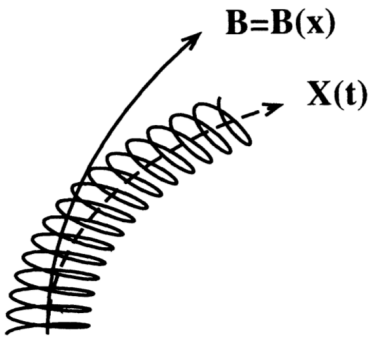


FIGURE 1.3 – Ligne de champ magnétique $\mathbf{B}(\mathbf{x})$ et mouvement de giration des particules du plasma autour des centres-guides $\mathbf{X}(t)$.

Le mouvement de giration des particules autour des centres-guides est caractérisé par deux grandeurs : son rayon d'enroulement autour des centres-guides $\rho_{\mathcal{L}}$, appelé rayon de Larmor, et sa fréquence Ω_{c}, appelée fréquence cyclotronique. Dans le cas d'un fort champ magnétique non-uniforme $\mathbf{B}$ comme dans le cas des tokamaks, la fréquence cyclotronique est supposée très élevée. Si de plus, les gradients spatiaux et temporels de $\mathbf{B}$ sont faibles devant Ω_{c}, alors l'étude des centres-guides constitue une bonne approximation de la dynamique des particules du plasma. Considérer les centres-guides revient à moyenner le mouvement rapide de giration des particules, ce qui est envisageable, car la fréquence cyclotronique est d'un ordre de grandeur bien supérieur à la dynamique parallèle du système (mouvement des particules le long des lignes de champs). Nous nous intéressons donc désormais à l'étude de la fonction de distribution des centres-guides $\bar{f}$ qui est reliée à la fonction de distribution des particules f par l'opérateur de gyromoyenne que nous détaillerons au chapitre 2. De plus, seuls les ions sont modélisés (une fonction de distribution par espèce ionique) et les électrons sont considérés comme ayant un comportement adiabatique (sans échange de chaleur). Les différentes équations peuvent être adaptées à la fonction de distribution des centres-guides pour obtenir leur version gyrocinétique. Le lecteur est renvoyé vers les publications suivantes [10, 30, 39] pour plus de détails.

Le système 6D est donc transformé en un problème 5D dont les variables sont les trois coordonnées spatiales $\mathbf{x}$, la vitesse parallèle aux lignes de champ $v_{\|}$ et μ le moment magnétique défini par

$$\mu = \frac{m v_{\perp}^2}{2B_0(r)} \tag{1.3}$$

avec $v_\perp$ la vitesse de giration autour des centres-guides dans le plan orthogonal au champ magnétique. Le moment magnétique est considéré comme un invariant adiabatique, c'est-à-dire qu'il ne joue le rôle que de paramètre dans les équations gyrocinétiques. Les équations de Maxwell sont substituées par une équation plus forte qui les englobe : l'équation de quasineutralité. Ainsi, le procédé d'évolution du plasma est donné par la Figure 1.4. Les coordonnées spatiales utilisées, r, θ et φ décrivent un tore (voir section 1.3.1 pour le détail de la géométrie). Le potentiel électrique est calculé à partir de la fonction de distribution des particules. La fonction de distribution des centres-guides est obtenue par application de l'opérateur de gyromoyenne à la fonction de distribution des particules. Le champ électrique, dérivé du potentiel, et le champ magnétique sont utilisés pour effectuer l'advection des centres-guides grâce aux équations de Boltzmann. Puis la gyromoyenne est réappliquée pour obtenir une nouvelle distribution de particules.

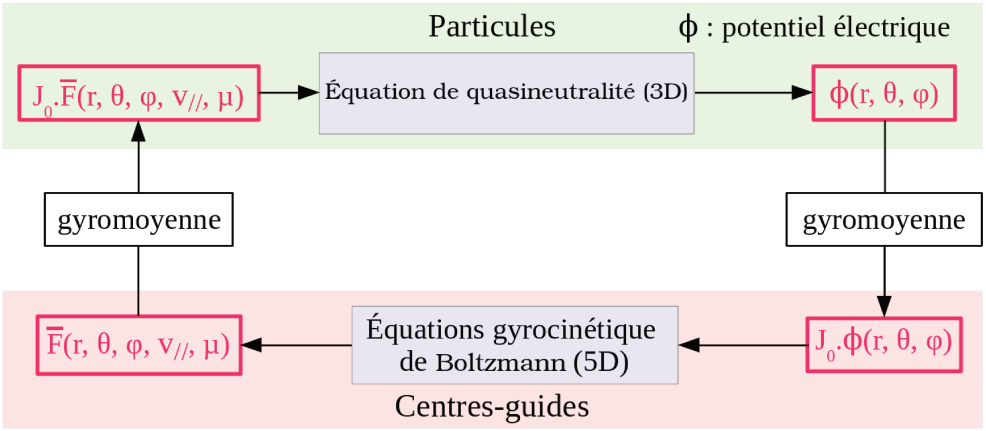


FIGURE 1.4 – Schéma algorithmique global d'une simulation gyrocinétique (voir section 1.3 pour le détail des coordonnées).

Le modèle 5D gyrocinétique offre une bonne approximation du mouvement des particules en s'intéressant à la projection de leur mouvement sur les centres-guides. De plus, il permet de réduire considérablement le pas de temps nécessaire afin de conserver une évolution du système physiquement cohérente. De ce fait, il représente le modèle offrant un bon compromis entre complexité calculatoire et précision physique et numérique. C'est dans ce cadre que les travaux sur l'opérateur de gyromoyenne ont été menés (chapitres 2 à 4). Ce modèle reste néanmoins complexe à mettre en œuvre et ne se prête que difficilement à l'expérimentation de nouveaux schémas numériques pour l'implémentation des différents opérateurs du modèle. C'est pourquoi il existe de nombreux modèles de dimensionnalité inférieure, propres à évaluer rapidement de nouveaux schémas en vue de leur incorporation dans un code gyrocinétique. Nous nous intéresserons particulièrement au modèle 4D *drift-kinetic*.

1.1.5 Modèle 4D *drift-kinetic*

Le modèle 4D *drift-kinetic* en géométrie slab est un modèle dérivé du modèle gyrocinétique. La géométrie slab [40, 64] considère le cas d'un cylindre plutôt que d'un tore afin de pouvoir s'affranchir des termes complexes liés à la courbure dans les équations.

Le champ magnétique est également supposé uniforme et le système est étudié à travers l'évolution des centres-guides. Le moment magnétique μ est considéré comme étant unique pour l'ensemble de la simulation. Le cas particulier $\mu = 0$ simplifie encore le problème puisque les centres-guides sont confondus avec la trajectoire des particules et l'opérateur de gyromoyenne est réduit à l'identité (voir section 2.1.1). Les équations s'en retrouvent grandement simplifiées et permettent de tester de nouveaux schémas numériques ou techniques d'optimisation rapidement sans avoir à mettre en œuvre l'opérateur de gyromoyenne.

Ce modèle est utilisé dans les développements présentés aux chapitres 5 à 7 dans la formulation présentée dans l'article [35]. Les équations sont alors données par (1.4) (où $\mathbf{v}_{\mathrm{cg}}$ désigne la vitesse des centres-guides) pour l'advection de Vlasov et par (1.5) pour la quasineutralitée de Poisson (voir chapitre 6) et seront détaillées dans leur section respective.

$$\frac{\partial \bar{f}}{\partial t} + \mathbf{v}_{\mathrm{cg}} \cdot \nabla_\perp \bar{f} + v_\parallel \frac{\partial \bar{f}}{\partial \varphi} + \dot{v_\parallel} \frac{\partial \bar{f}}{\partial v_\parallel} = 0 \tag{1.4}$$

La représentation spatiale du système utilise des coordonnées toroïdales (r, θ, φ) (voir 1.3) et le gradient perpendiculaire est défini par $\nabla_\perp = (\frac{\partial \cdot}{\partial r}, \frac{1}{r}\frac{\partial \cdot}{\partial \theta})$.

$$-\operatorname{div}_\perp \left[\frac{n_0}{B_0 \Omega_{\mathrm{c}}} \nabla_\perp \Phi \right] + \frac{n_0}{T_{\mathrm{e}}} \left(\Phi - \langle \Phi \rangle_{\theta,\varphi} \right) = n_{\mathrm{i}} - n_0 \tag{1.5}$$

n_{i} représente la densité ionique et n_0 et la densité ionique à l'équilibre, T_{e} la température électronique et B_0 le champ magnétique à l'axe magnétique. La divergence perpendiculaire est définie pour un champ vectoriel $\mathbf{A}$ par $\operatorname{div}_\perp(\mathbf{A}) = \frac{1}{r}\left(\frac{\partial (r A_r)}{\partial r} + \frac{\partial A_\theta}{\partial \theta}\right)$.

Il existe ainsi de nombreux modèles permettant de simuler l'évolution des turbulences au sein d'un plasma avec divers degrés de précision pour différents cas physiques. Néanmoins, tous nécessitent un maillage raffiné et donc de nombreux points afin d'obtenir des résultats quantitatifs satisfaisants et exploitables. C'est pourquoi ces simulations s'exécutent sur des calculateurs parallèles et la parallélisation des codes de simulation associés est une étape centrale pour obtenir des résultats avec des temps de simulation raisonnables.

1.2 Architectures HPC et problématiques

Le calcul haute performance (HPC, *High Performance Computing*) est un domaine de l'informatique centré sur la conception, le développement et l'exploitation de codes nécessitant une puissance de calcul et une capacité mémoire supérieures à celle d'un simple ordinateur de bureau. Il est utilisé dans des domaines variés tels que la prévision météorologique (MétéoFrance), l'exploration des sols (Total) ou les simulations aérodynamiques (Airbus), mais aussi en chimie moléculaire, atomique et quantique (CNRS), pour l'interprétation de données astronomiques (INSU/SKA) ou encore pour la fusion nucléaire (CEA). Bien qu'ayant tous des nécessités spécifiques, chacun de ces domaines a en commun un besoin toujours croissant en puissance de calcul. Les applications utilisant le HPC s'appuient sur des supercalculateurs, c'est-à-dire des machines comportant de nombreux processeurs et une grande quantité de mémoire et de stockage. À l'heure actuelle, les calculateurs les plus performants offrent une puissance de calcul de l'ordre de la centaine de pétaflops (10^{15} opérations flottantes par seconde) en assemblant plusieurs centaines de milliers de cœurs et en proposant des centaines de téraoctets de mémoire vive. La machine actuellement en

tête du top 500 est le calculateur états-unien Summit, mis en service en juin 2018 au Oak Ridge National Laboratory, composé de 2.3 millions de cœurs pour une puissance de 122 pétaflops. Il devance la machine chinoise Sunway TaihuLight avec 10.6 millions de cœurs pour 93 pétaflops. L'architecture de ces calculateurs est plus complexe que celle de simples ordinateurs bien qu'utilisant un design relativement proche au niveau des processeurs.

1.2.1 Architecture globale et fonctionnement

Un supercalculateur se comporte comme des milliers d'ordinateurs reliés entre eux par des liens rapides. Comme le montre la Figure 1.5, il est composé de baies (racks) contenant des nœuds interconnectés par un réseau rapide. Chaque nœud contient un certain nombre de processeurs associés à de la mémoire vive. Outre le nombre plus important de cœurs par processeur et une capacité mémoire plus élevée, la différence principale des supercalculateurs avec les ordinateurs conventionnels est la présence d'unités de calculs distribuées (les nœuds) qui ne partagent pas la même mémoire et doivent utiliser un réseau dédié afin de communiquer des informations. On peut donc distinguer deux niveaux de parallélisme. Un premier niveau entre nœuds de calcul, où les données sont distribuées et où des communications sont nécessaires pour échanger des données et maintenir la cohérence des calculs, et un deuxième niveau à l'échelle d'un nœud qui est une machine à mémoire partagée à part entière et dont les calculs peuvent être parallélisés entre les cœurs des différents processeurs sans donner lieu à des communications.

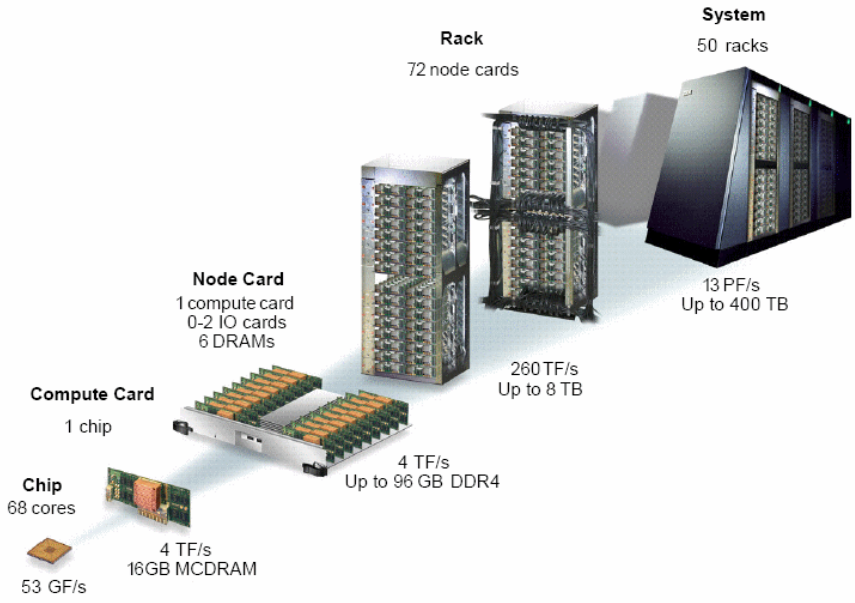


FIGURE 1.5 – Architecture de la machine Intel Marconi A2 utilisant le réseau OmniPath. [5]

La Figure 1.6 présente un extrait (fourni par l'outil hwloc) plus précis de l'architecture d'un calculateur parallèle, ici le calculateur Poincaré hébergé à l'IDRIS sur le plateau de Saclay. Cette petite machine parallèle est utilisée à des fins de développement et de benchmarking. Ce cluster est constitué de plusieurs nœuds de calcul (en jaune) interconnecté à

5. https://wiki.u-gov.it/confluence/display/SCAIUS/UG3.1%3A+MARCONI+UserGuide

l'aide d'un réseau de communication (ici Infiniband). Chaque nœud est divisé en plusieurs groupes NUMA (en vert) ; l'architecture NUMA est détaillée dans le prochain paragraphe. Chaque groupe NUMA héberge plusieurs processeurs (en bleu) et dispose d'une mémoire vive de grande capacité (en rose) accessible à l'ensemble des processeurs du nœud. Les processeurs sont eux-mêmes composés de plusieurs cœurs (en gris) et de plusieurs niveaux de cache (L1, L2 et L3 en blanc) similairement aux processeurs grand public. Finalement, les cœurs sont constitués de plusieurs unités de calculs (entier, flottant...) non représentées sur la Figure.

FIGURE 1.6 – Représentation d'un nœud de calcul sur le calculateur Poincaré donnée par l'outil hwloc (commande `lstopo`).

Les effets NUMA (*Non-Uniform Memory Access*) représentent le fait qu'au sein d'un nœud, il peut être plus coûteux d'accéder à un banc mémoire plutôt qu'à un autre. Cela est dû à la multiplication des processeurs sur les nœuds et donc à l'augmentation de la distance physique entre la mémoire et les cœurs. Ainsi, Figure 1.6, le processeur du groupe 0 mettra par exemple 1.2 fois plus de temps à accéder à la mémoire du groupe 1 qu'à la mémoire du groupe dont il fait partie. Il est essentiel de prendre en compte ces effets liés au fonctionnement des différentes hiérarchies mémoires (cache, mémoire vive et mémoire distante) lors de l'utilisation des données afin de minimiser les temps d'accès (notions de localité abordée dans un prochain paragraphe).

Temps d'accès aux données

Si l'on se réfère à la documentation des processeurs Intel de la famille Xeon 5500 [6] (utilisés dans Poincaré), la table 1.1 présente l'ordre des temps d'accès aux caches des processeurs de ce type, une estimation pour un accès NUMA dont le poids est 1.4, le temps de latence (initialisation de la connexion) sur un réseau Intel, une temps de lecture de donnée sur un SSD et sur un disque dur. Les temps d'accès moyens à la mémoire sont donnés en nombre de cycles d'horloge.

	Nombre de cycles par accès
L1	1
L2	10
L3	40
Mémoire vive (RAM)	100
Mémoire distante (NUMA 1.4)	140
Latence réseau	1 000
Lecture depuis SSD	150 000
Lecture depuis HDD	10 000 000

TABLE 1.1 – Temps d'accès moyens pour les processeurs Xeon 5500 en nombre de cycles CPU pour la lecture/écriture d'un élément.

Les effets de cache sont dus à un comportement similaire. Lorsqu'une donnée est requise par une unité de calcul, une recherche hiérarchique est effectuée dans les caches (L1, L2 puis L3). Si elle n'est pas trouvée en cache, elle est recherchée dans les mémoires vives (processeur, groupe NUMA puis nœud). Une fois trouvé, le bloc mémoire sur lequel est située la donnée est rapatrié en cache. Le temps d'accès à chacun des caches va croissant (proportionnellement à leur éloignement). De même, la bande passante (le débit des liens) suit la même courbe que le temps de latence et augmente à chaque niveau de cache, puis pour la RAM et le réseau. D'autre part, avec l'augmentation du nombre de cœurs, le lien entre ceux-ci et la mémoire vive partagée subit une concurrence toujours plus forte. C'est pourquoi il est très intéressant d'utiliser au maximum des données localisées dans les mêmes blocs mémoires. De par la grande différence de temps d'accès et de bande passante entre les caches et les mémoires vives, il est également essentiel d'arriver à utiliser au maximum les données actuellement en cache. Ces architectures spécifiques présentent ainsi de nombreux défis afin de pouvoir utiliser au mieux l'ensemble des ressources mises à disposition.

Problématiques du calcul parallèle

Si l'on se focalise sur la catégorie d'application HPC qui nous intéresse — les simulations numériques —, un des problèmes se situe dans la gestion de grandes quantités de données. En effet, les simulations se veulent de plus en plus précises et utilisent de ce fait des jeux de données de plus en plus conséquents. L'ensemble des données ne rentrent plus dans la mémoire d'un unique nœud de calcul ; elles doivent être distribuées entre plusieurs nœuds. Les données doivent ainsi être réparties de manière adaptée à chaque application afin de limiter les communications, bien plus longues qu'un simple accès mémoire (voir table 1.1).

6. `https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-5500-specification-update.pdf`

Lorsque les communications sont inévitables, elles doivent être effectuées de manière à minimiser leurs coûts. De plus, au sein d'un même nœud, tous les accès en mémoire ne se valent pas, du fait des effets de cache qui jouent sur la localité temporelle et spatiale, et des effets NUMA, qui jouent sur la localité spatiale [6]. La multiplicité des unités de calculs et leur répartition sur plusieurs nœuds communiquant impliquent une parallélisation à plusieurs échelles grâce à des paradigmes de programmation hybride [61] comme le modèle MPI+OpenMP que nous utiliserons, ou le modèle MPI+OpenACC pour les accélérateurs [73].

D'autre part, des effets liés aux schémas numériques peuvent jouer un rôle dans les performances. Par exemple, un traitement appliqué à un sous-ensemble de données peut être plus long et complexe que ce même traitement appliqué à un autre sous-ensemble. Cela entraîne un déséquilibre quant à la quantité de travail (calculs et accès mémoire). Au niveau d'un processus, des cœurs peuvent ainsi se retrouver en attente d'autres cœurs et rester inutilisés. Pour y remédier, on pratique un équilibrage de charges adapté ou dynamique [17], c'est-à-dire qu'on étudie une stratégie pour répartir le travail équitablement entre les unités de calculs. Ce même déséquilibre de charge entre processus peut entraîner des effets de synchronisation forts. Dans le pire des cas, tous les processus se retrouvent en attente du processus le plus lent. Plusieurs solutions sont possibles en fonction du type de synchronisation, recouvrement calcul – communication [60], traitement prioritaire d'une partie des données non dépendantes des synchronisations ou redécoupage de la distribution de données.

Ces problèmes sont exacerbés à chaque amélioration des calculateurs et plus particulièrement pour la nouvelle génération de calculateurs pré-*exascale*. Les algorithmes et les outils de parallélisation sont dès lors à réévaluer à chaque génération. Il est néanmoins désormais nécessaire de coupler finement ces développements et techniques d'optimisation avec les applications. Il est en effet courant que des applications n'utilisent pas plus de quelques pourcents de la capacité d'un calculateur. C'est pourquoi les recherches doivent être et sont actuellement menées de façon interdisciplinaire entre le calcul haute performance et les domaines applicatifs.

Dans un code HPC, il est également important de ne pas négliger le génie logiciel. La maintenabilité, l'adaptabilité et la portabilité sont des caractères essentiels pour une application haute performance. Un juste équilibre entre optimisation et abstraction doit être trouvé afin d'offrir de bonnes performances tout en permettant une compréhension rapide du code et de sa structure sans nécessiter plusieurs mois d'expertises. De plus, il est également important que les physiciens non experts en HPC puissent continuer à modifier directement leurs schémas numériques sans déléguer l'ensemble de la tâche aux informaticiens. D'autre part, un excès d'optimisations entraîne généralement une faible évolutivité du code et se fait en général pour des architectures spécifiques. La nécessité d'introduire un changement dans les algorithmes ou de changer de calculateur peut alors devenir problématique. C'est pourquoi, le développement de tels codes est un problème interdisciplinaire multi-contraint.

Architectures exascales

Les calculateurs de future génération, dits exascales, doivent atteindre une puissance de calcul de l'ordre de l'exaflops (10^{18} opération flottante par seconde). Plusieurs projets sont actuellement à l'étude ou en cours de réalisation. On notera le calculateur chinois Tianhe-3 qui devrait être le premier à voir le jour à l'horizon 2020, le projet américain Exascale

Computing Project qui vise 2022 pour sa machine ou encore l'initiative européenne EuroHPC et le futur calculateur japonais Post-K. On sait que ces machines disposeront de beaucoup plus de nœuds de calcul que leurs prédécesseurs et que chaque nœud disposera également de plus de cœurs (plusieurs dizaines de millions de cœurs au total dans un calculateur). Le développement des codes HPC sera donc fortement axé sur la gestion de la mémoire qui devient plus complexe avec le nombre de nœuds (plus de communications) et le nombre de cœurs (renforcement des effets NUMA et de la concurrence d'accès). Les temps de communications deviennent également critiques puisque la multiplication des nœuds entraîne une multiplication des baies et donc un étalement des clusters. De plus, le débit et la latence des réseaux ne s'améliorent pas aussi vite que la puissance de calcul des processeurs créant nécessairement un goulet d'étranglement.

Ces nouvelles machines exascales feront face à un problème de consommation énergétique important. De 0.7MW pour 137 téraflops en 2005 (BlueGene/L), nous sommes passés à 9MW pour 120 pétaflops (Summit) et les premières machines exascales sont attendues avec une consommation de l'ordre de 20 à 30MW pour un exaflop. Bien que la consommation par opération flottante ait diminué d'un facteur 50 ces dix dernières années, la quantité d'énergie consommée devient problématique. D'autre part, à ces échelles, la probabilité qu'un calcul ou une donnée soit affecté par une interférence extérieure (le rayonnement cosmique par exemple) ou qu'un matériel subisse une panne durant la simulation n'est plus négligeable. De nombreuses études existent pour pallier ces problèmes. On citera quelques solutions, dont la tolérance aux pannes, la redondance dans le stockage des données, la redondance des calculs [14].

De nombreuses inconnues persistent également, notamment quant au *hardware* qui sera utilisé. La présence d'accélérateurs et de GPU dans les nœuds de calcul n'est pas garantie, les types de mémoires utilisées ne sont pas encore certains de même que la nature des réseaux d'interconnexion entre nœuds. Par exemple, le calculateur Summit couple à chaque paire de processeurs six GPUs grâce à un lien NVLink2. La mémoire vive y est composée d'une part de DDR4 et d'une part de NVRAM. Au contraire, le calculateur TaihuLight n'utilise que des processeurs spécifiques contenant 256 cœurs (processeurs *manycore*) et de la mémoire vive DDR3 utilisant un contrôleur dédié.

Nous nous focaliserons dans les travaux présentés ci-après sur la distribution des données et la scalabilité mémoire, sur les effets de synchronisation dans les communications et sur l'équilibrage de charge. Ces problématiques, qui feront l'objet d'une présentation spécifique dans les sections suivantes, sont critiques pour avancer dans la direciton de l'*exascale*.

1.2.2 Localité, distribution de données et scalabilité mémoire

La localité spatiale caractérise le fait, pour un schéma numérique donné, d'utiliser le plus possible des données stockées de manière proche en mémoire. De par la façon dont les données sont chargées en cache, l'utilisation d'une première donnée chargera ainsi automatiquement un certain nombre d'autres données situées proche en mémoire. Ainsi, il est intéressant de stocker de manière proche en mémoire des données qui seront utilisées consécutivement. Cela a par exemple un impact sur l'ordre du stockage des dimensions dans les simulations multidimensionnelles [19]. À cause des effets NUMA, il est également bien plus efficace de positionner les données proches du processeur qui les traitera. La localité temporelle caractérise le fait de réutiliser rapidement les mêmes données pour différents calculs. Cela est dû aux effets de cache, plus rapides d'accès que la mémoire vive. Par

exemple sur un problème 1D, si la mise à jour d'une cellule se fait en combinant sa valeur et celle de ses deux voisines, mettre à jour la cellule voisine réutilisera la valeur de la cellule précédente qui sera déjà en cache.

Les distributions de données dans les simulations numériques doivent prendre en compte ces paramètres de localité. Les lois d'évolution physique font souvent intervenir, soit des combinaisons de grandeurs physiquement proches, soit des grandeurs dispersées sur l'ensemble du domaine. Lorsque les grandeurs sont réparties sur l'ensemble du domaine, il faut effectuer un maximum de précalculs localement avant de communiquer avec les processus distants. Lorsque les grandeurs sont physiquement proches, il convient de les stocker en mémoire dans un ordre qui correspond à la dimension la plus sollicitée. De plus, les algorithmes peuvent être adaptés afin de ne pas traiter le jeu de données dimension par dimension, mais blocs par blocs afin de maximiser la localité [75]. Néanmoins, différents opérateurs peuvent ne pas avoir les mêmes besoins en matière de structures de données et de localité ; plusieurs choix s'offrent alors. Soit on fixe une distribution de donnée en fonction de l'opérateur le plus exigeant [16], soit on se laisse la possibilité d'utiliser plusieurs distributions de données en redistribuant le jeu de données. Cela peut engendrer d'importants coûts de communication, et néanmoins être la solution la plus économe par rapport au ratio temps d'exécution – temps de développement.

L'empreinte mémoire désigne le pic de quantité de mémoire utilisée lors d'une simulation. Bien que les données à traiter puissent être réparties sur de nombreux nœuds afin d'absorber leur taille, d'autres données nécessaires à la simulation peuvent prendre une large place en mémoire et empêcher une simulation de s'exécuter. Les structures liées aux communications représentent généralement une fraction non négligeable du sous-domaine local, et leur nombre est souvent proportionnel au nombre de processus distants. Ainsi plus il y a de processus dans la simulation, plus la proportion des structures de communication dans l'empreinte mémoire est importante. On parle de scalabilité mémoire, l'objectif étant que l'empreinte mémoire de la simulation dépende le moins possible du nombre de nœuds allouées. Lors du développement d'un code HPC, il est ainsi essentiel de minimiser l'empreinte mémoire en n'hésitant pas à remodeler les algorithmes tels qu'on peut le faire pour optimiser la scalabilité des calculs.

1.2.3 Schéma de communication et asynchronisme

Les schémas de communication sont primordiaux dans la plupart des simulations numériques. Les communications sont effectuées par une interface dédiée sur un réseau spécifique. Plusieurs technologies sont utilisées pour la couche matérielle : InfiniBand [56] et Intel Omni-Path [7] sont les plus répandues. Pour la couche logicielle, la bibliothèque la plus utilisée dans le HPC reste MPI (*Message Passing Interface*) [29] bien que différentes alternatives existent. Dans une communication, deux paramètres sont importants. Le temps de latence – le temps nécessaire à l'établissement de la communication – et le débit du lien – la vitesse à laquelle les données sont effectivement échangée. Ainsi pour une communication donnée, son temps d'exécution est modélisé par

$$T_c = \lambda + \beta N \tag{1.6}$$

où N est la quantité d'information transmise (en octets) et où λ et β dépendent du réseau et de la localisation des processus émetteurs et récepteurs ainsi que de la pile logicielle. λ donne le temps d'initialisation de la communication et β le débit du lien. Ainsi il vaut

mieux envoyer beaucoup d'informations en une seule communication que de faire plusieurs envois moins conséquents. Néanmoins, trop retarder l'envoi de certaines données afin d'en faire un seul bloc pourrait également retarder l'exécution du code. Un critère pragmatique est d'atteindre N tel que $\lambda << \beta N$. Bien évidemment, cette modélisation des temps de communications ne tient pas compte d'une éventuelle contention du réseau qu'il faut évaluer par d'autres moyens.

De ces communications peuvent émerger des problèmes de synchronisation entre processus. En effet, les algorithmes peuvent être amenés à effectuer des communications pendant des phases de calcul, mettant potentiellement les processeurs à l'arrêt en attente de la terminaison des communications. Une bonne solution consiste alors à effectuer les communications dès que les données sont disponibles et simultanément à d'autres calculs. L'exécution parallèle de calculs et de communications est permise par leur découplage physique. Les communications, bien qu'initiées à partir des processeurs de calcul, sont effectuées par une carte réseau. Côté logiciel, les bibliothèques de communication se sont dotées de primitives dites non bloquantes [31]. Contrairement aux primitives classiques qui attendent que la communication soit bien effectuée avant de rendre la main, ces dernières retournent directement et la terminaison effective de la communication doit être testée avant de pouvoir utiliser les données transmises. Si de plus, les communications se déroulent effectivement lors des calculs et non uniquement lors de l'utilisation de la primitive de test, celles-ci sont dites asynchrones et le code effectue un schéma appelé recouvrement calcul – communication [66]. Généralement, dans le cas de calculs utilisant des *stencils* locaux, il faut d'abord lancer les communications permettant d'obtenir les données nécessaires au traitement des points au bord du domaine local, puis traiter en priorité les points intérieurs au domaine local. Cela permet aux communications d'avancer, voire de se terminer, avant que le traitement des points au bord du domaine local ne commence.

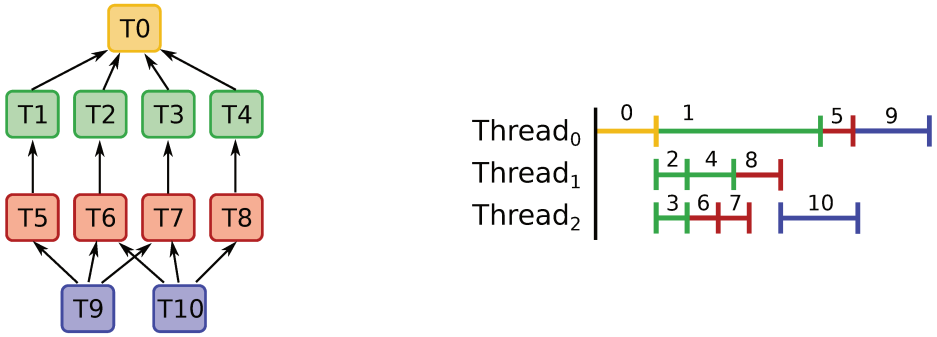
Certains environnements permettent de gérer simultanément la distribution des données et leur partage entre processus distants de manière transparente. Les langages PGAS, par exemple Global Array [55] ou Spartan [42], sont des environnements qui virtualisent une mémoire partagée sur une mémoire distribuée. Les lectures et écritures se font grâce à des primitives qui effectuent des communications si nécessaire. Cela n'enlève pas pour autant la nécessité de concevoir une distribution de données spécifique à chaque problème.

1.2.4 Programmation par tâches

Certaines simulations utilisent des algorithmes constitués de multiples parties distinctes successives ou effectuent un traitement très hétérogène entre différentes données. Il peut alors s'avérer complexe de répartir équitablement le travail entre les cœurs d'un nœud afin d'éviter les phases d'inactivité. Dans le cas d'un code régulier dont les instructions ne varient que peu spatialement et temporellement, une étude approfondie des coûts peut permettre de créer ponctuellement un bon équilibrage ad hoc. Néanmoins, en cas de dépendance spatiale des opérateurs, ou si l'évolution de la simulation modifie les traitements à effectuer, cela n'est plus possible et nécessite alors un système dynamique d'équilibrage de charge. Il est possible de créer soi-même un équilibrage de charge dynamique en subdivisant les calculs en petits blocs dont on distribue l'exécution à la volée aux ressources de calculs disponibles. Le modèle de programmation par tâches permet de faciliter ce type d'approche.

L'un des concepts de la programmation par tâches est de diviser un algorithme en un ensemble de tâches distinctes à effectuer et de laisser exécuter ces tâches par un ordonnan-

ceur. Une tâche est un ensemble d'instructions formant une entité cohérente, par exemple calculer l'ensemble des forces appliquées à un sous-ensemble de particules voisines peut représenter une tâche. La taille des tâches (quantité d'instructions de calculs ou volume de données à communiquer) est un des paramètres d'optimisation du modèle. Les dépendances entre tâches représentent l'enchaînement des calculs. Par exemple, pour mettre à jour les vitesses d'une particule nous avons besoin des forces qui lui sont appliquées ; il y a donc dépendance de la tâche de calcul de la vitesse par rapport à la tâche de calcul des forces. L'ordonnanceur, aussi appelé *runtime*, est l'outil qui va organiser l'exécution des tâches en fonction des dépendances et des ressources allouées. La complexité du modèle de programmation par tâches est donc portée sur la définition des tâches et du graphe de dépendances et sur l'ordonnacement des tâches à l'exécution. Les performances reposent fortement sur la capacité de l'algorithme de l'ordonnanceur à équilibrer les charges et à limiter les coûts liés à la gestion de celles-ci.



(a) Diagramme de tâches.

(b) Exécution du diagramme de tâche avec trois *threads* et une politique d'allocation gloutonne.

FIGURE 1.7 – Exemple de déroulement d'une simulation avec un modèle de tâches.

La Figure 1.7a donne un exemple de graphe de tâches. Les dépendances sont représentées par des flèches ; la tâche 9 dépend par exemple des tâches 5, 6 et 7. La Figure 1.7b présente une exécution possible du diagramme de tâches avec trois fils d'exécution, aussi appelés *threads*. À l'initialisation, la seule tâche sans dépendance est la tâche T0, elle est donc attribuée au premier *thread* inactif. À sa terminaison, les tâches vertes sont débloquées et les tâches T1, T2 et T3 sont allouées. Les tâches T2 et T3, plus courtes que la tâche T1, terminent et débloquent donc les tâches T6 et T7. L'ordonnanceur doit alors choisir deux tâches parmi trois pour occuper les *threads* inactifs. Différentes politiques d'allocation sont possibles en fonction de la priorité des tâches, la localisation des données utilisées en mémoire ou d'autres paramètres. Pour l'exemple nous utilisons une politique simple qui suit la numérotation des tâches. Ainsi T4 et T6 sont attribuées aux *threads* 1 et 2. T6 termine et le *thread* 2 effectue ensuite la tâche T7. Pendant ce temps T4 termine et libère T8 qui est attribué au *thread* 1. T7 termine et *thread* 2 se retrouve inactif faute de tâche sans dépendance. T8 termine et débloque T10 qui est effectuée par *thread* 2 et *thread* 1 reste en attente. T1 termine enfin et *thread* 0 effectue T5 puis T9.

Par définition, l'équilibrage de charge est un des atouts du modèle de programmation par tâches. La description par tâche facilite également l'implémentation d'un schéma de recouvrement calcul – communication. En effet, les communications peuvent être incluses dans des tâches en utilisant par exemple des dépendances sur de faux conteneurs de données

représentant les données distantes ; en utilisant des communications non bloquantes, cela encapsule l'implémentation du recouvrement. De plus, la description par tâches permet d'exhiber le parallélisme de façon nette, ce qui est un atout pour les architectures *many-core*. Les architectures hétérogènes peuvent également tirer parti de cette description. En ayant un noyau optimisé pour chaque matériel (processeur, GPU, accélérateur...), l'ordonnanceur peut aisément répartir les calculs entre eux pour en tirer le plus de puissance possible. Inversement, il peut aussi se servir du découpage en tâches pour associer celles-ci aux ressources de calcul de sorte à minimiser la consommation énergétique de l'application.

Le modèle de programmation par tâches autorise donc une description à grain variable du problème ce qui permet différentes optimisations. Différentes technologies existent pour mettre en place ce modèle avec différents degrés d'intrusion dans le code. Le modèle de tâches OpenMP [18, 25] est le moins intrusif car basé sur l'utilisation de pragmas et ayant une formulation similaire à une parallélisation par boucle. À l'opposé, StarPU [4] propose un modèle où tout est géré par la bibliothèque (mémoire, tâches, communications) ; le code est donc indissociable de celle-ci. Comme mentionné en début de section, le gain qui peut être attendu de ce modèle de programmation par tâches dépend fortement de l'application et de la présence de temps d'inactivité dans l'exécution. Comme décrit dans [46], le code Gysela pourrait profiter de cette approche afin d'utiliser au mieux l'ensemble des ressources computationnelles. C'est pourquoi le prototype 4D présenté dans les chapitres 5 à 7 utilise le modèle de programmation par tâches. Il permettra d'estimer la viabilité du modèle pour un code gyrocinétique. L'implémentation du modèle dans ce prototype est présenté dans le chapitre 5 et son apport est évalué au chapitre 7.

1.3 Le code Gysela

Le code Gysela [37] est un code 5D gyrocinétique semi–Lagrangien de simulation des turbulences issues des gradients de température (ionique principalement) dans un plasma de tokamak. Il permet également de simuler d'autres phénomènes comme les collisions intra- et inter- espèces ou encore la physique des électrons cinétiques. Il a servi de base aux travaux de cette thèse.

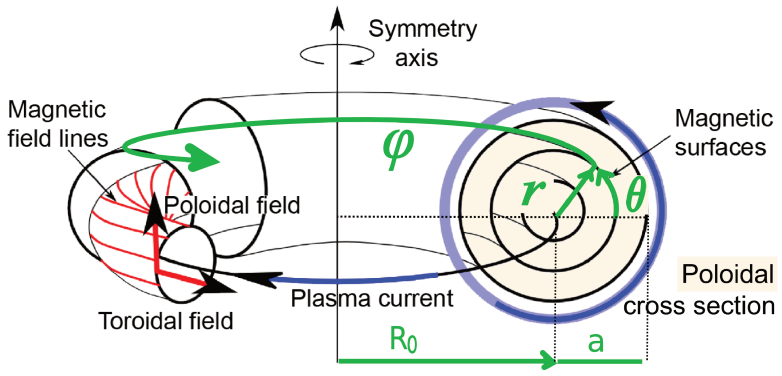
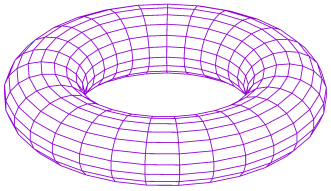


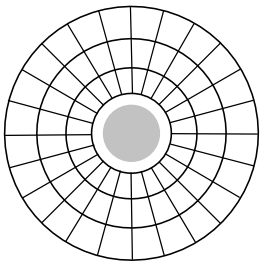
Figure 1.8 – Représentation géométrique du plasma dans Gysela.

Gysela utilise actuellement une géométrie de tokamak à coupe circulaire. Le plasma évolue dans un tore représenté comme indiqué en Figure 1.8. L'angle dans le tore, appelé angle toroïdal, est noté φ. À angle toroïdal fixe, l'intersection d'un plan avec le tore forme

un disque, appelé plan poloïdal. Il est décrit en coordonnée polaire par les variables r et θ. La vitesse parallèle est notée $v_\parallel$ et l'invariant adiabatique μ (voir 1.1.4). Les coordonnées en un point de l'espace seront notées $\mathbf{x} = r\,\mathbf{e}_r + \theta\,\mathbf{e}_\theta + \varphi\,\mathbf{e}_\varphi$ dans le référentiel polaire et $\mathbf{x} = x\,\mathbf{e}_x + y\,\mathbf{e}_y + \varphi\,\mathbf{e}_\varphi$ dans le référentiel cartésien avec $x = r\cos(\theta)$ et $y = r\sin(\theta)$. S'il n'y a pas d'ambiguïté sur le référentiel utilisé, la notation sera simplifiée en $(r, \theta, \varphi, v_\parallel, \mu)$ et $(x, y, \varphi, v_\parallel, \mu)$. La Figure 1.9a donne un aperçu du maillage 3D du tore. La dimension r n'est pas simulée jusqu'à 0 du fait d'instabilités numériques résultant de l'expression de l'opérateur de Poisson (termes en $\frac{1}{r}$). Ainsi, le plan poloïdal n'est pas un disque, mais un anneau de rayons r_{min} à r_{max}. La Figure 1.9b le montre sur une coupe poloïdal à φ constant du tore. Le maillage est uniforme dans toutes les dimensions et nous noterons par la suite N_r (respectivement N_θ, N_φ, $N_{v_\parallel}$, N_μ) le nombre de points dans la direction r, Δr (respectivement $\Delta\theta$, $\Delta\varphi$, $\Delta v_\parallel$, $\Delta\mu$) le pas, L_r (respectivement L_θ, L_φ, $L_{v_\parallel}$, L_μ) la taille du domaine avec $dX = \frac{L_X}{N_X}$ quelque soit la dimension X.



(a) Maillage d'une surface magnétique (r = constante).



(b) Maillage du plan poloïdal.

FIGURE 1.9 – Représentation du maillage spatial du tokamak dans GYSELA.

Les détails des hypothèses physiques autres que celles nécessaires à l'application du modèle gyrocinétique ne seront pas détaillées ici. Le lecteur pourra trouver plus d'informations dans [37].

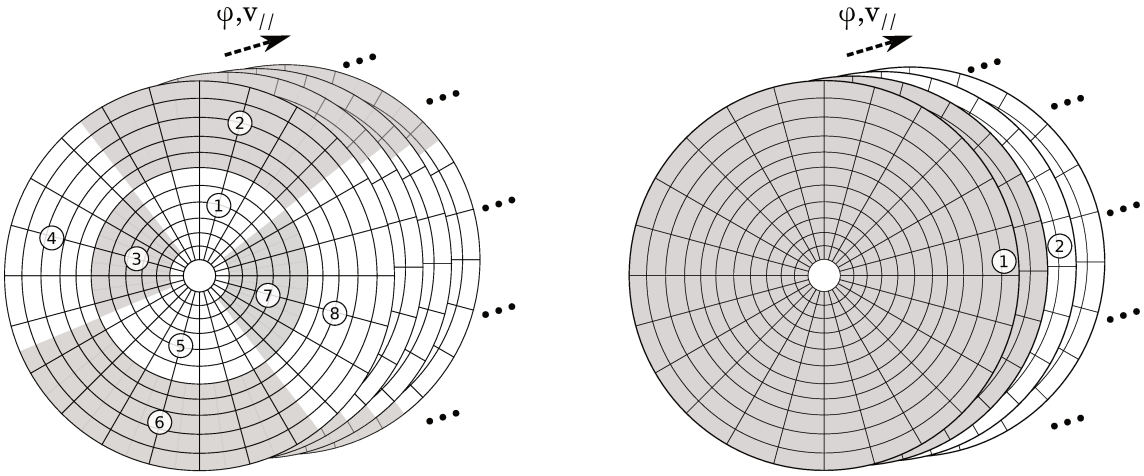
1.3.1 Cadre d'implémentation

GYSELA est un code hybride MPI+OpenMP massivement parallélisé, écrit en Fortran et ayant fait l'objet de nombreuses recherches et optimisations [46, 47, 63]. Nous présenterons d'abord la parallélisation MPI et quelques notations avant d'introduire la parallélisation OpenMP et quelques optimisations fondamentales pour les opérateurs les plus importants.

La dimension μ permet de séparer le problème en autant de sous-problèmes que de valeurs de μ puisqu'il est invariant (il n'y a pas de transport dans cette dimension). Des communicateurs MPI indicés par μ sont donc créés et chaque processus MPI ne prend en charge q'un seul et unique μ. Au sein d'un communicateur, plusieurs distributions de données sont utilisées pour répartir la fonction de distribution entre les processus au cours de la simulation. Les deux principales, que nous nommerons $\mathbf{D}_1$ et $\mathbf{D}_2$ sont représentées en Figure 1.10. En distribution $\mathbf{D}_1$ (Figure 1.10a), les dimensions r et θ sont distribuées entre les processus qui dispose donc d'un sous-domaine de chaque plan poloïdal pour tout φ et $v_\parallel$. En distribution $\mathbf{D}_2$ (Figure 1.10b), les dimensions φ et $v_\parallel$ sont distribuées entre les processus qui dispose donc d'un ensemble de plan poloïdaux complets. Ces distributions peuvent être exprimées de façon analytique. Commençons par noter $N_{Lr} = \frac{N_r}{\mathtt{Np}_r}$ et $N_{L\theta} = \frac{N_\theta}{\mathtt{Np}_\theta}$ le nombre de points de chaque sous-domaine dans les dimensions r et θ avec

$\texttt{Np}_r$ (respectivement $\texttt{Np}_\theta$) le nombre de processus dans la dimension r (respectivement θ). Chaque processus dispose donc d'un bloc de données contenant $N_{Lr}N_{L\theta}N_\varphi N_{v_\parallel}$ points. Pour un processus $P_{i,j}$ indicé en fonction de sa localisation dans le plan $r \times \theta$ au sein d'un communicateur en μ, les distributions de données présentée précédemment s'expriment :

$$\begin{aligned} \mathbf{D}_1(P_{i,j}) &= (iN_{Lr} : (i+1)N_{Lr},\, jN_{L\theta} : (j+1)N_{L\theta},\, *,\, *,\, \mu) \\ \mathbf{D}_2(P_{i,j}) &= (*,\, *,\, iN_{L\varphi} : (i+1)N_{L\varphi},\, jN_{Lv_\parallel} : (j+1)N_{Lv_\parallel},\, \mu). \end{aligned} \tag{1.7}$$



(a) Processus en distribution $\mathbf{D}_1$. Les dimensions r et θ sont distribuées.

(b) Processus en distribution $\mathbf{D}_2$. Les dimensions φ et $v_\parallel$ sont distribuées.

FIGURE 1.10 – Schéma des distributions $\mathbf{D}_1$ et $\mathbf{D}_2$ sur un sous-ensemble de plan poloïdaux.

Algorithme 1 : Schéma des différents niveaux de parallélisme pour l'advection dans la direction φ en distribution $\mathbf{D}_1$.

Entrée : $\bar{f}_n(r, \theta, \varphi, v_\parallel, \mu)$
Sortie : $\bar{f}_n^*(r, \theta, \varphi, v_\parallel, \mu)$

pour $\mu = 0 : N_\mu - 1$ **faire** en parallèle **MPI**
  pour $r = 0 : N_r - 1$ **faire** en parallèle **MPI**
    pour $\theta = 0 : N_\theta - 1$ **faire** en parallèle **MPI**
      pour $r' = iN_{Lr} : (i+1)N_{Lr} - 1$ **faire** en parallèle **OpenMP**
        pour $\theta' = jN_{L\theta} : (j+1)N_{L\theta}$ **faire** en parallèle **OpenMP**
          pour $v_\parallel = 0 : N_{v_\parallel} - 1$ **faire**
            Calcul des splines cubiques de $\bar{f}_n(r', \theta', \varphi = *, v_\parallel, \mu)$
            pour $\varphi = 0 : N_\varphi - 1$ **faire**
              $d\varphi \leftarrow (v_\parallel +$ d'autres termes$)\Delta t$
              $\bar{f}_n^*(r', \theta', \varphi, v_\parallel, \mu) \leftarrow \texttt{interpolation}(\bar{f}_n(r', \theta', \varphi - d\varphi, v_\parallel, \mu))$

Dans chaque processus la parallélisation multicœur est effectuée avec OpenMP via un parallélisme de boucles dans les directions r, θ, φ ou $v_\parallel$ suivant les configurations. L'algorithme 1 donne un exemple complet de la parallélisation d'un opérateur : l'advection en φ. Cette opérateur est utilisé avec la distribution de données $\mathbf{D}_1$: pour chaque *cluster* μ, les dimensions r et θ sont distribuées et parallélisées en MPI, et sur chaque processus le domaine local en r et θ est parallélisé entre les *threads* OpenMP. Chaque *thread* traite ainsi l'ensemble d'un plan $\varphi \times v_\parallel$.

Algorithme 2 : Schéma du splitting directionnel utilisé par GYSELA pour résoudre l'équation de Vlasov sur un pas de temps.

Entrées : $\bar{f}_n(r,\theta,\varphi,v_\parallel,\mu)$, $\mathbf{E}_{n+\frac{1}{2}}(r,\theta,\varphi)$
Sortie : $\bar{f}_{n+1}(r,\theta,\varphi,v_\parallel,\mu)$

pour $\mu = 0 : N_\mu - 1$ **faire** en parallèle **MPI**
- `/* Distribution` $\mathbf{D}_1$ `*/`
- **pour** $r = 0 : N_r - 1$ **faire** en parallèle **MPI**
  - **pour** $\theta = 0 : N_\theta - 1$ **faire** en parallèle **MPI**
    - Advection $1D$ de $\frac{\Delta t}{2}$ en $v_\parallel$ (boucles OpenMP)
    - Advection $1D$ de $\frac{\Delta t}{2}$ en φ (boucles OpenMP)
- Transposition de la distribution de $\mathbf{D}_1$ vers $\mathbf{D}_2$
- `/* Distribution` $\mathbf{D}_2$ `*/`
- **pour** $\varphi = 0 : N_\varphi - 1$ **faire** en parallèle **MPI**
  - **pour** $v_\parallel = 0 : N_{v_\parallel} - 1$ **faire** en parallèle **MPI**
    - Advection $2D$ de Δt en (r,θ) (boucles OpenMP)
- Transposition de la distribution de $\mathbf{D}_2$ vers $\mathbf{D}_1$
- `/* Distribution` $\mathbf{D}_1$ `*/`
- **pour** $r = 0 : N_r - 1$ **faire** en parallèle **MPI**
  - **pour** $\theta = 0 : N_\theta - 1$ **faire** en parallèle **MPI**
    - Advection $1D$ de $\frac{\Delta t}{2}$ en φ (boucles OpenMP)
    - Advection $1D$ de $\frac{\Delta t}{2}$ en $v_\parallel$ (boucles OpenMP)

L'opérateur d'advection de Vlasov a pour objet de déplacer les particules en faisant évoluer la fonction de distribution. Il utilise une séparation directionnelle appelée *splitting* de Strang (*cf.* section 3.2.1 «Time-splitting» de [35]) afin d'éviter une advection 4D coûteuse en calculs et communications. L'advection est donc effectuée en combinant plusieurs advections 1D et 2D sur des pas de temps plus courts tel que représenté par l'algorithme 2. Néanmoins, la distribution de données $\mathbf{D}_1$ n'est pas adaptée à l'advection 2D dans la direction $r \times \theta$. La solution initiale était basée sur des splines locales et des zones buffer de taille fixe autour des sous-domaines. Cela impliquait que les déplacements en $r \times \theta$ ne dépassent pas un certain nombre de pas de maillage introduisant une condition de type CFL [7] sur le pas de temps. Afin de lever cette restriction, une autre distribution de données, cette fois distribuée dans les dimensions φ et $v_\parallel$, permettant d'avoir accès à tout le plan poloïdal localement a été introduite. Cette solution présente toutefois un coût de communication

7. Courant-Friedrich-Lévy

non négligeable (augmentation de $8-12\%$ du temps d'exécution total) à pas de temps équivalent. Cette augmentation est contre-balancée par la levée de la condition CFL sur le pas de temps, ce qui permet de diminuer le pas de temps d'un facteur au moins dix, réduisant d'autant les temps de simulation.

L'opérateur de gyromoyenne permet de passer de la fonction de distribution des particules à celle des centres-guides. Il était jusqu'à récemment approché dans GYSELA en effectuant une approximation de Padé [67]. Cette approximation, bien que d'un faible coût calculatoire, n'est valide que pour de faibles rayons de Larmor. De plus, elle présente l'inconvénient d'utiliser des transformées de Fourier dans la direction θ combinée à des différences finies dans la direction r. Cela implique d'avoir localement l'ensemble du plan poloïdal (distribution $\mathbf{D}_2$). Hors, certains appels à l'opérateur de gyromoyenne se font depuis une zone du code en distribution $\mathbf{D}_1$. Il est donc nécessaire de transposer la distribution de données le temps de l'opérateur de gyromoyenne, générant de coûteuses communications. C'est pourquoi un autre opérateur pouvant utiliser les distributions $\mathbf{D}_1$ ou $\mathbf{D}_2$ et basé sur une discrétisation de la formule exacte a été étudié, développé et implémenté dans GYSELA [63] (voir chapitre 2).

L'opérateur de Poisson calcule le champ électrique utilisé lors de l'étape d'advection. Il est utilisé sous la forme d'une équation de quasineutralité. Il a également fait l'objet d'une parallélisation poussée dans [47].

1.3.2 Limitations et contraintes du code

La parallélisation du code GYSELA lui a permis de s'exécuter sur des configurations allant jusqu'à plusieurs dizaines de milliers de cœurs avec une efficacité avoisinant les 0.9 (voir Figure 1.11). Le passage à l'échelle supérieur et à la centaine de milliers de cœurs ne se fait actuellement que difficilement du fait de la scalabilité des algorithmes, voire pas du tout à cause de la scalabilité mémoire limitée.

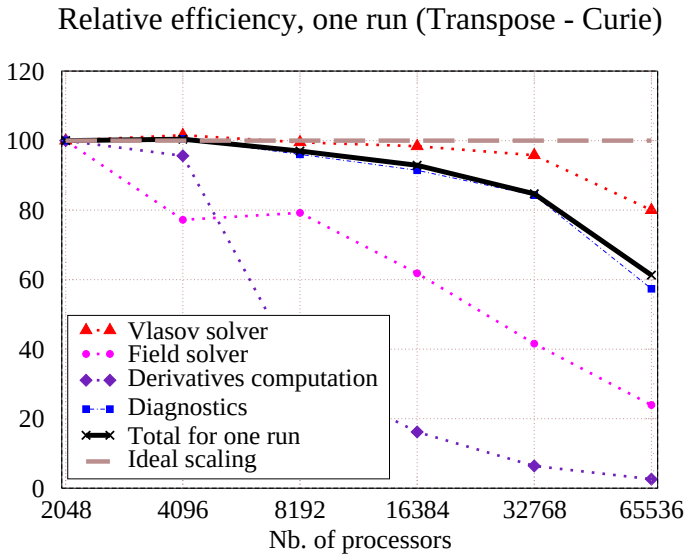


FIGURE 1.11 – *Strong scaling* relatif de l'efficacité du code GYSELA pour un cas de taille $(512 \times 512 \times 128 \times 128 \times 32)$ en fonction du nombre de cœurs utilisé sur la machine Curie de l'IDRIS. Extrait de [45].

Les problèmes de scalabilité mémoire de GYSELA ont déjà fait l'objet d'études [62] et ont été nettement résorbés. Néanmoins, ces problèmes sont structurels et une refonte en profondeur du code serait nécessaire pour lui permettre de s'exécuter sur des configurations encore plus grandes. L'effort doit essentiellement porter sur les structures temporaires 2D et 3D qui utilisent à elles seules plus de 50% de la mémoire sur les grosses configurations (voir table 1.2). Les structures 2D sont essentiellement utilisées pour la parallélisation OpenMP, mais également pour les communications et le traitement d'opérateurs dans le plan poloïdal (gyromoyenne, Poisson). Dans tous les cas, il est complexe de réduire la mémoire utilisée dans ces structures temporaires sans changer radicalement les algorithmes et schémas de communication.

Nombre de cœurs Nombre de processus MPI	$2k$ 128	$4k$ 256	$8k$ 512	$16k$ 1024	$32k$ 2048
structures 4D	207.2 79.2%	104.4 71.5%	53.7 65.6%	27.3 52.2%	14.4 42.0%
structures 3D	42.0 16.1%	31.1 21.3%	18.6 22.7%	15.9 30.4%	11.0 32.1%
structures 2D	7.1 2.7%	7.1 4.9%	7.1 8.7%	7.1 13.6%	7.1 20.8%
structures 1D	5.2 2.0%	3.3 2.3%	2.4 3.0%	2.0 3.8%	1.7 5.1%
Total par processus MPI en Go	261.5	145.9	81.9	52.3	34.3

TABLE 1.2 – Taille des allocations en Go par processus MPI et pourcentage par rapport au pic mémoire de chaque catégorie de structure de données pour un maillage de taille $1024 \times 4096 \times 1024 \times 128 \times 2$. Extrait de [62].

Ces schémas de communication sont également un des points faibles de GYSELA, de même que les synchronisations bloquantes qu'ils génèrent. Certains opérateurs actuels reposent sur l'utilisation des transpositions de la fonction de distribution 4D (changement de distribution entre $\mathbf{D}_1$ et $\mathbf{D}_2$), ce qui est très coûteux. Elles pourraient être évitées en changeant certains algorithmes parallèles. Néanmoins, réduire le nombre de communications se fait souvent au détriment de l'empreinte mémoire en agglomérant les données afin de les envoyer plus tard. Il y a donc un compromis à trouver entre fréquence de communication et empreinte mémoire.

Les écritures et lectures sur disques liées aux diagnostics effectués lors de la simulation représentent une part non négligeable du temps de la simulation. Les *check-points* ont été récemment modifiés afin de pouvoir s'exécuter en parallèle du calcul des pas de temps suivants, et ce au prix du stockage d'une autre copie de la fonction de distribution.

Toutes ces limitations sont un frein au portage du code GYSELA sur les nouvelles machines pré-*exascales*. Leur puissance de calcul sera pourtant nécessaire pour permettre l'introduction d'une nouvelle classe de particules permettant de s'approcher encore de la réalité : les électrons cinétiques. Leur introduction nécessite de modéliser une espèce additionnelle et donc d'avoir une fonction de distribution supplémentaire. Cette nouvelle physique est également plus gourmande en calcul et plus exigeante quant à la discrétisation de maillage. C'est pourquoi cette thèse a engagé des travaux axés vers l'accompagnement

de cette physique additionnelle et vers l'adaptation aux nouvelles architectures.

1.4 Positionnement et objectifs

Les travaux réalisés dans cette thèse portent sur l'optimisation des aspects relatifs à la parallélisation de l'application Gysela. Ils sont réalisés en deux parties : une première axée sur la modification de l'opérateur de gyromoyenne au sein du code Gysela et une deuxième orientée vers le développement d'un nouveau prototype et l'étude de nouvelles solutions de parallélisation afin de porter le code vers les architectures exascales dans les meilleures conditions.

Opérateur de gyromoyenne

Les travaux sur la gyromoyenne entamés par Fabien Rozar [62] lors de son doctorat ont permis d'obtenir un opérateur numériquement fiable. Néanmoins, sa parallélisation n'est pas adaptée à toutes les sections du code où elle est utilisée. L'objectif est donc de proposer une nouvelle parallélisation et d'obtenir de meilleurs temps d'exécution vis-à-vis de l'autre version parallèle. L'empreinte mémoire de l'opérateur, qui double actuellement la fonction de distribution 5D, doit également être revue à la baisse.

Pour ce faire, un système de zones buffers autour des sous-domaines est mis en place ainsi qu'un schéma de communication économe entre processus voisins. Les différentes données à traiter sont alors divisées en sous-ensembles indépendants afin de diminuer le grain de parallélisation et de faciliter la mise en place d'un schéma de recouvrement calculs – communications. Le fonctionnement des caches est mis à profit grâce à des techniques de traitement par bloc au niveau des sous-ensembles. Ces optimisations sont éprouvées sur des cas de taille moyenne et permettent de gagner en temps d'exécution. De plus, un nouvel interpolateur de Lagrange est expérimenté et ses résultats numériques ainsi que ses performances sont comparés à l'interpolateur Hermite actuel.

Ce nouvel interpolateur est mis en place dans la perspective d'une évolution du maillage du plan poloïdal. Le maillage polaire uniforme a pour avantage sa simplicité, mais possède une répartition inégale des points en termes de densité. Un maillage polaire réduit et adaptatif est évalué sur un prototype 2D et donne des résultats concluants concernant la précision numérique.

Expérimentation de nouvelles solutions de parallélisation

L'intégration de ce maillage réduit dans le code Gysela ne peut se faire aisément et d'autres techniques de parallélisation doivent être envisagées. Un prototype 4D *drift-kinetic* est donc élaboré dans lequel le maillage réduit sera testé à échelle satisfaisante ainsi que le nouvel interpolateur. Le modèle de programmation par tâches est utilisé pour ce prototype afin de réduire les problèmes de synchronisation de Gysela et de tirer le maximum de profit des ressources de calculs. De fait, une structure de données adaptée sous forme de blocs de données 4D est utilisée. Ce découpage permet une création simplifiée des tâches. De plus, une structure de communication asynchrone est conçue grâce au multithreading et s'adapte parfaitement à la granularité des tâches pour leur permettre de s'ordonnancer de façon performante. Les opérateurs de Poisson et de Vlasov sont retravaillés de manière à s'adapter à la répartition des calculs par tâches. Ce prototype doit permettre d'évaluer au mieux cette nouvelle approche de parallélisation dans l'optique d'une refonte intégrale

de l'application GYSELA. L'objectif est de se doter de capacités étendues en terme de physique et de performance HPC.

Première partie

Étude de l'opérateur de gyromoyenne

Chapitre 2

Introduction à l'opérateur de gyromoyenne

Sommaire

Comme mentionné en introduction, l'opérateur de gyromoyenne est un outil central à la théorie gyrocinétique. Il permet de prendre en compte le mouvement de rotation rapide des particules autour des lignes de champs sans avoir à le simuler explicitement. Il est fréquemment utilisé dans le code Gysela : deux fois dans la boucle principale (voir section 1.3.1), mais également de nombreuses fois dans les différents diagnostics qui sont effectués durant l'exécution. Sa vitesse d'exécution est donc fondamentale pour les performances du code et son implémentation dans certaines parties du code est limitante.

Nous détaillerons dans un premier temps sa formulation ainsi que le schéma numérique utilisé dans le code Gysela en section 2.1.1. Par la suite, l'interpolateur d'Hermite utilisé pour la gyromoyenne sera introduit en section 2.1.2 ; de même, le schéma numérique utilisé est présenté en section 2.1.3. Finalement, la section 2.2 permet de préciser l'implémentation faite dans Gysela de l'opérateur de gyromoyenne et en analyse les performances.

2.1 Gyromoyenne

Dans un tokamak, les particules du plasma s'enroulent rapidement de façon hélicoïdale autour des centres-guides, trajectoires proches des lignes de champ magnétique. L'opérateur de gyromoyenne permet de passer de la fonction de distribution des particules (qui intègre le mouvement rapide) à la fonction de distribution des centres-guides (qui moyenne le mouvement rapide, se référer à la section 1.1.4). Dans le modèle gyrocinétique, la fréquence de rotation Ω_c des particules autour des centres-guides est considérée comme suffisamment

élevée pour que ce mouvement puisse être traité seulement dans le plan poloïdal (voir [28] pour les détails de l'approximation). Le moment magnétique μ permet de caractériser ce mouvement de giration des particules qui repose sur la valeur du rayon de Larmor $\rho_{\mathcal{L}}$. Ce dernier représente le rayon de giration des particules. Il est défini à partir de μ par l'équation (2.1) où B donne la norme du champ magnétique et q la charge de l'espèce de particules considérée :

$$\rho_{\mathcal{L}} = \frac{1}{\pi q}\sqrt{\frac{m\mu}{2B}}. \tag{2.1}$$

La gyromoyenne consiste alors à moyenner la fonction de distribution sur un cercle correspondant au mouvement de giration des particules autour des centres-guides.

2.1.1 Opérateur de gyromoyenne

L'opérateur de gyromoyenne $\mathcal{J}$ s'exprime en un point $\mathbf{x}$, pour une fonction de distribution f et un rayon de Larmor $\rho_{\mathcal{L}}$ donnés, par la relation (2.2) suivante

$$\mathcal{J}\cdot f(\mathbf{x},\mathbf{v}) = \frac{1}{2\pi}\int_0^{2\pi} f(\mathbf{x}+\mathbf{x}_{\rho_{\mathcal{L}}}(\alpha),\mathbf{v})\mathrm{d}\alpha \tag{2.2}$$

où le vecteur $\mathbf{x}_{\rho_{\mathcal{L}}}(\alpha) = \rho_{\mathcal{L}}\cos(\alpha)\mathbf{e}_x + \rho_{\mathcal{L}}\sin(\alpha)\mathbf{e}_y$ décrit un cercle de rayon $\rho_{\mathcal{L}}$ sur $\alpha \in [0, 2\pi]$. Le couple de vecteurs $(\mathbf{e}_x, \mathbf{e}_y)$ forme une base cartésienne du plan poloïdal. Comme mentionné en 1.3.1, une approximation de Padé était utilisée jusqu'à peu. Cette approximation de l'opérateur est détaillée dans [69]. Dans les grandes lignes, l'opérateur de gyromoyenne est d'abord exprimé dans l'espace de Fourier avec la transformée spatiale de f que s'écrit

$$f(\mathbf{x}) = \int_{\mathcal{R}^3} \widehat{f}(\mathbf{k})e^{i\mathbf{kx}}d\mathbf{k} \tag{2.3}$$

avec $\mathbf{k} = (k_\perp\cos(\phi), k_\perp\sin(\phi), k_\parallel)$ le vecteur d'onde. Après quelques calculs, l'opérateur peut être identifié à une multiplication par une fonction de Bessel d'ordre 0. Cette fonction est remplacée par une approximation de Padé en (2.4) :

$$J_0(k_\perp\rho_{\mathcal{L}})e \sim \frac{4}{4+(k_\perp\rho_{\mathcal{L}})^2}. \tag{2.4}$$

L'ensemble est ensuite transformé pour revenir dans l'espace des positions et l'opérateur de gyromoyenne peut alors s'exprimer

$$(1-\frac{\rho_{\mathcal{L}}^2}{4}\nabla_\perp^2)\mathcal{J}\cdot f(\mathbf{x},\mathbf{v}) \simeq f(\mathbf{x},\mathbf{v}). \tag{2.5}$$

Cette approximation grossière (voir Figure 2.1) suffisait jusqu'à présent, mais elle s'adapte désormais mal aux évolutions du code. Il est nécessaire d'avoir accès à de grands domaines dans le plan poloïdal et surtout de pouvoir utiliser différentes distributions de données (voir chapitres suivants), ce qui est rendu difficile par les transformées de Fourier. De plus, une meilleure approximation est nécessaire pour améliorer les capacités physiques du code. L'opérateur a donc été remplacé par une discrétisation de l'équation (2.2) lors des travaux menés dans [63]. Celle-ci se formule

$$\mathcal{J}\cdot f(\mathbf{x},\mathbf{v}) \approx \frac{1}{N_{\mathcal{L}}}\sum_{k=0}^{N-1} f(\mathbf{x}+\mathbf{x}_{\rho_{\mathcal{L}}}(\frac{k}{N_{\mathcal{L}}}),\mathbf{v}) \tag{2.6}$$

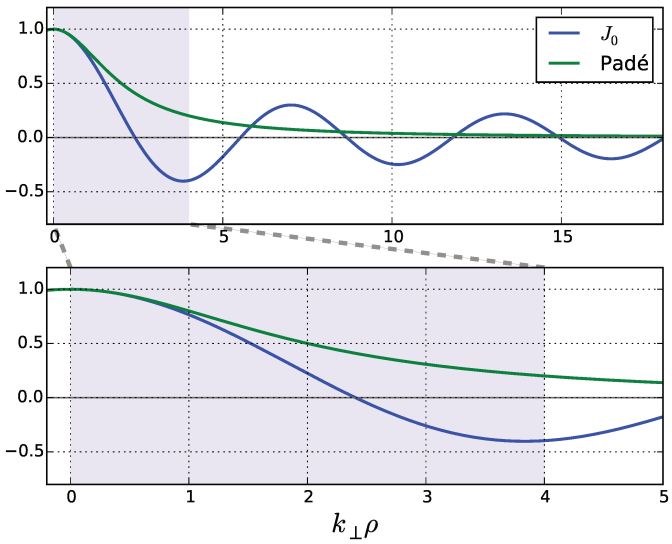


FIGURE 2.1 – Comparaison entre des valeurs de la fonction de Bessel J_0 et son approximation de Padé dans l'espace de Fourier pour la variable $k_\perp \rho_\mathcal{L}$ (voir équation (2.4)). Extrait de [45].

où $\mathbf{x}_{\rho_\mathcal{L}}(\frac{k}{N_\mathcal{L}}) = \rho_\mathcal{L} \cos(2\pi \frac{k}{N_\mathcal{L}})\mathbf{e}_x + \rho_\mathcal{L} \sin(2\pi \frac{k}{N_\mathcal{L}})\mathbf{e}_y$ et $N_\mathcal{L}$ est le nombre de points utilisés pour approcher l'intégrale.

On appellera cercle de Larmor le cercle centré en un point où l'on désire calculer la gyromoyenne et de rayon $\rho_\mathcal{L}$ (rayon de Larmor). À $\rho_\mathcal{L}$ donné, la précision de l'opérateur de gyromoyenne est directement liée à $N_\mathcal{L}$ sans que la position de ces points sur le cercle de Larmor n'influe beaucoup, tant qu'ils restent presque uniformément répartis (voir la section 5.3.2 dans [62]). Ce calcul peut être représenté graphiquement comme en Figure 2.2. Ainsi, l'application de l'opérateur de gyromoyenne au point ● se fait en moyennant les valeurs de la fonction aux $N_\mathcal{L} = 5$ points ▲, répartis uniformément sur le cercle de Larmor. Comme ces points correspondent rarement avec un point du maillage, une interpolation est nécessaire et est effectuée avec des polynômes d'Hermite de degré trois. Les points ■ sont les points auxquels la valeur et les dérivées de la fonction de distribution sont relevées pour être combinées dans l'interpolation d'Hermite. Si un point du cercle de Larmor se retrouve hors du plan poloïdal ($r > r_{\max}$ ou $r < r_{\min}$), une projection radiale sur le rayon $r_{\min}$ ou $r_{\max}$le plus proche est effectuée.

L'interpolation d'Hermite a été préférée aux splines cubiques utilisées pour les interpolations dans GYSELA. Bien que moins précise que ces dernières, l'interpolation d'Hermite offre néanmoins une précision ajustable (avec $N_\mathcal{L}$) et suffisante (comparativement à Padé, voir Figure 2.3). D'autre part, elle présente un caractère local (*stencil* compact) avantageux dans le cadre de la parallélisation.

2.1.2 Interpolation d'Hermite

L'interpolateur d'Hermite permet de reconstruire une fonction f de classe $\mathcal{C}^1$ sur un maillage discretisé. Cette reconstruction fait correspondre valeurs et dérivées de la fonction initiale et de la fonction reconstruite aux points du maillage. Considérons un cas 1D avec

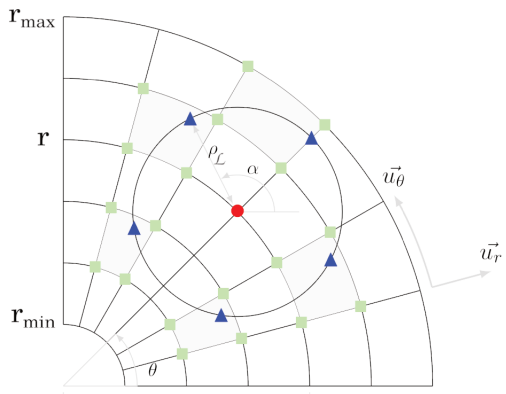
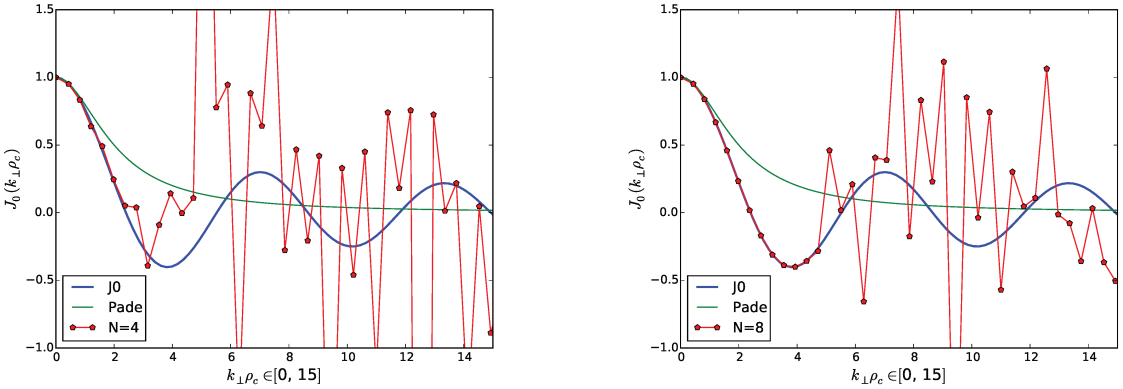


FIGURE 2.2 – Calcul de la gyromoyenne discrète basée sur l'interpolation d'Hermite.



(a) $N_{\mathcal{L}} = 4$. (b) $N_{\mathcal{L}} = 8$.

FIGURE 2.3 – Comparaison de la précision de l'approximation de la fonction de Bessel J_0 par Padé et par discrétisation et interpolation d'Hermite dans l'espace de Fourier. Extrait de [45].

une discrétisation uniforme en N points $(x_i)_{i\in[\![0,N-1]\!]}$ tel que $\forall i \in [\![0, N-1]\!]$, $x_i = x_0 + i\cdot dx$, avec dx et x_0 donnés. Alors pour une fonction f de classe $\mathcal{C}^1$ donnée, la reconstruction d'Hermite $\mathcal{H}(f)$ de cette fonction est $\mathcal{C}^1$ et assure que

$$\forall i \in [\![0, N-1]\!], \quad \begin{cases} \mathcal{H}(f)(x_i) = f(x_i) \\ \frac{\mathrm{d}\mathcal{H}(f)}{\mathrm{d}x}(x_i) = \frac{\mathrm{d}f}{\mathrm{d}x}(x_i). \end{cases} \tag{2.7}$$

En pratique, $\mathcal{H}(f)$ est un polynôme de degré $2N-1$ dit interpolateur et osculateur. Il peut être construit à partir des polynômes de Lagrange L_i par la relation

$$\mathcal{H}(f)(X) = \sum_{i=0}^{N-1} L_i^2(X) P_i(X) \tag{2.8}$$

où les polynômes de Lagrange sont donnés par

$$L_i(X) = \prod_{j=0, j\neq i}^{N-1} \frac{X - x_j}{x_i - x_j}$$

et les polynômes P_i sont définis par l'équation

$$P_i(X) = f(x_i) + (X - x_i)(f'(x_i) - 2L_i(x_i)L_i'(x_i)f(x_i)). \tag{2.9}$$

La preuve de la construction ne sera pas présentée ici et le lecteur peut se référer à [71] pour plus de détails. Contrairement aux splines cubiques qui utilisent l'ensemble des points du maillage et font disparaître les conditions liées aux dérivées, l'interpolation d'Hermite nécessite de reconstruire explicitement les dérivées de la fonction qui ne sont pas connues a priori. Les dérivées sont reconstruites en utilisant un schéma de différences finies. Cette reconstruction rajoute une erreur supplémentaire dans l'interpolation qui reste néanmoins ajustable avec le nombre de points d'interpolation du schéma de différences finies. Par la suite, nous utiliserons un stencil centré à cinq points pour le calcul de la dérivée qui est le plus adapté à notre cas [70].

L'interpolateur d'Hermite présente l'avantage, contrairement aux polynômes de Lagrange, de borner l'erreur commise pour toutes fonctions $\mathcal{C}^{2N}$. Il reste néanmoins, comme l'interpolation de Lagrange, sujet à de fortes oscillations parasites si le maillage est équidistant. Cet effet porte le nom de phénomène de Runge [15] et est illustré en Figure 2.4. Si l'ensemble du domaine est considéré pour une interpolation (ici Hermite sept points), le degré élevé du polynôme interpolateur (ici degré 13) engendre de fortes divergences entre les points du maillage. Alors que considérer chaque cellule du maillage séparément (ici Hermite deux points) génère une suite de polynômes de degré 3 qui se raccordent de façon $\mathcal{C}_1$ sur les points du maillage et suivent la fonction au plus près. Il est donc préférable d'utiliser l'interpolation d'Hermite de manière locale. C'est également un atout pour la parallélisation.

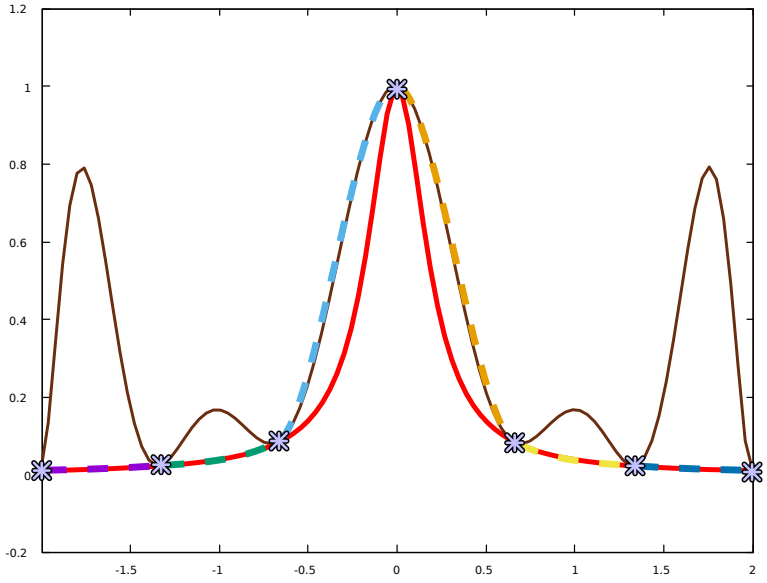


FIGURE 2.4 – Représentation de la fonction cloche de Runge (rouge), d'une interpolation d'Hermite 7 points sur l'ensemble du domaine (marron) et des interpolations d'Hermite 2 points sur chacun des intervalles (pointillés).

C'est cette approche appelée splines cubiques d'Hermite qui est utilisée dans l'opérateur de gyromoyenne. La reconstruction d'Hermite $\mathcal{H}(f)$ est alors constituée d'un ensemble

de polynômes de degrés trois $(H_i)_{i\in[\![0,N-1]\!]}$ qui respectent les conditions (2.7) sur leur intervalle respectif. Sous certaines conditions détaillées dans [62] (nombre de points pair dans le stencil du calcul des dérivées par différences finies), $\mathcal{H}(f)$ est $\mathcal{C}^1$ et les polynômes H_i peuvent s'écrire de la manière suivante :

$$H_i(X) = \varphi_1(X)f(x_i) + \varphi_2(X)f(x_{i+1}) + \Delta x(\varphi_3(X)f'(x_i) + \varphi_4(X)f'(x_{i+1})) \tag{2.10}$$

où Δx le pas du maillage et φ_1, φ_2, φ_3 et φ_4 sont des polynômes de degrés 3. Leur expression est directement déduite des conditions (2.7) et ils forment ce que l'on appelle la base d'Hermite (voir [62]). La base d'Hermite étant connue et constante, les différents H_i peuvent être recalculés à chaque changement de la fonction et stockés jusqu'à la prochaine modification, ou bien être simplement calculés à la volée lorsqu'une interpolation est nécessaire. De par la complexité du calcul, cette dernière approche est privilégiée.

Dans le cas d'un maillage 2D, l'interpolation d'Hermite combine les deux dimensions. Elle peut être considérée comme produit tensoriel de deux interpolations 1D. Ainsi, l'interpolation d'Hermite dans le plan $r \times \theta$ est effectuée en enchaînant une interpolation d'Hermite dans la dimension θ puis une interpolation d'Hermite dans la dimension r. Une interpolation en quatre points (un de chaque côté dans chaque direction), qui a prouvé être suffisamment précise (voir [70]), peut être représentée graphiquement comme montré en Figure 2.5. L'interpolation en un point ▲ du cercle de Larmor se déroule ainsi : à partir des quatre coins ■ de la cellule, les deux points intermédiaires ♦ sont calculés en interpolant dans la direction θ, puis la valeur au point final est obtenue en interpolant dans la direction r.

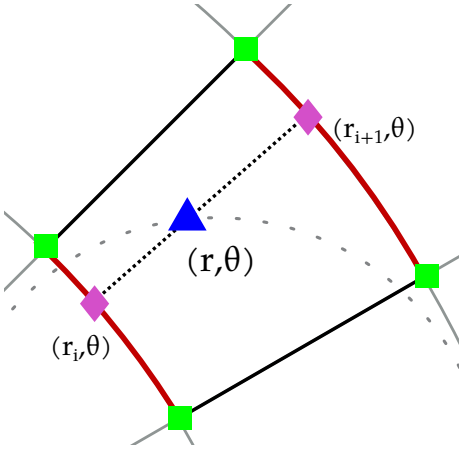


FIGURE 2.5 – Schéma d'une interpolation d'Hermite à 4 points.

L'interpolation d'Hermite offre un bon compromis entre précision, temps de calcul et usage mémoire. Contrairement aux splines, il est possible de ne pas stocker de coefficients nécessitant de scanner tout le domaine. Le caractère local de cet interpolateur est essentiel dans les développements présentés au chapitre 3 sur l'amélioration de la parallélisation dans GYSELA. Munis d'un opérateur d'interpolation d'Hermite, il est aussi possible de considérer l'application de la gyromoyenne à un sous-domaine comme un produit matriciel, facilitant ainsi l'introduction de sa parallélisation.

2.1.3 Schéma numérique

L'opérateur de gyromoyenne présenté en 2.1.1 peut être modélisé de façon matricielle afin d'optimiser les calculs [63] sous certaines conditions. Il est possible de choisir une distribution des points du cercle de Larmor sur le cercle fixe dans le temps et uniforme en espace sans impacter la précision numérique. Certains calculs peuvent alors être partiellement effectués à l'initialisation de la simulation. En effet, le maillage étant fixe dans le temps, le calcul de la gyromoyenne pour un point donné fait alors toujours intervenir les mêmes points du cercle de Larmor et donc les mêmes combinaisons linéaires de points du maillage. Il ne reste plus qu'à compléter cette combinaison avec les bonnes valeurs actualisées de la fonction.

Considérons un plan poloïdal $\mathcal{P}$ localisé en $(\varphi, v_\parallel, \mu)$. On représente les valeurs et les dérivées de la fonction de distribution f par le vecteur M_f de taille $4N_rN_\theta$, avec N_r et N_θ le nombre de points dans leur dimension respective (voir section 1.3). On sait alors construire une matrice de coefficients telle que, quel que soit le plan $\mathcal{P}$, le vecteur contenant la fonction de distribution gyromoyennée $M_{\mathcal{J}\cdot f}$ vaut :

$$M_{\mathcal{J}\cdot f} = M_{\text{coef}} \times M_f$$

avec

$$\begin{cases} M_{\mathcal{J}\cdot f} & \in \mathcal{M}_{N_rN_\theta,1}(\mathbb{R}), \\ M_{\text{coef}} & \in \mathcal{M}_{N_rN_\theta,4N_rN_\theta}(\mathbb{R}), \\ M_f & \in \mathcal{M}_{4N_rN_\theta,1}(\mathbb{R}). \end{cases}$$

La matrice M_f contient, pour chacun des N_rN_θ points du plan, les valeurs et les trois dérivées de la fonction (dérivées premières $\frac{\partial\cdot}{\partial r}, \frac{\partial\cdot}{\partial\theta}$ et dérivée croisée $\frac{\partial^2\cdot}{\partial r\partial\theta}$, d'où le facteur 4 pour les matrices de taille $4N_rN_\theta$). Celles-ci sont combinées aux coefficients des interpolations d'Hermite (relatifs aux points de Larmor) contenus dans M_{coef}. On remarque facilement que la matrice de coefficients M_{coef} est creuse puisque pour chaque point du maillage (r_i, θ_j), seuls quelques points du plan aux coins de cellules interceptées par le cercle de Larmor sont impliqués dans le calcul de la gyromoyenne. Leur nombre dépend du nombre de points utilisés pour la gyromoyenne $N_{\mathcal{L}}$, mais aussi du rayon de Larmor $\rho_{\mathcal{L}}$ et de la position radiale du point r (cela sera détaillé dans la section 3.1.1).

Cette représentation présente de nombreuses redondances évitables. D'une part, il n'est pas nécessaire de stocker l'ensemble des coefficients de tous les points du plan pour la gyromoyenne de chaque point du maillage. Un stockage compact est envisageable. D'autre part, on peut choisir de fixer la position des points du cercle de Larmor par rapport au référentiel polaire plutôt que cartésien. Si $\mathbf{p}_0^{\mathbf{x}_0}$ est le premier point du cercle de Larmor centré en $\mathbf{x_0}$, fixons l'angle formé par $\mathbf{e}_\theta$ et la droite passant par $\mathbf{p}_0^{\mathbf{x}_0}$ et $\mathbf{x_0}$ à une constante (0 par exemple). Alors, pour un rayon r_i donné et quel que soit l'angle θ_j du point à gyromoyenner, sa position relative par rapport aux points du cercle de Larmor est fixe. Prenons l'exemple de la Figure 2.6 et les deux points $\mathbf{x}_0$ et $\mathbf{x}_1$ situés sur le même rayon. Considérons les deuxièmes points de leur cercle de Larmor respectif $\mathbf{p}_2^{\mathbf{x}_0}$ et $\mathbf{p}_2^{\mathbf{x}_1}$. Alors en coordonnées polaires $\mathbf{p}_2^{\mathbf{x}_0} - \mathbf{x}_0 = \mathbf{p}_2^{\mathbf{x}_1} - \mathbf{x}_1$ et si $\mathbf{p}_2^{\mathbf{x}_0}$ est calculé par une combinaison linéaire des carrés verts alors $\mathbf{p}_2^{\mathbf{x}_1}$ est calculé par la même combinaison linéaire des carrés orange.

Cette symétrie axiale nous permet de ne considérer que les points se situant à $\theta = 0$ lors du calcul des coefficients des interpolations des points de Larmor. La matrice M_{coef} ne contient donc que les coefficients des points $(r_i, \theta{=}0)$. Le calcul de la gyromoyenne pour

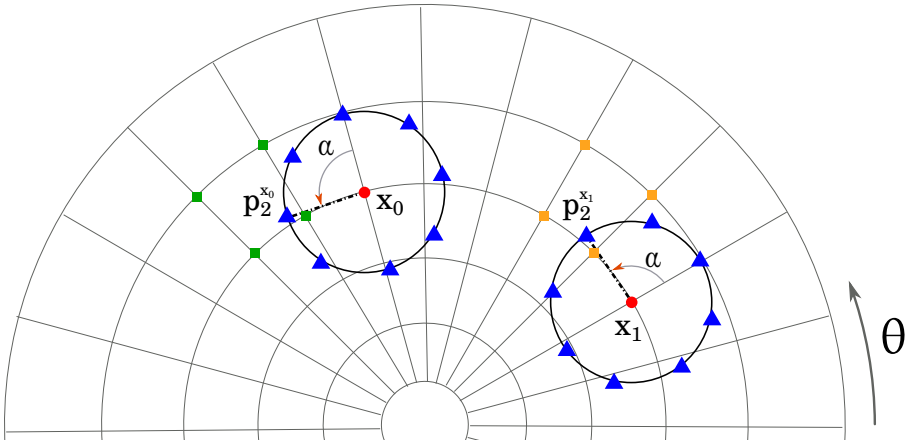


FIGURE 2.6 – Symétrie en θ de l'opérateur de gyromoyenne en x_0 et x_1 pour α donné.

un point (r_i, θ_j) du plan est alors effectué en utilisant les coefficients du point $(r_i, 0)$ et en ajoutant θ_j à l'index θ des points référencés par la matrice M_{coef}.

2.2 Mise en œuvre

2.2.1 Implémentation

L'implémentation dans GYSELA du schéma numérique présenté dans les sections précédentes a été réalisée pour la distribution de données $\mathbf{D}_2$, c'est-à-dire que les sous-domaines sont constitués de plans poloïdaux (voir 1.3.1). La matrice de coefficients contient donc un jeu de coefficients stockés de manière compacte pour chaque rayon r_i du maillage. M_{coef} est représentée par un tableau de dimension N_r dont chaque case pointe vers deux tableaux : un contenant les coordonnées relatives des points intervenant dans l'interpolation des points de Larmor de chaque point $(r_i, 0)$, et un contenant tous les coefficients non nuls associés aux valeurs et dérivées de ces points. Ainsi la matrice M_{coef} est réduite au minimum de données possible. De plus, il faut noter que le problème est invariant dans le temps, M_{coef} donc est créée et remplie une fois pour toutes à l'initialisation du programme.

Pour chaque point de Larmor, l'interpolation d'Hermite combine les valeurs et dérivées des quatre points les plus proches du point à interpoler, formant une cellule $(r \times \theta)$. Le calcul des dérivées est effectué en utilisant un *stencil* carré de 5×5 points [49]. Pour l'ensemble des quatre coins de la cellule, le calcul des dérivées combine les valeurs de trente-six points au total. Ainsi, si chaque point du cercle de Larmor est situé dans une cellule distincte, le nombre de points du maillage impliqués dans une gyromoyenne est au plus de $36N_{\mathcal{L}}$. Si plusieurs points du cercle sont situés dans la même cellule ou dans des cellules adjacentes, les contributions sont simplement sommées dans M_{coef}, réduisant l'empreinte mémoire. La taille totale de la matrice de coefficients est donc bornée et $|M_{\text{coef}}| \leq 36N_rN_{\mathcal{L}}$. Nous nous intéresserons d'abord à un cas où il n'est nécessaire de n'avoir qu'une seule valeur de μ (donc qu'un seul rayon de Larmor) par processus comme c'est le cas dans le diagnostic qui nous servira de terrain d'expérimentation (voir section 2.2.3). Une unique matrice M_{coef} par processus est alors requise. Les simulations lancées avec GYSELA le sont généralement avec $N_{\text{L}} = 8$, ce qui assure une précision suffisante au regard des autres opérateurs du

code (voir section 5.3 dans [62]). L'empreinte mémoire de la matrice de coefficients est donc inférieure à celle d'un plan $r\times\theta$. La matrice M_f est construite au début d'un appel à l'opérateur de gyromoyenne pour un plan et sa taille est fixe : $4N_r \times N_\theta$.

2.2.2 Complexité calculatoire

La complexité de l'opérateur de gyromoyenne en termes de calcul dépend de plusieurs paramètres. La complexité du calcul des dérivées en un point est en $O(N_{\text{deriv}}^2)$, où N_{deriv} est la taille du stencil (5 dans notre cas). On a donc une complexité de $O(N_{\text{deriv}}^2 N_r N_\theta)$ pour la construction de M_f. Le calcul de la gyromoyenne engendre un nombre de calculs proportionnel à $O(N_{\text{herm}} N_{\text{deriv}}^2 N_{\mathcal{L}} N_r N_\theta)$ où N_{herm} représente le nombre de points dans l'interpolation d'Hermite (4 dans notre cas, deux dans chaque dimension), et donc $N_{\text{herm}} N_{\text{deriv}}^2$ est le coût calculatoire d'une interpolation d'Hermite. On remarque que la complexité du calcul de la gyromoyenne ne dépend pas du rayon de Larmor $\rho_{\mathcal{L}}$, sauf dans le cas où celui-ci est réduit suffisamment pour que le cercle de Larmor n'intercepte que peu de cellules (adjacentes au point), voire aucune, autour du point à gyromoyenner. La complexité ne dépend alors plus du nombre de points de Larmor puisque tous sont inclus dans les mêmes quelques cellules. Elle peut s'écrire dans ce cas $O(N_{\text{herm}} N_{\text{deriv}}^2 N_r N_\theta)$. Comme il n'y a que peu de dépendance en μ sur la complexité (un facteur $N_{\mathcal{L}}$ pour les grands rayons), il ne devrait pas avoir de déséquilibre important au niveau de la charge de calcul entre les clusters de processus en μ.

Cette implémentation présente l'avantage de ne pas être excessivement complexe à mettre en place et de bénéficier d'une simplification géométrique. Une autre solution aurait toutefois pu être envisagée quant à la construction des matrices M_f et M_{coeff}. Il est possible de réduire le coût de stockage en ne calculant pas a priori les dérivées, mais en intégrant les coefficients du schéma de différences finies dans la matrice de coefficients. Ainsi, la matrice M_f ne contient plus que les valeurs de la fonction et sa taille est divisée par quatre. Les calculs des coefficients des dérivées doivent néanmoins toujours être effectués, mais sans aller chercher les valeurs aux points. De plus, la complexité des calculs de la gyromoyenne est portée à $O((N_{\text{herm}} + \frac{N_{\text{deriv}}}{2})^2 N_{\mathcal{L}} N_r N_\theta)$, qui est supérieure au coût précédent. Comme souvent en calcul parallèle, améliorer les performances d'un algorithme pour un critère (coût calculatoire, empreinte mémoire, volume de communication) entraîne un report de coûts sur un autre critère.

2.2.3 Étude des coûts

Cette implémentation de l'opérateur de gyromoyenne apporte une meilleure précision numérique, mais nécessite encore d'avoir l'ensemble d'un plan poloïdal en mémoire locale, c'est-à-dire d'avoir les données en distribution $\mathbf{D}_2$. Néanmoins, certains appels à la gyromoyenne sont effectués en distribution $\mathbf{D}_1$ et impliquent donc des transpositions de l'ensemble de la fonction de distribution 5D, au début et à la fin de l'opérateur pour revenir à la distribution initiale.

Dans la suite de cette étude, nous considérerons l'implémentation de l'opérateur de gyromoyenne dans le diagnostic qui calcule les moments fluides de GYSELA. Celui-ci se trouve dans la configuration où il est nécessaire d'effectuer les transpositions directes et inverses spécialement pour la gyromoyenne. Il permet d'extraire le moment fluide de la fonction de distribution en effectuant une intégrale pondérée et nécessite l'application d'une gyromoyenne qui représente son coût principal. L'algorithme 3 montre comment l'opérateur

Algorithme 3 : Gyromoyenne Hermite dans GYSELA.

Entrées : f, $N_{L\varphi}$ et $N_{Lv_\parallel}$
Sorties : $\mathcal{J}.f$ et p_f

```
1 f^T ← transpose_forward(f)
  pour i : 0 → N_Lφ − 1 faire en parallèle OpenMP
      pour j : 0 → N_Lv∥ − 1 faire en parallèle OpenMP
2         P ← preprocessing(f^T(∗, ∗, i, j))
3         f^T(∗, ∗, i, j) ← gyroaverage(P)
4         postprocessing(f^T(∗, ∗, i, j))
5 f ← transpose_backward(f^T)
```

de gyromoyenne en interpolation d'Hermite a été intégré dans ce diagnostic. Les quantités $N_{L\varphi}$ et $N_{Lv_\parallel}$ représentent les dimensions locales des sous-domaines dans les directions φ et $v_\parallel$. Les lignes `1` et `5` correspondent au changement de distribution de données. L'étape de `preprocessing` (ligne `2`) effectue l'application d'un traitement physique préalable nécessaire aux diagnostics. L'appel à `gyroaverage` (ligne `3`) correspond à l'opérateur décrit plus haut : calcul des dérivées des points du plan et multiplication par la matrice de coefficients. Le `postprocessing` (ligne `4`) permet d'extraire les données (moments fluides p_f) de la fonction gyromoyennée en appliquant un certain nombre de traitements.

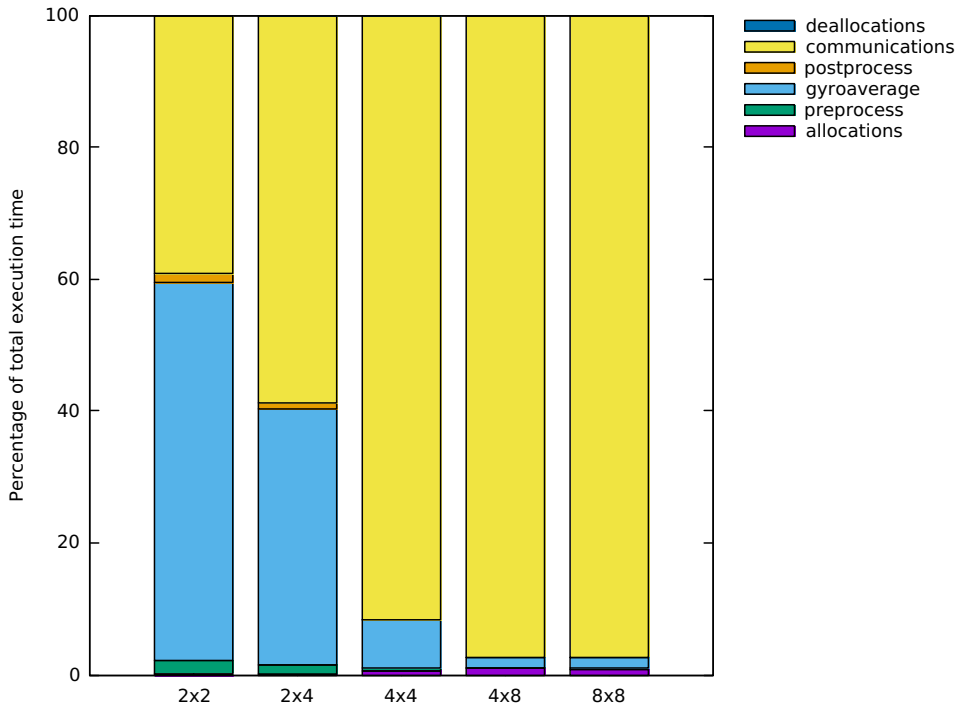


FIGURE 2.7 – Pourcentage du temps d'exécution moyen des différentes parties du diagnostic moments fluides pour un maillage de 512×512×128×64×1 sur différentes distributions de processus en $r \times \theta$ (de 4 à 64 processus MPI) avec 8 *threads*.

La Figure 2.7 représente le pourcentage des différents temps d'exécution moyens lors d'un appel à ce diagnostic. Cette simulation a utilisé un maillage de $512 \times 512 \times 128 \times 64 \times 1$ et a été exécutée sur le cluster Poincaré de la Maison de la Simulation à l'IDRIS. Chaque nœud est composé de deux processeurs Intel Xeon E5-2670, chacun possédant huit cœurs et 16GiO de mémoire. On observe la prépondérance du temps de communication pour les grandes distributions. Ces coûts de communication pourraient être drastiquement réduits avec un opérateur de gyromoyenne pouvant s'exécuter directement en distribution $\mathbf{D}_1$, ce que nous investiguerons dans le chapitre suivant. Un échange de données serait certes toujours nécessaire, mais de par la localité de l'opérateur, le volume de données transférées serait réduit.

L'opérateur de gyromoyenne utilisant un schéma discretisé et l'interpolation d'Hermite présente de nombreux avantages. La discrétisation de la formule exacte apporte une meilleure précision que l'approximation de Padé pour un surcoût acceptable. Précision et coût calculatoire sont modulables avec le nombre de points de Larmor et la taille des stencils d'interpolation. L'utilisation d'une interpolation d'Hermite deux points présente un caractère local bénéfique pour les performances. En distribution $\mathbf{D}_1$, le volume de communication engendré par les transpositions de la fonction 5D est rédhibitoire et un nouvel algorithme adapté à cette distribution de données est nécessaire.

Chapitre 3

Parallélisation de l'opérateur de gyromoyenne

Sommaire

Ce chapitre détaille un nouvel algorithme pour l'opérateur de gyromoyenne utilisant l'interpolation d'Hermite présenté au chapitre 2. Il est adapté pour utiliser des données distribuées dans le plan poloïdal. Cette nouvelle parallélisation change de fait le schéma de communication mais le schéma numérique reste le même. Des optimisations sont apportées à la parallélisation intra-processus, notamment par l'instauration d'un traitement par blocs. Le schéma de communication est ensuite amélioré par un algorithme de recouvrement afin qu'un maximum de communications soient effectuées simultanément aux calculs. Dans le développement de cette parallélisation, le rayon central du plan poloïdal est considéré large (de l'ordre de $0.1\,r_{\max}$ [1]) et prépondérant devant le rayon de Larmor (c'est-à-dire $\rho_{\mathcal{L}} \ll r_{\min}$). Ces choix sont faits en cohérence avec hypothèses physiques de simulation utilisées lors de l'élaboration de cet algorithme.

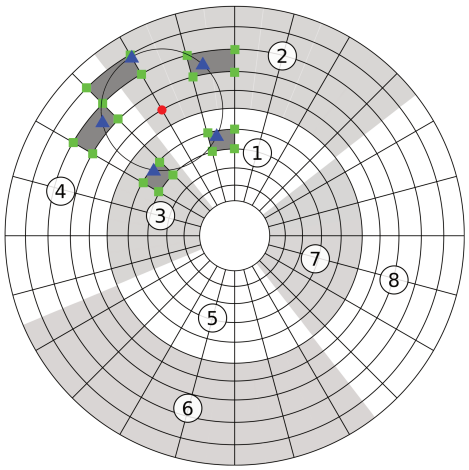
1. voir le maillage de Gysela en section 1.3

La section 3.1 présente une nouvelle parallélisation de l'opérateur de gyromoyenne en distribution $\mathbf{D}_1$ (voir section 1.3.1), à comparer avec celle présentée en section 2.2, ainsi qu'une étude des coûts associés. Différentes optimisations sont proposées en section 3.2. Leur efficacité est évaluée en section 3.3 et les performances des parallélisations de la gyromoyenne en distribution $\mathbf{D}_1$ et $\mathbf{D}_2$ sont comparées. Les travaux présentés dans ce chapitre ont fait l'objet d'une publication à IEEE [9].

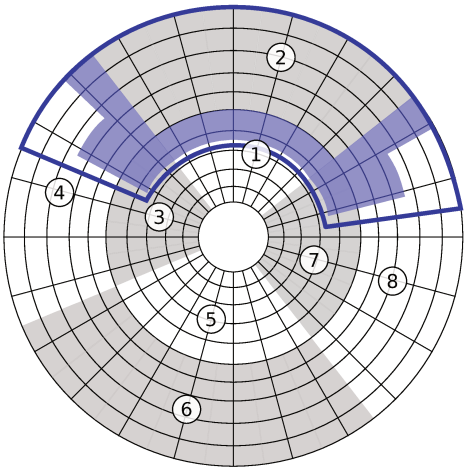
3.1 Parallélisation en distribution $\mathbf{D}_1$

Nous nous plaçons dans la configuration où les données sont réparties entre les processus selon la distribution $\mathbf{D}_1(P_{i,j}) = (iN_{Lr} : (i+1)N_{Lr},\ jN_{L\theta} : (j+1)N_{L\theta},\ *,\ *,\ \mu)$, représentée Figure 3.1a. Afin de calculer la gyromoyenne d'une fonction f sur un sous-domaine $r \times \theta$ du plan, il est nécessaire de disposer d'un certain nombre de données stockées sur des processus distants. En effet, les points localisés en bord de sous-domaine auront un cercle de Larmor qui sortira du domaine local. Des points appartenant à des sous-domaines adjacents seront ainsi nécessaires au calcul de leur gyromoyenne.

Un exemple est donné Figure 3.1a : le point ●, localisé sur le processus 2, est gyromoyenné grâce aux valeurs et dérivées des points ■, localisés sur le sous-domaine local et sur les processus adjacents 1, 3 et 4. Leurs dérivées sont obtenues en combinant les valeurs d'un certain nombre de points voisins (voir section 2.1.2).



(a) Plan poloïdal réparti en distribution $\mathbf{D}_2$ sur huit processus et exemple de contribution pour une gyromoyenne.



(b) Halo minimal nécessaire à la gyromoyenne (fond bleu transparent) et halo implémenté (ligne bleue foncé) pour le sous-domaine 2.

FIGURE 3.1 – Représentation géométrique du problème de gyromoyenne distribuée.

Afin de ne pas effectuer les communications séparément pour chaque point à gyromoyenner, ce qui ralentirait fortement l'exécution de l'opérateur, l'ensemble des données distantes requises pour l'application de la gyromoyenne au sous-domaine local sont préchargées dans une extension du sous-domaine appelée halo.

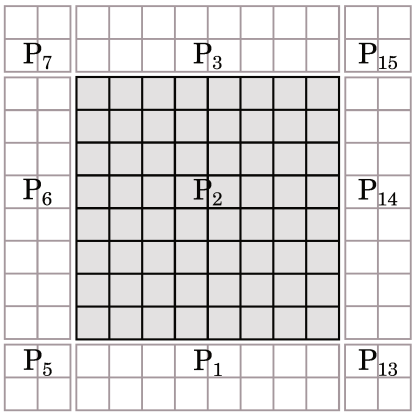
3.1.1 Modèle de communication par échange de halos

Le halo (aussi appelé zone tampon ou zone *buffer*) constitue une extension du sous-domaine local, mis à jour périodiquement avec l'ensemble des points distants nécessaires au calcul de la gyromoyenne de tous les points du sous-domaine (voir Figure 3.1b). Il est possible de parcourir tous les points d'un sous-domaine afin d'en extraire la liste exhaustive des points distants. En pratique, ceux-ci sont inclus dans un rectangle dont les dimensions peuvent être déterminées grâce aux caractéristiques de la simulation (voir Figure 3.1b). Il faut veiller à ce que la taille du halo reste suffisamment petite afin que les temps de communication ne deviennent pas prépondérants (ce qui ôterait l'intérêt de la méthode par rapport à l'algorithme avec transpositions). De plus, les conditions sur la taille du rayon central et du rayon de Larmor indiquées précédemment permettent de prendre comme hypothèse que seuls les processus directement adjacents seront concernés. Par exemple, les processus 1, 3, 4, 7 et 8 dans le cas du processus 2 sur la Figure 3.1b. Pour le processus 1, le processus 5 radialement opposé n'est pas considéré comme voisin. Une fois les halos identifiés, le schéma de communication consiste à envoyer à chaque processus voisin l'ensemble des points du sous-domaine local qui interviennent dans son halo, et ce pour chaque plan poloïdal.

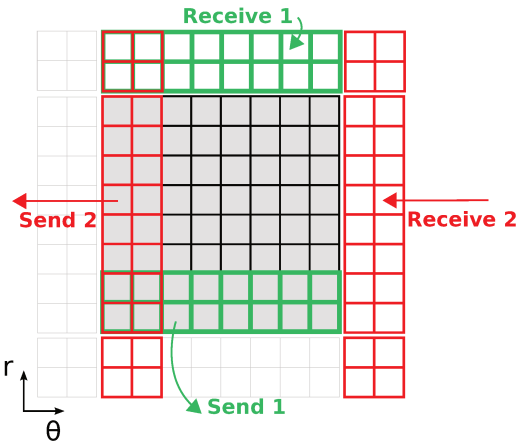
Un halo est représenté de façon logique tel que montré en Figure 3.2a. Le sous-domaine local est en gris et le reste représente le halo. Un sous-domaine a son halo distribué sur au plus huit processus distants. À noter que pour un processus à l'intérieur ou à l'extérieur du plan ($r_{\min}$ ou $r_{\max}$), une condition de bord de Neumann est appliquée à la partie du halo hors du plan. Pour constituer son halo, un processus doit donc recevoir des données de huit processus (ou moins s'il est situé sur un bord). Il est néanmoins possible de réduire le nombre de communications effectuées à quatre en procédant de manière synchronisée. La communication des halos se fait en deux étapes comme décrit en Figure 3.2b. Dans un premier temps, les parties supérieures et inférieures du halo (dans la direction r) sont échangées. Sur la figure, seule la mise à jour inférieure est représentée (en vert) pour une meilleure lisibilité. Dans un second temps, les parties latérales et les coins du halo sont échangés en une fois, après avoir reçu les parties supérieures et inférieures. La mise à jour vers la gauche est dessinée sur la figure et montre bien qu'il est nécessaire d'avoir reçu la partie verte des halos afin de pouvoir mettre à jour les coins. En procédant ainsi, on évite les communications avec les processus en coins, au prix d'une synchronisation entre les étapes d'échange dans les dimensions r et θ.

Une fois les halos échangés et toutes les données nécessaires disponibles localement, les dérivées de l'ensemble des points du sous-domaine et du halo sont calculées, puis les valeurs de la gyromoyenne de la fonction sur le sous-domaine sont calculées en utilisant la matrice de coefficients. Il est à noter que les dérivés de certains points sont calculés plusieurs fois : une fois dans leur sous-domaine respectif et éventuellement dans les halos des processus voisins (en fonction de leur position). Cela entraîne une redondance des calculs et une augmentation du coût calculatoire de l'opérateur. Nous appellerons cet algorithme *gyromoyenne distribuée*. L'algorithme utilisant la distribution $\mathbf{D}_2$ (section 2.2) sera désigné lui par l'appellation *gyromoyenne transposée*.

Il faut cependant remarquer que pour un même plan poloïdal, la taille du halo n'est pas la même pour tous les processus. Prenons l'exemple de la Figure 3.3. Pour un cercle de Larmor donné, si celui-ci est situé plus proche du centre poloïdal il interceptera un angle poloïdal plus important. Dans l'exemple, le cercle le plus éloigné du centre intercepte un angle d'environ $1.5\Delta\theta$ (deux cellules en θ), alors que le cercle le plus au centre intercepte



(a) Représentation logique du sous-domaine du processeur 2 et de son halo pour une distribution du plan poloïdal sur 16 processus.



(b) Mises à jour inférieures et vers la gauche du schéma de communication des halos.

FIGURE 3.2

un angle d'environ $6\Delta\theta$ (six cellules). Ainsi, plus un processus est proche du centre, plus son halo sera étendu dans la direction θ. La taille du halo est en fait la même pour tous les processus situés sur une même couronne. On appelle «couronne», un ensemble de processus se partageant la même zone radiale (les processus 1, 3, 5 et 7 sur la Figure 3.1a par exemple).

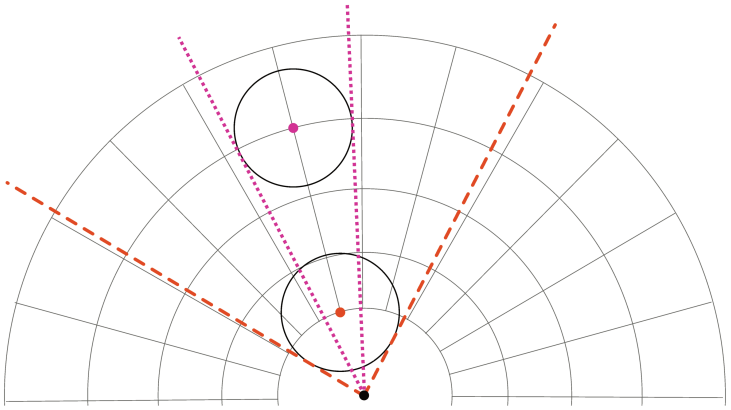


FIGURE 3.3 – Différence d'interception angulaire d'un même cercle de Larmor situé à différents rayons.

Les éléments proposés permettent d'effectuer une première implémentation de cette version distribuée de l'opérateur de gyromoyenne.

3.1.2 Implémentation de la gyromoyenne distribuée

L'algorithme s'applique considérant en entrée la distribution de données $\mathbf{D}_1(P_{i,j}) = (iN_{Lr} : (i+1)N_{Lr},\ jN_{L\theta} : (j+1)N_{L\theta},\ *,\ *,\ \mu)$; chaque processus doit effectuer le calcul de la gyromoyenne pour $N_\varphi \times N_{v_\parallel}$ sous-domaines poloïdaux. Afin de mettre en perspective les différentes optimisations développées dans cette section, nous nous focaliserons sur leur implémentation dans le diagnostic sur les moments fluides. Celui-ci se situe dans une zone critique proche du pic mémoire et peut bénéficier de l'utilisation de la gyromoyenne

distribuée. Son déroulement est décrit par l'algorithme 4 pour un processus MPI. La phase 1 consiste en l'extraction d'un sous-domaine poloïdal de f suivi d'un pré-traitement permettant de calculer des grandeurs physiques. La phase 2 constitue l'échange de halos par communication point à point synchrone (`MPI_Sendrecv()`) avant l'application de l'opérateur en phase 3. La contribution du sous-domaine de f aux moments fluides est extraite en phase 4 avant de recopier la fonction gyromoyennée dans la fonction initiale pour la suite des diagnostics (phase 5).

Algorithme 4 : Diagnostic sur les moments fluides avec gyromoyenne distribuée.

Entrée : f
Sorties : $\mathcal{J}.f$, $\mathcal{M}_{\text{fluid}}$

`/*` r_l, r_u, θ_l `et` θ_u `sont les bornes du sous-domaine local en` $\mathbf{D}_1$ `*/`
pour $i : 0 \to N_\varphi N_{v_\parallel} - 1$ **faire** en parallèle **OpenMP**
    $i_\varphi \leftarrow i \mod N_\varphi$
    $i_v \leftarrow i \div N_\varphi$
1   $f_{\text{tmp}} \leftarrow$ `preprocess(`$f(r_l : r_u, \theta_l : \theta_u, \varphi_{i_\varphi}, v_{\parallel i_v}, \mu)$`)`
2   `send_receive_halo(`f_{tmp}`)`
3   `gyroaverage(`f_{tmp}`)`
4   $\mathcal{M}_{\text{fluid}} \leftarrow +$ `postprocess(`f_{tmp}`)`
5   $f(r_l : r_u, \theta_l : \theta_u, \varphi_{i_\varphi}, v_{\parallel i_v}, \mu) \leftarrow f_{\text{tmp}}$

Dans une optique de simplification de l'implémentation (voir section 1.2.1 – «Problématique de calcul parallèle»), une taille unique de halo est fixée. Le maximum de l'ensemble des tailles de halos est donc choisi, il correspond toujours à la taille de halo de la couronne de processus la plus proche du centre du plan poloïdal. Ce choix a un impact sur les coûts des communications et sur l'empreinte mémoire qui sont étudiés en section 3.1.3. Il est en effet important d'estimer l'ensemble des différents coûts de cet opérateur de gyromoyenne distribuée afin d'évaluer les gains qui peuvent en être tirés.

3.1.3 Coûts en mémoire, calcul et communication

Les coûts de communication et l'empreinte mémoire de l'opérateur de gyromoyenne distribuée augmentent proportionnellement à la taille des halos. Le nombre de points qu'il est nécessaire de récupérer sur les processus distants dépend de nombreux facteurs.

Premièrement, plus le rayon de Larmor est grand, plus il faut aller chercher des points loin dans les sous-domaines distants. Le rayon de Larmor est directement corrélé à la taille du halo. Deuxièmement, plus le nombre de points sur le cercle de Larmor $N_{\mathcal{L}}$ est grand, plus il y a de points à utiliser pour les interpolations et donc plus de données distantes sont nécessaires pour une gyromoyenne. En pratique, si l'on considère l'ensemble des points du sous-domaine, les points nécessaires à leur gyromoyenne constituent une bande entière de points au bord du sous-domaine des processus distants. L'influence de $N_{\mathcal{L}}$ sur la taille du halo est donc négligeable mais modifie les coûts de calcul (voir section 2.2.2). Troisièmement, plus la discrétisation du maillage est fine, plus le rayon de Larmor est grand devant un pas du maillage. En conclusion, ce sont les ratios $\frac{\rho_{\mathcal{L}}}{\Delta r}$ et $\frac{\rho_{\mathcal{L}}}{\Delta \theta}$ qui sont décisifs pour le volume des communications.

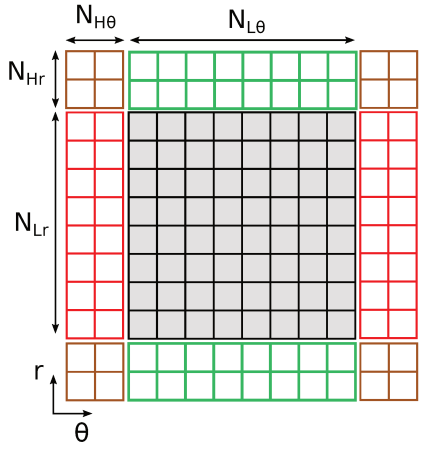
Dimensions du halo

Pour faciliter l'implémentation, le halo est légèrement agrandi pour former un rectangle logique englobant tous les points nécessaires (voir Figure 3.1b), ceci afin d'avoir des dimensions fixes, un schéma de communication simple et uniforme et un stockage régulier et logique des points. Nous notons les dimensions du halo $N_{\mathrm{H}r}$ $N_{\mathrm{H}\theta}$ telles que décrites en Figure 3.4a. La valeur de $N_{\mathrm{H}r}$ est aisément calculable ; en effet, la distance entre deux points du maillage radialement consécutifs (r_i, θ_j) et $(r_i + 1, \theta_j)$ est la même quels que soient i et j. Ainsi

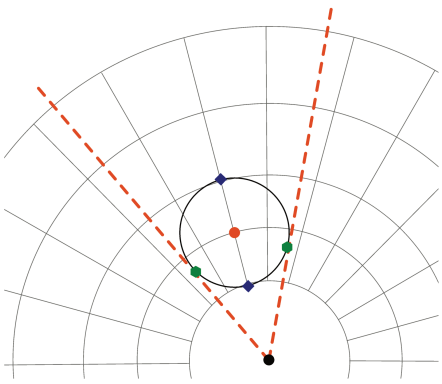
$$N_{\mathrm{H}r} = \lceil \frac{\rho_{\mathcal{L}}}{\Delta r} \rceil + \lceil \frac{N_{\mathrm{deriv}}}{2} \rceil \tag{3.1}$$

où N_{deriv} est le nombre de points utilisés pour le stencil de calcul des dérivées. La valeur de $N_{\mathrm{H}\theta}$ est plus complexe à estimer puisque la distance $r\Delta\theta$ entre deux points (r_i, θ_j) et $(r_i, \theta_j + 1)$ poloïdalement consécutifs varie en fonction du rayon. Le point du cercle de Larmor ayant la coordonnée θ la plus éloignée de celle de son centre est le point de la droite passant par le centre du plan poloïdal et tangente au cercle de Larmor, tel que représenté en Figure 3.4b. La coordonnée radiale du centre du cercle de Larmor et le rayon de Larmor étant connus, il est possible d'en déduire l'angle entre le point de contact et le centre du cercle de Larmor et de là, la taille du halo dans la direction θ par la relation :

$$N_{\mathrm{H}\theta}(r) = \lceil \frac{1}{\Delta\theta} \arcsin(\frac{\rho_{\mathcal{L}}}{r}) \rceil + \lceil \frac{N_{\mathrm{deriv}}}{2} \rceil. \tag{3.2}$$



(a) Dimensions d'un sous-domaine $r \times \theta$ et de son halo.



(b) Points de coordonnées r et θ extrêmes sur le cercle de Larmor.

FIGURE 3.4

Finalement, pour un sous-domaine de dimensions $N_{Lr} \times N_{L\theta}$ dont le rayon minimum vaut a et pour un rayon de Larmor $\rho_{\mathcal{L}}$ donné, la taille du halo N_{H} est donnée par la relation

$$N_{\mathrm{H}}(\rho_{\mathcal{L}}, a, N_{Lr}, N_{L\theta}) = 4N_{\mathrm{H}r}N_{\mathrm{H}\theta}(a) + 2(N_{\mathrm{H}r}N_{L\theta} + N_{\mathrm{H}\theta}(a)N_{Lr}). \tag{3.3}$$

Indice de couronne	0	1	2	3	4	5	6	7
Rayon minimal	0.1	0.2125	0.325	0.4375	0.55	0.6625	0.775	0.8875
$N_{\mathrm{H}r}$	15	15	15	15	15	15	15	15
$N_{\mathrm{H}\theta}$	20	11	9	7	6	6	6	5
N_{H}	10160	7316	6684	6052	5736	5736	5736	5420
$N_{\mathrm{H}} \div (N_{Lr} N_{L\theta})$	62%	44%	40%	36%	35%	35%	35%	33%

TABLE 3.1 – Dimensions de halo par position radiale des processus pour un maillage de 1024×1024 points divisé en 8×8 processus. $r \in [0.1; 1]$, $\rho_{\mathcal{L}} = 0.01$ et $N_{\mathrm{deriv}} = 5$.

Coût du halo unique

Comme indiqué en section 3.1.2, un schéma de communication unique a été mis en place en sélectionnant le halo le plus grand de l'ensemble des processus du plan poloïdal. Pour un rayon de Larmor $\rho_{\mathcal{L}}$ et une distribution $N_{Lr} \times N_{L\theta}$ données, chaque processus aura ainsi le même halo de taille $N_{\mathrm{H}}(r_{\min})$. Ce schéma présente l'avantage, en plus de simplifier l'implémentation, d'uniformiser le volume de communication des halos. Cela profite à l'algorithme de communication en deux temps du halo (voir section 3.1.1) puisque, quel que soit le processus, le volume de communication avant la synchronisation est le même. Ce halo de taille unique représente néanmoins un surcoût qu'il faut quantifier.

Prenons pour cela l'exemple d'un plan poloïdal de 1024×1024 points divisé en 8×8 processus. Prenons également r compris entre 0.1 et 1 et $\rho_{\mathcal{L}}$ fixé à 0.01, tous deux normalisés. Ces valeurs sont représentatives des plus grosses simulations effectuées en productions. Les dimensions des halos pour chaque couronne de processus sont données par la table 3.1 (N_{deriv} est fixé à 5, voire section 2.2.1). On remarque tout d'abord que $N_{\mathrm{H}\theta}$ croit rapidement lorsqu'on approche du rayon central, mais les hypothèses sur la géométrie du plan l'empêche de croître de façon nuisible à l'algorithme. D'autre part, bien que la dimension du halo en θ passe du simple au quadruple, la taille totale du halo n'augmente que d'un facteur deux entre les deux extrémités du plan. Simplifier le schéma de communication en adoptant un halo unique a donc un coût non négligeable que nous absorberons d'une autre façon ; ceci sera traité *via* différentes optimisations présentées en section 3.2.

Comparaison des coûts

Afin d'évaluer les gains attendus pour ce nouvel algorithme, nous allons comparer l'empreinte mémoire et le volume de communications avec la version transposée de l'opérateur. Nous estimerons également le surcoût calculatoire engendré par la réplication du calcul des dérivées dans les halos.

Concernant l'empreinte mémoire du halo, la dernière ligne de la table 3.1 nous donne la taille du halo relativement à la taille du sous-domaine. On observe globalement une augmentation de l'empreinte mémoire de 40% pour le stockage de l'ensemble des halos pour l'ensemble de la simulation (de 62% pour la couronne intérieure à 33% pour la couronne extérieure). Néanmoins, il n'est pas nécessaire de stocker simultanément l'ensemble des halos. En effet, il est possible de ne mettre à jour que quelques halos (typiquement le nombre de *threads*), puis de traiter les sous-domaines correspondant en parallélisation multicœurs avant de mettre à jour les halos suivants et ainsi de suite. Le surplus d'empreinte mémoire

engendré par la gyromoyenne distribuée est donc ajustable : pour le cas de la Table 3.1 avec $N_\varphi = 32$, $N_{v_\parallel} = 128$ et 16 *threads*, il vaut moins de 0.1% si l'on ne garde que 16 halos en simultané et donc 40% si l'on traite l'ensemble des halos en simultané. Ce coût reste dans tous les cas plus faible que celui de la gyromoyenne transposée qui dupliquait la fonction de distribution lors de la redistribution des données. Dans la plupart des cas, l'empreinte mémoire est réduite d'au moins 50%.

Le surcoût calculatoire engendré par le calcul redondant des dérivées des points présents dans le halo des processus voisins représente une augmentation faible mais non négligeable du temps de calcul. Tout d'abord, la quantité de données en charge de chaque processus reste la même en distribution $\mathbf{D}_1$ ou en distribution $\mathbf{D}_2$. Le surcoût provient donc du calcul des dérivées dans les halos et est donc de l'ordre de $O(N_{\text{deriv}}^2 N_{\text{H}} \texttt{Np}_r \texttt{Np}_\theta)$, avec $\texttt{Np}_r$ et $\texttt{Np}_\theta$ le nombre de processus dans les directions r et θ, N_{H} la taille d'un halo et N_{deriv} la taille du stencil pour le calcul des dérivées 1D. Le rapport entre les complexités du calcul de la gyromoyenne ($O(N_{\text{herm}} N_{\text{deriv}}^2 N_{\mathcal{L}} N_r N\theta)$, voir section 2.2.2) et du calcul des dérivées des halos est alors donné par

$$\frac{N_{\text{herm}} N_{\text{deriv}}^2 N_{\mathcal{L}} N_r N\theta}{N_{\text{deriv}}^2 N_{\text{H}} \texttt{Np}_r \texttt{Np}_\theta} = \frac{N_{\text{herm}} N_{\mathcal{L}} N_{Lr} N_{L\theta}}{N_{\text{H}}}. \tag{3.4}$$

La quantité $\frac{N_{Lr} N_{L\theta}}{N_{\text{H}}}$ est inférieure et rarement égale à 1. Ainsi, le surcoût engendré par la réplication du calcul des dérivées est inférieur à $(N_{\text{herm}} N_{\mathcal{L}})^{-1}$ fois le coût du calcul de la gyromoyenne. D'autre part, il est possible d'envisager d'effectuer le calcul des dérivées avant la communication des halos afin de ne pas créer de redondance de calcul. Il serait alors nécessaire de quadrupler le volume de communication, ce qui représenterait alors un surcoût réel.

Le volume de communication dans la gyromoyenne distribuée pour un processus et un plan poloïdal est de N_{H} répartie en quatre communications point à point avec les processus voisins. Cela représente pour l'ensemble du diagnostic un volume de communication de $N_{\text{H}} N_\varphi N_{v_\parallel} \texttt{Np}_r \texttt{Np}_\theta$. Pour la gyromoyenne transposée, la redistribution des données mobilise l'ensemble des données soit un volume de $N_r N_\theta N_\varphi N_{v_\parallel}$. Le volume est donc moins important pour la gyromoyenne distribuée. De plus, les échanges de données pour la version distribuée ne consistent qu'en quatre communications avec les processus voisins. La redistribution des données pour la version transposée effectue elle un échange de données avec l'ensemble des processus de la simulation (communication globale `All_to_all` entre $\texttt{Np}_r \texttt{Np}_\theta - 1$ processus, soit soixante-quatre processus dans l'exemple de la table 3.1).

Finalement, ce nouvel opérateur de gyromoyenne en distribution $\mathbf{D}_1$ présente une empreinte mémoire réduite d'au moins 50% dans la plupart des cas et des coûts de communications moins élevés que ceux de l'opérateur en distribution $\mathbf{D}_2$. De plus, il possède une plus grande flexibilité (plans poloïdaux indépendants et traitables séparément) et offre la possibilité d'ajouter de nombreuses optimisations.

3.2 Optimisation de l'opérateur de gyromoyenne distribuée

Si l'opérateur de gyromoyenne distribuée est implémenté uniquement tel que décrit par l'algorithme 4, il n'apporte qu'une légère amélioration du temps d'exécution du fait du nombre élevé de communications (quatre par processus et par plan). Il est possible de perfectionner cet algorithme en traitant les plans par *blocs* (section 3.2.1), ce qui permettra

d'exploiter au mieux le potentiel de parallélisation de cette version de la gyromoyenne. La répartition des temps d'exécution au sein de l'opérateur permet d'envisager une autre optimisation au niveau du déroulement des calculs et des communications (section 3.2.2).

3.2.1 Traitement par bloc de l'opérateur

Principe du découpage en blocs

Comme mentionné en section 3.1.3, il est possible d'échanger l'ensemble des $N_\varphi N_{v_\parallel}$ sous-domaines poloïdaux de chaque processus en une fois au début de l'opérateur de gyromoyenne de manière similaire à l'implémentation utilisant des transpositions. Cette approche n'offre néanmoins aucun gain en mémoire, alors que cet opérateur est utilisé dans une zone du code proche du pic mémoire. À l'inverse, ne procéder à l'échange de halos que plan poloïdal par plan poloïdal générerait autant communications que de processus et de plans, ce qui est à éviter pour que les coûts d'initialisation de communications ne deviennent prépondérants. Il faut donc trouver un juste milieu entre empreinte mémoire et quantité de communications en procédant au traitement des plans poloïdaux bloc par bloc. Cela consiste à faire en sorte que plusieurs plans poloïdaux consécutifs soient assemblés en un bloc et que leurs différents halos soient échangés grâce à un unique jeu de communications. Ainsi, une fois cette communication terminée, plusieurs plans sont prêts à être gyromoyennés et ceux-ci peuvent donc être répartis entre plusieurs *threads* et traités en parallèle.

L'opérateur de gyromoyenne distribuée peut alors s'exprimer tel que décrit par l'algorithme 5. Le paramètre b_s représentant la taille de bloc (le nombre de plans poloïdaux dans un bloc) est ajouté et peut être choisi de sorte à équilibrer empreinte mémoire et amélioration du temps d'exécution. Notons $N_{\text{block}} = \frac{N_\varphi N_{v_\parallel}}{b_s}$ le nombre total de blocs. Désormais, les sous-domaines de plusieurs plans successifs sont stockés dans la variable f_{block} de dimensions $b_s \times (N_{Lr} + 2N_{\text{Hr}}) \times (N_{L\theta} + 2N_{H\theta})$ lors de la phase 1. L'ensemble des halos du bloc sont échangés simultanément lors de la phase 2 grâce à un type MPI spécifique . Puis l'ensemble des sous-domaines des plans du bloc sont gyromoyennés, traités et recopiés avant de passer au bloc suivant. Les calculs des indices de plan i_φ et i_v sont omis pour une meilleure lisibilité et valent

$$\begin{aligned} i_v &= (i_{\text{block}} \times b_s + i) \mod N_\varphi \\ i_\varphi &= (i_{\text{block}} \times b_s + i) \div N_\varphi. \end{aligned}$$

Les phases 1, 3, 4 et 5 sont effectuées en parallèle grâce à des *threads* OpenMP. Les différentes itérations de chaque boucle sont réparties entre les *threads* qui les exécutent. À nombre d'itérations données, cette répartition reste la même afin de maximiser la localité des données ; chaque *thread* effectue l'ensemble des traitements pour une itération donnée. Cette distribution est dite statique. En choisissant une taille de bloc et un nombre de *threads* en puissance de deux, la répartition est assurée d'être équilibrée.

Impact de la taille des blocs

Comme mentionné précédemment, la taille des blocs est une variable qui permet d'ajuster l'empreinte mémoire, le volume des communications ainsi que la vitesse d'exécution

Algorithme 5 : Diagnostic avec gyromoyenne distribuée avec blocs.

Entrées : f, b_s
Sorties : $\mathcal{J}.f$, $\mathcal{M}_{\text{fluid}}$

`/* ` r_l`, `r_u`, `θ_l` et `θ_u` sont les bornes du sous-domaine local en ` $\mathbf{D}_1$ `*/`
pour $i_{\text{block}} : 0 \to N_{\text{block}} - 1$ **faire**
  pour $i : 0 \to b_s - 1$ **faire** en parallèle **OpenMP**
1 $f_{\text{block}}(i) \leftarrow$ `preprocess(`$f(r_l : r_u, \theta_l : \theta_u, \varphi_{i_\varphi}, v_{\parallel i_v}, \mu)$`)`
2 `send_receive_halos(`f_{block}`)`
  pour $i : 0 \to b_s - 1$ **faire** en parallèle **OpenMP**
3 `gyroaverage(`$f_{\text{block}}(i)$`)`
4 $\mathcal{M}_{\text{fluid}} \leftarrow$ `postprocess(`$f_{\text{block}}(i)$`)`
5 $f(r_l : r_u, \theta_l : \theta_u, \varphi_{i_\varphi}, v_{\parallel i_v}, \mu) \leftarrow f_{\text{block}}$

(intensité calculatoire). Pour ce faire, des simulations ont été exécutées afin de pouvoir tester différentes tailles de blocs. Ces expérimentations ont été menées sur le cluster Poincaré (16 cœurs par nœud, voir section 2.2.3).

La Figure 3.5 présente les temps d'exécution normalisés du diagnostic sur les moments fluides implémenté avec la gyromoyenne distribuée optimisée par blocs pour différentes configurations en fonction de la taille de bloc choisie. Les simulations utilisent un maillage de $(512 \times 512 \times 64 \times 31)$ points et le nombre de processus en $r \times \theta$ utilisé pour les différentes configurations est 2×2 (bleu), 4×4 (vert) et 8×8 (rouge). Chaque processus dispose d'un processeur et exécute 16 *threads*. La normalisation des résultats s'effectue en fonction du temps d'exécution pour une taille de bloc de 4. Des valeurs inférieures montrent un temps d'exécution trop important pour être considérées. Les courbes forment un plateau pour des tailles de blocs supérieures à celles représentées. Elles ne sont pas affichées pour une meilleure lisibilité. L'échelle en ordonnée commence à 0.55.

Ce graphique permet d'évaluer la taille de bloc optimale donnant le temps d'exécution le plus court en fonction de certains paramètres. On remarque dans un premier temps que le nombre de processeurs entre lesquels les données sont distribuées influe sur la taille de bloc optimale. Par exemple, pour $\mu = 4$, la taille de bloc optimale pour 64 processeurs est 64 mais elle est supérieure à 512 pour 4 processeurs. Pour un maillage donné, un plus grand nombre de processus signifie de plus petits domaines et donc la taille du halo par rapport à la taille du domaine devient plus importante. Le poids des communications s'accentue face à la quantité de calculs (voir les commentaires de la Figure 3.7). Ainsi, la taille de bloc optimal varie.

D'autre part, la valeur de μ joue également un rôle en modifiant le rayon de Larmor et donc la taille des halos. Pour 16 processus, la taille de bloc optimale est de 64 pour $\mu = 4$, mais se situe entre 8 et 16 pour $\mu = 8$. Plus le rayon de Larmor est grand, plus le halo l'est également et le ratio communication – calcul s'accroît et la taille de bloc optimale change.

On observe que la taille de bloc optimale varie en fonction de plusieurs paramètres. Il pourrait ainsi être utile d'avoir une taille de bloc calculée en fonction du maillage et de la répartition des données dans le plan poloïdal. De même, pour les cas de production s'exécutant avec plusieurs valeurs de μ, il serait intéressant d'avoir une taille de bloc spécifique pour chaque μ.

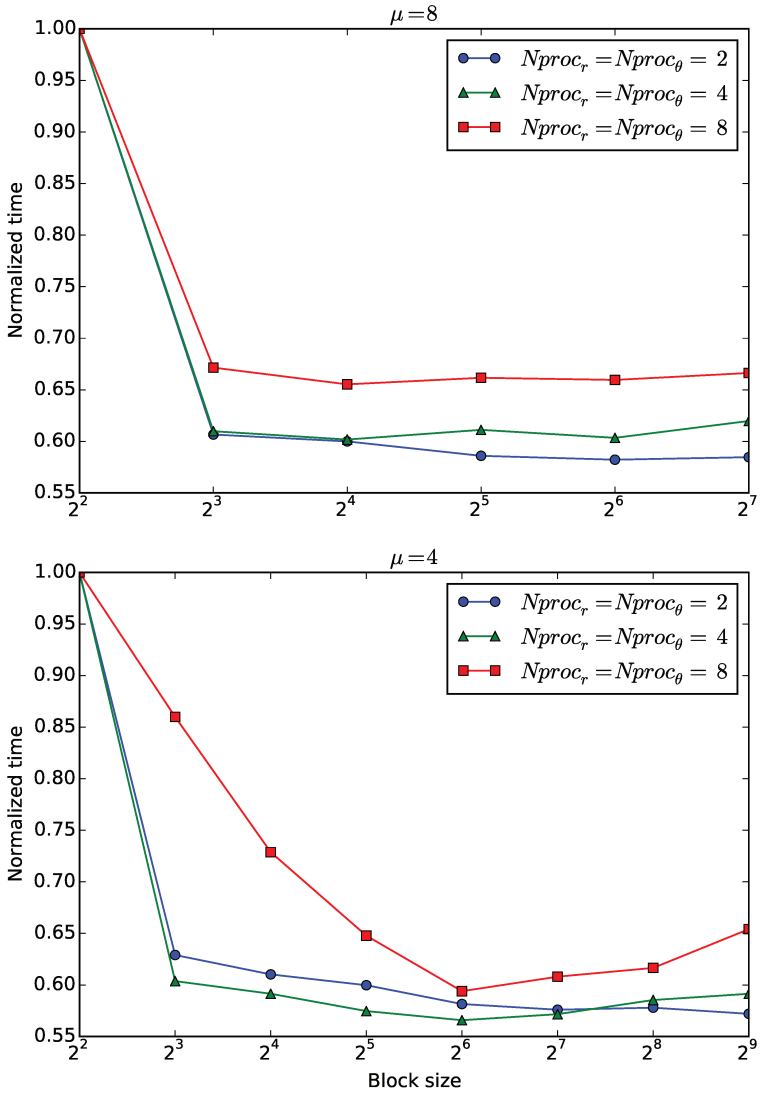


FIGURE 3.5 – Temps d'exécution normalisé du diagnostic moments fluides en fonction de la taille de blocs pour différents rayons de Larmor (c.-à-d. différents μ) et différents nombres de processus sur un maillage de $(512 \times 512 \times 64 \times 31)$ points.

Performance et analyse de la parallélisation par blocs

Suite à ces premières optimisations, des simulations de taille moyenne sont exécutées afin de comparer les performances de la gyromoyenne distribuée parallélisée par blocs et multithreadée et la gyromoyenne transposée dans le diagnostic moments fluides. Nous utilisons pour cela un maillage de dimension $(1024 \times 1024 \times 64 \times 32 \times 1)$ avec une valeur de $\mu = 4.0$ et une taille de bloc égale à 128, toujours sur la machine Poincaré. La Figure 3.6 donne les temps d'exécution du diagnostic moments fluide pour les deux gyromoyennes en fonction du nombre de cœurs. Le nombre de cœurs par processus est égal au nombre de *threads* et vaut 8. Le nombre de processus varie en doublant alternativement le nombre de processus dans les dimensions r et θ. On remarque que la gyromoyenne distribuée est

deux fois plus rapide que la gyromoyenne transposée. Elles perdent néanmoins toutes deux en efficacité avec le nombre de cœurs. Cela est dû à la baisse de la charge de calcul par cœur et à l'augmentation du poids des communications dans le temps d'exécution des deux opérateurs comme expliqué ci-après.

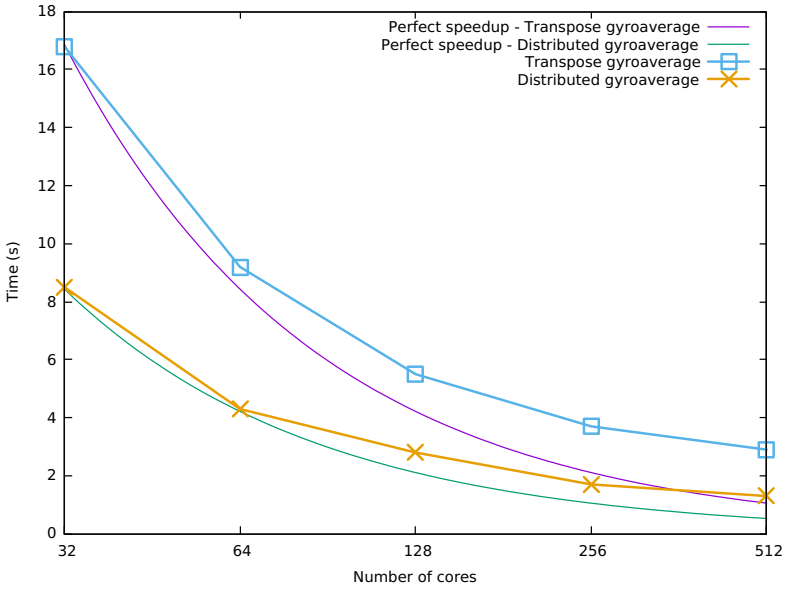


FIGURE 3.6 – Strong scaling relatif et comparaison des speedups théoriques et des temps d'exécution du diagnostic moments fluides utilisant chaque version de la gyromoyenne parallèle pour un maillage de $(1024 \times 1024 \times 64 \times 32 \times 1)$ points.

Afin d'améliorer la parallélisation du diagnostic utilisant la gyromoyenne distribuée, il est utile de savoir où se situe le goulet d'étranglement, c'est-à-dire la partie de l'opérateur s'exécutant le moins rapidement. La Figure 3.7 présente le pourcentage des différents temps moyens d'exécution lors d'un appel à ce diagnostic. On observe encore la prépondérance du temps de communication pour les grandes distributions, mais à moindre importance que pour la gyromoyenne transposée (voir section 2.2.3). On note également que la complexité de la gyromoyenne rejoint celle de l'extraction des résultats (voir 3.2.2). Afin d'améliorer encore les temps d'exécution, on peut ainsi procéder à une optimisation permettant d'absorber les coûts des communications en les effectuant simultanément aux phases de calculs, ce que l'on aborde dans la section suivante.

3.2.2 Recouvrement calcul – communication avec OpenMP

Le recouvrement calcul – communication repose sur l'exécution simultanée de calculs par les processeurs et de communications par les cartes réseau. Pour ce faire, l'existence de primitives de communications non bloquantes et asynchrones facilite l'implémentation. Le caractère non bloquant permet que les processus ne restent pas dans la primitive jusqu'à la terminaison de la communication. Le caractère asynchrone permet que la communication s'étale dans le temps entre son lancement et la vérification de sa terminaison et qu'elle ne s'effectue pas uniquement lors de la vérification. Une étude des schémas de recouvrement calcul – communication peut être trouvée dans [22, 24] ainsi que des suggestions pour les différents paramètres d'optimisation.

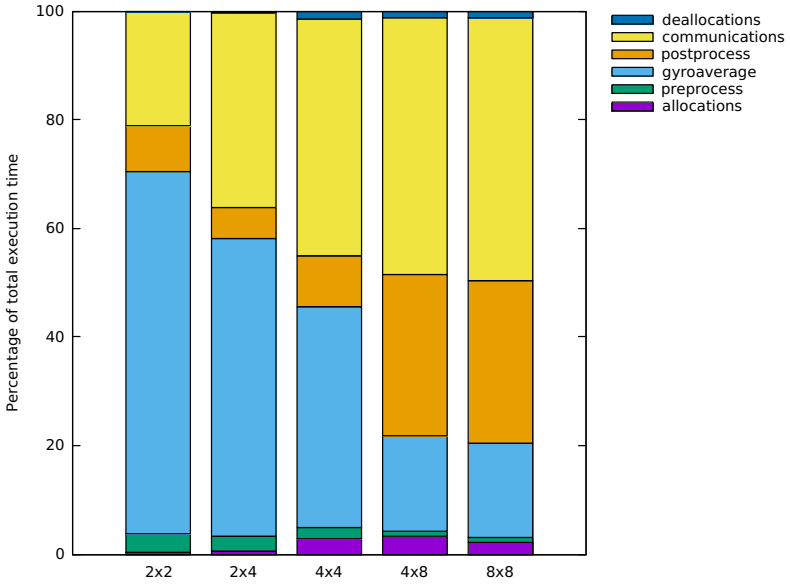


FIGURE 3.7 – Pourcentage du temps moyen d'exécution des différentes parties du diagnostic moments fluides avec gyromoyenne distribuée pour un maillage $512 \times 512 \times 128 \times 64$ sur différentes distributions de processus en $r \times \theta$ avec 8 *threads* par processus MPI.

Gyromoyenne distribuée avec recouvrement

Appliqué à la gyromoyenne distribuée, le recouvrement s'appuiera sur le traitement par blocs explicité auparavant. Le principe est d'entamer les communications des halos du bloc suivant tout en effectuant le calcul de la gyromoyenne pour le bloc actuel. Il serait possible d'effectuer simultanément les communications sur plus de blocs mais nous nous limiterons à deux blocs en parallèle – un en cours de traitement et un en cours d'échange de halos. Le déroulement de l'opérateur avec recouvrement est détaillé par l'algorithme 6. Tout d'abord, deux variables f_{block1} et f_{block2} sont désormais utilisées pour contenir les deux blocs. Il est nécessaire de commencer par lancer l'échange des deux premiers blocs afin de démarrer le pipeline calcul – communication. L'étape 1 permet d'initialiser les buffers et de lancer l'échange de halos (2) pour les blocs 0 à $N_{\text{block}} - 1$ dans les boucles indicées 0 à $N_{\text{block}} - 1$. La phase 3 vérifie la terminaison de l'échange de halo (phase 4), toujours pour les blocs de 0 à $N_{\text{block}} - 1$, mais dans les boucles indicées de 1 à N_{block}. L'étape 5 inverse les références des variables vers les blocs. Ainsi, au tour de boucle 0, le bloc 0 est initialisé dans f_{block1} et son échange de halos débute. Ensuite, les références sont inversées et le bloc 0 est désormais référencé par f_{block2}. Au tour de boucle 1, le bloc 1 est initialisé dans f_{block1} et son échange de halo débute. Puis, la terminaison de l'échange de halo du bloc 0 est vérifiée dans f_{block2} et les traitements sont exécutés. Finalement, les références sont de nouveau inversées et ainsi de suite jusqu'à la dernière boucle où seuls la réception et les traitements du dernier bloc sont effectués.

Étude des capacités du recouvrement

Avec l'algorithme donné en 3.2.2, il est possible d'estimer le gain que de performance que le recouvrement peut apporter. Nous présenterons pour cela une version simplifiée du

Algorithme 6 : Diagnostic avec gyromoyenne distribuée et recouvrement.

Entrées : f, b_s
Sorties : $\mathcal{J}.f$, $\mathcal{M}_{\text{fluid}}$

pour $i_{\text{block}} : 0 \to N_{\text{block}}$ **faire**
1 **si** $i_{\text{block}} \neq N_{\text{block}}$ **alors**
pour $i : 0 \to b_s - 1$ **faire** en parallèle **OpenMP**
$i_\varphi = (i_{\text{block}} \times b_s + i) \mod N_\varphi$
$i_v = (i_{\text{block}} \times b_s + i) \div N_\varphi$
$f_{\text{block1}}(i) \leftarrow$ `preprocess`$(f(r_l : r_u, \theta_l : \theta_u, \varphi_{i_\varphi}, v_{\parallel i_v}, \mu))$
2 `start_send_receive_halos`(f_{block1})
3 **si** $i_{\text{block}} \neq 0$ **alors**
4 `terminate_send_receive_halos`(f_{block2})
pour $i : 0 \to b_s - 1$ **faire** en parallèle **OpenMP**
$i_\varphi = ((i_{\text{block}} - 1) \times b_s + i) \mod N_\varphi$
$i_v = ((i_{\text{block}} - 1) \times b_s + i) \div N_\varphi$
`gyroaverage`$(f_{\text{block2}}(i))$
$\mathcal{M}_{\text{fluid}} \leftarrow$ `postprocess`$(f_{\text{block2}}(i))$
$f(r_l : r_u, \theta_l : \theta_u, \varphi_{i_\varphi}, v_{\parallel i_v}, \mu) \leftarrow f_{\text{block2}}$
5 `swap`$(f_{\text{block1}}, f_{\text{block2}})$

raisonnement détaillé dans [60]. Dans l'étude des temps d'exécution, nous ne considérerons que deux phases : la phase de communication qui inclut l'étape d'initialisation des blocs et l'étape de communication, et la phase de calcul qui inclut l'étape de calcul de la gyromoyenne et l'étape de post-traitement.

Notons $\mathcal{T}_{\text{comm}}$ et $\mathcal{T}_{\text{calc}}$ les temps d'exécution d'un échange de halo et du calcul de la gyromoyenne sur un sous-domaine. Notons également $\alpha = \frac{\mathcal{T}_{\text{comm}}}{\mathcal{T}_{\text{calc}}}$ le ratio entre les deux temps d'exécution et $\mathcal{T}_{\text{overlap}}$ le temps d'exécution total du diagnostic utilisant la gyromoyenne distribuée avec recouvrement. Plusieurs cas de figure pour le déroulement du recouvrement peuvent se présenter en fonction du rapport α tel que le montre la Figure 3.8. Si les temps d'exécution de la phase de communication et de la phase de calcul sont identiques ($\alpha = 1$), alors le recouvrement est parfait ; il n'y a pas de temps d'attente et le temps d'exécution est minimal. Néanmoins, si le rapport se déséquilibre ($\alpha < 1$ et $\alpha > 1$), une partie du matériel est en attente durant le déroulement de certaines sections de l'opérateur et le temps d'exécution n'est plus optimal. À noter que ce temps d'exécution est toujours meilleur que celui de l'opérateur sans recouvrement (dans la figure tous les segments seraient mis bout à bout).

Notons $\mathcal{T}_{\text{distrib}}$ le temps d'exécution du diagnostic utilisant la gyromoyenne avec blocs sans recouvrement. Il est possible d'exprimer les temps total d'exécution grâce aux formules suivantes :

$$\mathcal{T}_{\text{overlap}} = N_{\text{block}} \cdot \texttt{max}(\mathcal{T}_{\text{comm}}, \mathcal{T}_{\text{calc}}) + \texttt{min}(\mathcal{T}_{\text{comm}}, \mathcal{T}_{\text{calc}})$$
$$\mathcal{T}_{\text{distrib}} = N_{\text{block}} \times (\mathcal{T}_{\text{comm}} + \mathcal{T}_{\text{calc}}).$$

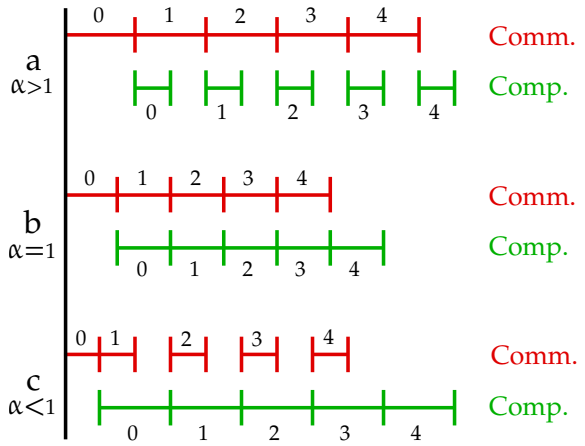


FIGURE 3.8 – Diagramme de Gantt du déroulement de blocs de communications et de calculs pour différents rapports de temps d'exécution α dans le cas d'un recouvrement.

L'accélération offerte par le recouvrement calcul – communication peut alors s'écrire

$$\frac{\mathcal{T}_{\text{distrib}}}{\mathcal{T}_{\text{overlap}}} = \frac{1 + \frac{\mathcal{T}_{\text{comm}}}{\mathcal{T}_{\text{calc}}}}{\frac{\texttt{max}(\mathcal{T}_{\text{comm}},\mathcal{T}_{\text{calc}})}{\mathcal{T}_{\text{calc}}} + \frac{\texttt{min}(\mathcal{T}_{\text{comm}},\mathcal{T}_{\text{calc}})}{\mathcal{T}_{\text{calc}}\, N_{\text{block}}}} = \frac{1+\alpha}{\delta} \tag{3.5}$$

où δ dépend du ratio temps de communication sur temps de calcul et est défini par

$$\delta = \begin{cases} 1 + \frac{\alpha}{N_{\text{block}}} & \text{si } T_{\text{comm}} \leq T_{\text{calc}} \ (\alpha \leq 1) \\ \\ \alpha + \frac{1}{N_{\text{block}}} & \text{si } T_{\text{comm}} > T_{\text{calc}} \ (\alpha > 1). \end{cases}$$

L'équation (3.5) peut être représentée graphiquement en fonction des paramètres α et N_{block} par la Figure 3.9. On observe dans un premier temps que l'accélération qui peut être espérée du recouvrement décroît rapidement avec la divergence des temps de communication et de calcul. On observe également que l'accélération maximale de 2 est obtenue pour un ratio $\alpha = 1$ et que celle-ci tend vers deux avec le nombre de blocs. Cela confirme ce qui pouvait être déduit de la Figure 3.8 et permet d'avoir une estimation grossière du gain apporté par le recouvrement.

Si l'on se réfère à l'équation (1.6) (section 1.2.3), le temps d'exécution de l'échange de halos peut être écrit $\mathcal{T}_{\text{comm}} = \lambda + b_s\, \tau_{\text{init}} + \beta\, b_s\, N_{\mathcal{H}}$ avec b_s le nombre de plans par bloc, τ_{init} le temps d'initialisation d'un bloc et N_{H} la taille d'un halo. Le temps d'exécution du calcul de la gyromoyenne peut également être écrit $\mathcal{T}_{\text{comp}} = \gamma\, b_s + \tau_{\text{extract}}$ avec τ_{extract} le temps d'exécution de l'extraction des moments fluide pour un bloc. La taille de bloc optimale peut alors être déduite analytiquement mais nécessite une connaissance fine des performances de l'algorithme (γ) et de l'architecture du réseau (λ et β). L'impact des effets de localité mémoire est également à prendre en compte. Cette analyse ne sera pas conduite ici, mais le lecteur peut trouver une étude similaire dans [23].

Implémentation du recouvrement

L'implémentation de l'algorithme de recouvrement nécessite l'utilisation de communication MPI pouvant s'effectuer en tâche de fond durant les calculs. Les primitives non

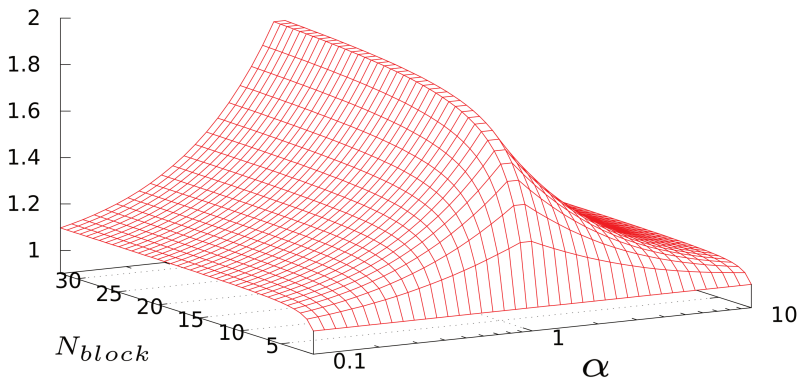


FIGURE 3.9 – Accélération du temps d'exécution pouvant être attendu du schéma de recouvrement.

bloquantes du modèle MPI `MPI_Isend`, `MPI_Irecv` et `MPI_Wait` sont un bon choix pour cet usage. Néanmoins, celles-ci ne sont pas asynchrones dans la plupart des bibliothèques MPI (OpenMPI, IntelMPI ou MVAPich) [38], c'est-à-dire que les communications sont bien lancées de manière non bloquante par les primitives d'envoi et de réception, mais elles ne sont pas réellement effectuées en tâche de fond. Les communications n'avancent que lorsque le programme se situe dans un appel à la bibliothèque MPI ; en pratique, l'ensemble d'une communication s'effectue lors de la primitive d'attente de fin de communication `MPI_Wait`. Ce comportement est supporté par la norme MPI qui n'impose pas le caractère asynchrone aux primitives non bloquantes. Plusieurs implémentations proposent des palliatifs, soit en rajoutant une interface asynchrone entre le code et la bibliothèque MPI [74], soit en implémentant leurs propres primitives [5]. Le choix a été fait de préférer une solution adaptée utilisant le multithreading OpenMP afin de ne pas faire dépendre le code d'une bibliothèque trop spécifique et de garder une solution portable sur toutes les machines utilisées en production (voir section 1.2.1 – «Problématique de calcul parallèle»).

Pour obtenir le comportement asynchrone des communications, celles-ci sont effectuées et gérées par un *thread* dédié, le *thread* maître en l'occurrence. Ainsi, si l'on se réfère à l'algorithme 6, le *thread* communicant est le seul à appeler la primitive de communication de halo (étape 4) tandis que les autres *threads* passent directement au calcul de la gyromoyenne du bloc précédent. De plus, la politique de distribution des itérations de boucles OpenMP entre *threads* est définie comme dynamique, c'est-à-dire que lorsqu'un *thread* termine l'itération qui lui a été attribuée, l'ordonnanceur lui en attribue une autre sans considération de localité ou répartition de charge prédéfinie. De cette façon, aucune itération de boucle n'est pré-attribuée au *thread* communicant et les calculs peuvent se dérouler intégralement en parallèle des communications. Cela permet également au *thread* communicant de rejoindre le groupe de calcul s'il termine ses communications rapidement. La Figure 3.10 présente la répartition des tâches entre les *threads* et le déroulement de leur exécution dans le cas où les communications sont plus courtes que les calculs. Le *thread* le plus à gauche est le *thread* maître (communicant).

La politique d'ordonnancement de boucles dynamique engendre une perte de localité en mémoire de cache puisque les différents traitements des plans ne sont plus liés à un unique *thread*, ce qui était le cas avec un politique statique (l'ensemble des itérations est divisé en

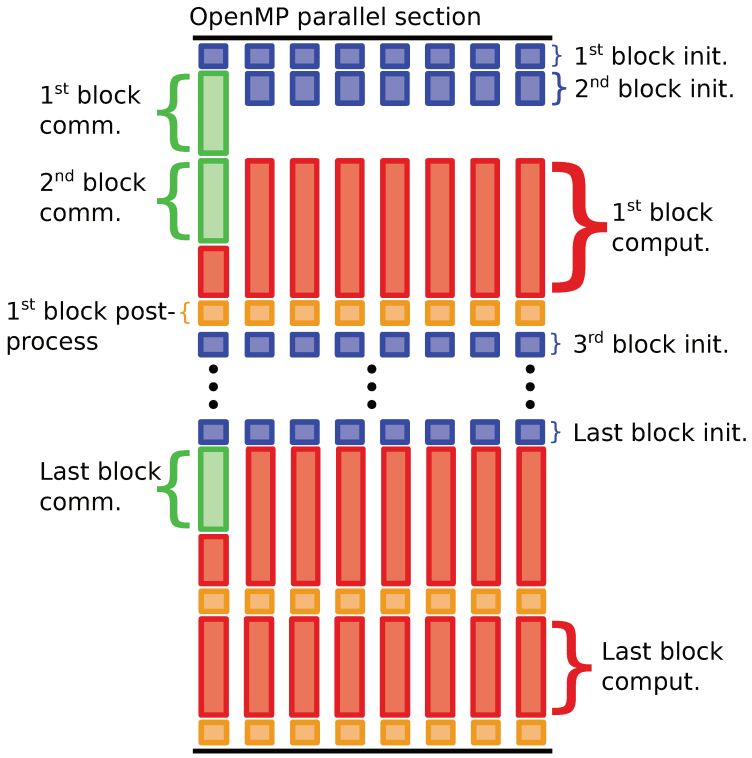


FIGURE 3.10 – Déroulement du diagnostic moments fluides avec recouvrement exécuté par huit *threads* avec $\alpha < 1$ et N_{block} blocs.

autant de blocs que de *threads* et réparti par ordre de numérotation). Cela entraîne une petite baisse de performance de la partie calcul et de post-traitement du diagnostic qui reste acceptable tel que nous le verrons dans la prochaine section.

3.3 Performances de la gyromoyenne distribuée

Cette section présente les performances du diagnostic moments fluides utilisant l'opérateur de gyromoyenne distribuée avec recouvrement. Celles-ci sont également comparées aux performances de la gyromoyenne distribuée avec blocs et de la gyromoyenne transposée (voir section 3.2.1). Pour ce faire, le cas envisagé utilise un maillage de $(1024 \times 1024 \times 64 \times 32 \times 1)$ et une valeur de $\mu = 4$. Il est lancé dans différentes configurations sur la machine Poincaré.

La Table 3.2 présente et compare la scalabilité des deux versions parallèles de la gyromoyenne distribuée (blocs et blocs+recouvrement) pour deux tailles de bloc différentes : une taille de 64 optimale pour cette configuration et une taille de 4 très performante. On remarque tout d'abord que, quelle que soit la taille de bloc, les deux versions passent à l'échelle correctement (entre 70 et 80% d'efficacité relative sur 1024 cœurs). On remarque ensuite que la taille de bloc a beaucoup moins d'impact sur la version avec recouvrement que sur la version par bloc initiale. Cela permet de ne pas se préoccuper de la taille de bloc optimale et ôte par là même un paramètre d'optimisation. De plus, la problématique d'adaptation de la taille de bloc en fonction de la valeur de μ est également résolue puisque la taille de bloc n'a plus que peu d'importance pour les performances (voir section 3.2.1 –

«Impact de la taille des blocs»). Finalement, il n'est plus essentiel d'avoir un nombre de blocs et de *threads* en puissance de deux du fait de la distribution dynamique de charge. Il est néanmoins à noter que les gains apportés par le recouvrement ne sont pas à la hauteur des attentes et n'atteignent que 10% en comparaison de la version par bloc avec une taille de bloc optimale. Le modèle d'estimation de l'accélération apportée par le recouvrement est probablement trop simplifié et ne prend pas en compte de manière assez fine le déroulement de l'opérateur, les caractéristiques du matériel, ou les contentions sur les accès mémoire. Le recouvrement nous permet toutefois de gagner à nouveau en temps d'exécution.

Version et taille de bloc	Nombre de cœurs				
	64	128	256	512	1024
Version blocs (4)	20.67s	10.16s	5.21s	2.83s	1.56s
Version recouvrement (4)	11.55s	5.73s	2.62s	1.39s	0.95s
Version blocs (64)	12.23s	6.27s	3.02s	1.67s	1.01s
Version recouvrement (64)	11.39s	5.66s	2.65s	1.43s	0.98s

TABLE 3.2 – Strong scaling du temps d'exécution de la gyromoyenne distribuée en version bloc et bloc+recouvrement pour deux tailles de bloc différentes (entre parenthèses).

Afin d'évaluer en condition réelle les améliorations apportées à la parallélisation de l'opérateur de gyromoyenne dans le diagnostic moments fluides, des simulations proches des cas de production de GYSELA utilisant plusieurs valeurs de μ sont exécutées. Le maillage utilisé est cette fois de $(1024 \times 1024 \times 128 \times 64 \times 4)$ points et les simulations utilisent 1024 cœurs répartis en 64 processus ($\mathtt{Np}_r = 4, \mathtt{Np}_\theta = 4, \mathtt{Np}_\mu = 4$) utilisant chacun 16 *threads*. La Table 3.3 présente le temps total passé dans l'exécution du diagnostic moments fluide pour chacune des versions parallèles de la gyromoyenne : transposée, distribuée avec blocs et distribuée avec recouvrement. La simulation effectue vingt-cinq pas de temps et le diagnostic est lancé tous les trois pas de temps. Les versions distribuées la gyromoyenne offrent une bonne réduction du temps d'exécution mais qui est dépendante de la valeur de μ (voir section 2.2) contrairement à la version transposée dont la quantité de calcul et le volume de communication restent fixes. Néanmoins, les cas de production n'utilisent pratiquement pas de valeurs de μ supérieures à seize (ou dans des cas très particuliers) et les versions distribuées montrent toujours une amélioration dans ces conditions. Comme vu précédemment, la gyromoyenne distribuée avec recouvrement présente une accélération d'environ 10% par rapport à la version distribuée avec blocs uniquement. Les 24.53 secondes de la gyromoyenne transposée pour $\mu = 0$ sont due aux transpositions qui sont effectuées pour l'extraction des moments par le diagnostic, alors que l'opérateur de gyromoyenne pour $\mu = 0$ est l'identité.

L'amélioration des performances apportée par cette nouvelle implémentation de l'opérateur de gyromoyenne est sensible puisque la part du diagnostic moments fluides dans le temps d'exécution total de la simulation passe 5.7% pour la gyromoyenne transposée à 2.7% pour la gyromoyenne distribuée avec blocs et jusqu'à 2.4% pour la version avec recouvrement. Dans ce cas, le temps d'exécution du diagnostic est réduit d'un facteur deux. Le gain d'empreinte mémoire est également important et essentiel puisque l'appel à ce diagnostic se situe dans l'un des pics mémoire de l'application GYSELA. Pour cette simulation, on passe d'un coût en mémoire pour le diagnostic pour l'ensemble des processus de 256 GiO pour la version transposée (le coût de deux fonctions de distribution) à 5.6 GiO pour la version distribuée avec recouvrement (le coût de deux blocs de soixante-quatre

Version	Valeurs de μ			
	0.	2.6667	5.3333	8.
Version transposée	24.53s	81.74s	80.71s	81.14s
Version blocs	4.785s	36.91s	45.73s	51.75s
Version recouvrement	5.01s	32.66s	38.30s	44.40s

TABLE 3.3 – Temps d'exécution total des neuf appels au diagnostic moments fluides en fonction de la valeur de μ pour chaque version parallèle de la gyromoyenne sur un cas de production utilisant un maillage ($1024 \times 1024 \times 128 \times 64 \times 4$) et 1024 cœurs répartis en 8×8 processus de 16 *threads*.

plans essentiellement). La réduction d'empreinte mémoire est ainsi de l'ordre de 6% sur l'ensemble de cette simulation, ce qui n'est pas négligeable.

Cette version distribuée de la gyromoyenne permet donc une amélioration substantielle des performances en proposant un algorithme adapté à la distribution de données. Le volume des communications est diminué et même si la quantité de calculs et le nombre de communications augmentent, ceux-ci sont intelligemment entrelacés afin d'absorber leur coût. L'empreinte mémoire de l'opérateur est également réduite, permettant potentiellement d'accéder à de plus grosses simulations. Ces améliorations pourraient être accentuées pour les très grands plans poloïdaux (2048×2048 voire 4096×4096) nécessaires à l'introduction des électrons cinétiques. D'autre part, la reconstitution des plans poloïdaux entiers imposée par l'ancienne version de la gyromoyenne constituait un obstacle à la mise en place d'une décomposition de domaine MPI en 5 dimensions plutôt que 3 actuellement, ce qui est rendu possible par cette gyromoyenne distribuée. Cette nouvelle décomposition MPI sera un des objets de la partie II. Néanmoins, l'évolution de la physique simulée par le code remet en cause certaines hypothèses de géométrie du problème. Nous verrons dans le chapitre 4 quelles évolutions et extensions peuvent être envisagées pour encore améliorer l'opérateur : prise en compte d'une nouvelle géométrie, réduction des volumes de communications.

Chapitre 4

Extensions de l'opérateur de gyromoyenne

Sommaire

L'opérateur de gyromoyenne distribuée introduit au chapitre précédent permet de calculer directement la gyromoyenne d'une fonction dont les dimensions r et θ sont distribuées entre les processus (distribution $\mathbf{D}_1$, voir section 1.3.1). Le schéma proposé utilise des halos et pose des conditions sur le rayon de Larmor et le rayon minimal du plan poloïdal afin de simplifier le schéma de communication pour le réduire à des échanges point à point avec les seuls processus voisins. Le code Gysela évolue du point de vue de la physique, dans une direction où ces conditions ne seront plus satisfaites dans de nombreuses configurations. Il est nécessaire de s'intéresser aux problématiques qui en découleront et de proposer une extension de l'opérateur de gyromoyenne. Cette étude sera menée dans la section 4.1. D'autre part, les réacteurs tokamaks adoptent aujourd'hui des géométries plus complexes qu'un simple tore à section circulaire (géométrie D-shape, point X). L'utilisation de maillages faiblement structurés dans le code permet de mieux les modéliser. L'interpolation d'Hermite n'y est pas applicable directement puisqu'elle nécessite les calculs des dérivées

et les points utilisés dans le schéma de différences finies seront peut-être inexistants dans un nouveau maillage. C'est pourquoi une solution basée sur l'interpolation de Lagrange est étudiée dans la section 4.2 qui ne nécessite pas l'estimation des dérivées. Par la suite, un nouveau maillage réduit du plan poloïdal est proposé. Celui-ci permet d'avoir un contrôle plus fin sur la résolution du maillage et fait l'objet d'une étude approfondie dans la section 4.3. Différents opérateurs utilisés dans GYSELA ont été adaptés à ce maillage et leur précision numérique est évaluée sur un prototype 2D.

4.1 Limites du schéma numérique et nouvelles propositions

Un changement dans l'opérateur de Poisson a récemment permis d'abaisser la valeur de la condition de bord radiale $r_{\min}$. Celui-ci atteint désormais des valeurs proches du pas du maillage Δr. Les conditions ne sont alors plus réunies pour que le schéma de communication simplifié de la gyromoyenne distribuée reste valide. Pour les sous-domaines au centre du maillage, il peut alors être nécessaire d'accéder à des données localisées sur des processus distants non voisins, voire sur l'ensemble des processus de la couronne centrale dans le pire des cas (voir section 4.1.1). Plusieurs solutions sont alors envisagées afin de pallier ce problème pour les processus concernés (voir sections 4.1.2 et 4.1.3).

4.1.1 Conditions d'invalidité du schéma simplifié

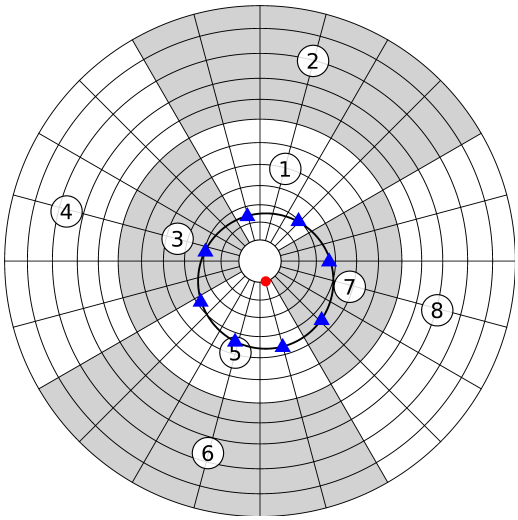
En s'inspirant de l'étude rapportée dans la note [44], un positionnement de $r_{\min}$ à $\frac{\Delta r}{2}$ dans le schéma de différences finies utilisé dans l'opérateur de Poisson a récemment été introduit dans GYSELA. Il permet de diminuer drastiquement les artefacts numériques des termes en $\frac{1}{r}$ et de réduire le rayon central $r_{\min}$ afin d'améliorer la qualité des simulations. Néanmoins, l'opérateur de gyromoyenne distribuée est fortement impacté par cette évolution qui rompt avec les hypothèses posées au début du chapitre 3. Afin de proposer un schéma de communication simple, la condition de bord $r_{\min}$ était considérée large (de l'ordre de $0.1 r_{\max}$ et prépondérant devant le rayon de Larmor $\rho_{\mathcal{L}}$).

Avec $r_{\min}$ de l'ordre de Δr, on a désormais plusieurs configurations possibles. Si le rayon de Larmor est grand, typiquement lorsque $r_{\min} < \rho_{\mathcal{L}}$, les processus de la couronne centrale ont besoin de points situés sur l'ensemble des processus de cette couronne. La Figure 4.1a illustre ce problème. Pour les rayons de Larmor les plus petits, il est encore possible d'utiliser le schéma simplifié. Si le rayon de Larmor se situe entre les deux, les données requises pour la gyromoyenne d'un sous-domaine sont situées à plus d'un sous-domaine de distance dans la direction θ, mais pas sur l'ensemble de la couronne. À noter que la dimension r n'est pas impactée puisque le rayon de Larmor reste inférieur à la hauteur du sous-domaine en r dans les simulations GYSELA ($\rho_{\mathcal{L}} < N_{Lr}\,\Delta r$ tant que $\mathtt{Np}_r <= 16$). Les cercles de Larmor ne peuvent donc pas chevaucher plus de deux sous-domaines dans la direction r. La limite de validité du schéma simplifié en fonction de la configuration peut être calculée avec l'équation (3.2) (section 3.1.3), donnant la taille du halo dans la direction θ, rappelée ci-dessous :

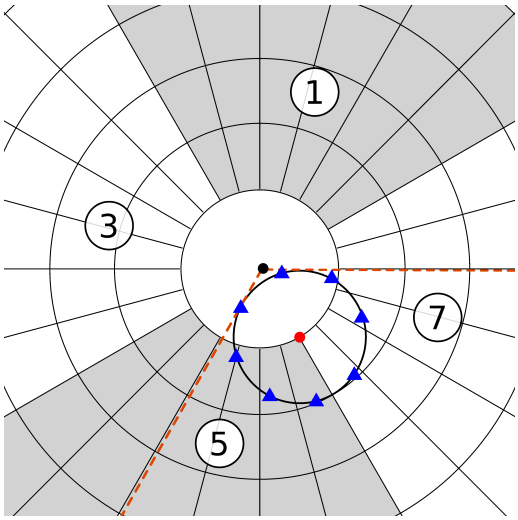
$$N_{\mathrm{H}\theta}(r) = \lceil \frac{1}{\Delta\theta} \arcsin(\frac{\rho_{\mathcal{L}}}{r}) \rceil + \lceil \frac{N_{\mathrm{deriv}}}{2} \rceil.$$

Pour une configuration donnée (N_θ, $\mathtt{Np}_\theta$ et N_{deriv}), choisir un rayon de Larmor tel que $N_{\mathrm{H}\theta} < N_{L\theta} - \lceil \frac{N_{\mathrm{deriv}}}{2} \rceil$ permet de s'assurer de la validité du schéma simplifié (exemple en

Figure 4.1b). De façon générale, on peut connaître le nombre de processus impliqués dans le halo avec l'équation (3.2) si $\rho_{\mathcal{L}} < r_{\min}$ (sinon tous les processus de la couronne sont impliqués).



(a) Un cercle de Larmor du sous-domaine 5 intercepte les sous-domaines de l'ensemble de la couronne (processus 1, 3 et 7).



(b) Limite de validité du schéma de communication simplifié pour un *stencil* de dérivée à 4 points ($N_{\mathrm{L}\theta} = 6$, $N_{\mathrm{H}\theta} = 4$).

FIGURE 4.1

La stratégie utilisée pour résoudre ce problème consiste donc à établir un schéma spécifique pour la couronne centrale. En effet, la deuxième couronne de processus ne peut être impactée dans les configurations actuelles. La taille maximale du rayon de Larmor est choisie pour ne pas dépasser 64 pas de maillage en r ce qui, dans les configurations de production, représente dans le pire des cas un quart de la hauteur du sous-domaine ($\rho_{\mathcal{L}} = \frac{1}{4} N_{Lr} \Delta r$). La taille du halo en θ vaut alors

$$N_{\mathrm{H}\theta}(\rho_{\mathcal{L}}) = \lceil \frac{\arcsin(\frac{1}{4})}{\Delta\theta} \rceil + \lceil \frac{N_{\mathrm{deriv}}}{2} \rceil \simeq \frac{N_\theta}{24} + \lceil \frac{N_{\mathrm{deriv}}}{2} \rceil.$$

Cela signifie qu'à moins qu'il y ait 24 processus en θ ou plus, le schéma simplifié reste applicable à la seconde couronne de processus. Nous nous limiterons donc ici à l'étude d'un schéma spécifique pour la couronne centrale. À noter que les maillages nécessaires aux électrons cinétiques qui seront à terme intégrés dans GYSELA augmentent fortement le nombre de mailles interceptées par le rayon de Larmor, ce qui peut rendre le schéma simplifié *caduc* pour la seconde couronne également. Il serait alors nécessaire de revoir entièrement le calcul de la gyromoyenne, le schéma central étant trop coûteux et dépendant de la taille des données à traiter comme nous le verrons par la suite. Il faut donc établir un nouvel algorithme pour la couronne centrale pour laquelle le schéma de communication simplifié ne peut plus être utilisé. Plusieurs solutions sont envisageables.

4.1.2 Schéma de communication global pour la couronne centrale

Ce premier schéma vise à apporter une solution rapide et simple à mettre en œuvre au problème des halos étendus à plusieurs processus dans la direction θ. Cette approche

propose une solution unique, quel que soit le nombre de processus impliqués dans les halos. Le principe est de faire communiquer tous les processus au centre afin qu'ils s'échangent l'ensemble de leurs données, étendant le halo de chaque processus à tous les sous-domaines au centre (voir Figure 4.2). Ainsi, à la fin de la phase de communication, tous les processus du centre auront à leur disposition localement l'ensemble des données de la couronne. Ils peuvent donc effectuer la gyromoyenne de leurs points respectifs comme vu dans les chapitres précédents.

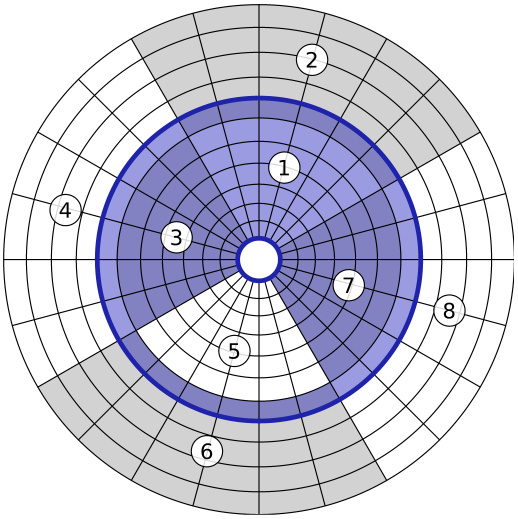


FIGURE 4.2 – Exemple de halo pour le processus 5 de la couronne centrale avec schéma de communication global.

Similairement au schéma de communication simplifié (voir section 3.1.1), l'échange de halos s'effectue ici en deux étapes. Une première étape durant laquelle chaque processus de la couronne échange son halo avec le processus situé au-dessus dans la direction r et une seconde étape durant laquelle tous les processus de la couronne partagent leur sous-domaine dans une communication collective. Cette communication est effectuée par un appel à `MPI_Allgather`. Nous appellerons la gyromoyenne utilisant ce schéma spécifique pour les processus de la couronne central «gyromoyenne *globale*». Cette solution est implémentée dans le diagnostic sur les moments fluides.

Les premiers résultats de performance montrent des temps d'exécution du diagnostic bien supérieurs aux temps de la version initiale utilisant la gyromoyenne transposée. La Figure 4.3 représente ces résultats sur une configuration de $(512 \times 512 \times 64 \times 32 \times 1)$ points pour un nombre variable de cœurs de calcul. Chaque processus dispose de 4 *threads* et μ vaut 4. La valeur de $r_{\min}$ utilisée pour les versions de gyromoyenne transposée et distribuée est de 0.1 afin de ne pas avoir le problème au centre dans la version distribuée ; elle est de 0.01 pour la gyromoyenne globale afin d'utiliser le schéma global sur la couronne centrale. La valeur de r_{min} n'influe pas sur les temps de calcul (voir section 2.2.2), mais impacte la taille du halo et donc les coûts de communication dans la version distribuée (voir section 3.1.3). Néanmoins, le schéma de communication de la couronne impactée est remplacé par le schéma global ; les effets sur les performances dus à la valeur de $r_{\min}$ sont donc négligeables. On remarque que les temps d'exécution de la gyromoyenne globale évoluent par paliers : un premier palier à 32 et 64 cœurs ($(\mathtt{Np}_r, \mathtt{Np}_\theta)$ valent respectivement $(2, 2)$ et $(2, 4)$), un second à 128 et 256 cœurs ($(4, 4)$ et $(4, 8)$) et un dernier à 512 cœurs ($(8, 8)$). On

observe donc une forte dépendance au nombre de processus en r qui découle de plusieurs caractéristiques de ce schéma que nous verrons par la suite.

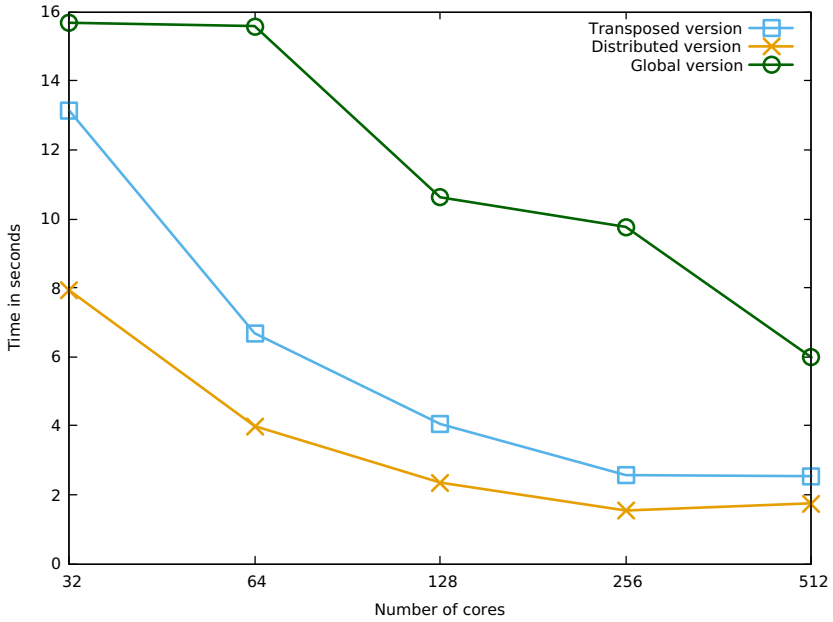


FIGURE 4.3 – Strong scaling relatif et comparaison des temps d'une exécution du diagnostic moments fluides utilisant chaque version de la gyromoyenne parallèle pour un maillage de $(512 \times 512 \times 64 \times 32 \times 1)$ points. La version distribuée utilise $r_{\min} = 0.1$ et les autres $r_{\min} = 0.01$.

Dans cette version globale, plusieurs facteurs viennent dégrader les performances :

— le coût des communications dont seule la seconde partie change (mise à jour dans la direction θ). Celle-ci passe, pour chaque processus, d'un échange de halo de taille $N_{\mathrm{H}\theta}(N_{Lr} + 2N_{\mathrm{H}r})$ avec deux voisins en θ à un échange de sous-domaines de taille $N_\theta(N_{Lr} + 2N_{\mathrm{H}r})$ entre $\mathtt{Np}_\theta$ processus ;

— le coût des différentes conditions dans les boucles internes nécessaires à cause de la périodicité du domaine en θ (calculs de *modulos* notamment). En effet, les coordonnées des points de Larmor n'étant calculées que pour les points $(r_i, \theta{=}0)$, il est nécessaire de ramener l'indice θ des points de Larmor entre 0 et $N_\theta - 1$ lorsqu'on y ajoute l'indice θ_j du point courant. Ce n'était pas le cas pour l'algorithme distribué puisque l'étendue en θ du domaine était toujours inférieure à π (pour $\mathtt{Np}_\theta \geq 2$) ;

— le coût du calcul du second membre (dérivées et dérivée seconde), qui passe de $N_{Lr}\,N_{L\theta}$ à $N_{Lr}\,N_\theta$ (multiplication par $\mathtt{Np}_\theta$) pour chaque processus ;

— l'empreinte mémoire qui augmente largement (multiplication par $\mathtt{Np}_\theta$) pour ces processus ;

— la perte de localité, et donc des bénéfices des effets de caches, liée à l'augmentation de la taille des données à traiter pour chaque processus.

Afin de le vérifier, la Table 4.1 détaille les temps d'exécution présentés en Figure 4.3 pour la version distribuée. Trois temps sont donnés : le temps de communication (Comm.), le temps de calcul des dérivées (RHS) et le temps de calcul de la gyromoyenne (Gyroavg.). À noter que la somme des trois n'est pas égale au temps d'exécution totale du fait du recouvrement et des temps d'initialisation et de post-traitement non représentés. Les Tables 4.2a et 4.2b

présentent les temps d'exécution de la version globale respectivement pour les processus de la couronne centrale (utilisant les communications globales) et les autres processus (utilisant le schéma simplifié). On observe que les temps d'exécution du diagnostic avec la gyromoyenne globale sont largement dominés par les processus de la couronne centrale.

Pour ces processus intérieurs, les temps de communication sont prépondérants. Les temps de communication et de calcul des dérivées sont proportionnels à $\mathtt{Np}_r$. En effet, le volume de données à traiter dans ces deux étapes correspond à l'ensemble des données de la couronne dont la taille dépend de $\mathtt{Np}_r$. Ainsi pour $\mathtt{Np}_r = 2$, elles représentent 50% du plan, pour $\mathtt{Np}_r = 4$, 25%, etc. On remarque également une légère augmentation du temps de communication entre les configurations (2,2) et (2,4) et entre les configurations (4,4) et (4,8) bien que le volume de données ne change pas. Cela est dû à l'augmentation du nombre de communications, proportionnel au nombre de processus en θ. Les temps de communication sont donc plus fortement impactés par le volume que par le nombre de communications. Concernant les temps du calcul de la gyromoyenne, ceux-ci sont fortement impactés par la périodicité du domaine qui nécessite l'utilisation de modulo dans la boucle la plus interne du calcul.

Pour les $(\mathtt{Np}_r - 1)\,\mathtt{Np}_\theta$ processus des couronnes extérieures dans l'algorithme global, on remarque que l'ensemble des temps d'exécution suivent la même tendance que ceux des $\mathtt{Np}_r\,\mathtt{Np}_\theta$ processus pour la gyromoyenne distribuée, tout en étant légèrement inférieurs. En effet, dans la version globale, la taille de halo en θ est calculée sans prendre en compte les processus intérieurs qui ont leur propre schéma. $N_{\mathrm{H}\theta}$ est donc légèrement inférieur qu'en version distribuée et les temps d'exécution sont plus courts.

$(\mathtt{Np}_r, \mathtt{Np}_\theta)$	Comm.	RHS	Gyroavg.
(2,2)	2.28	0.77	1.57
(2,4)	1.57	0.38	0.71
(4,4)	1.07	0.20	0.37
(4,8)	0.90	0.12	0.20
(8,8)	0.82	0.08	0.14

TABLE 4.1 – Détails des temps d'exécution moyens du diagnostic en version distribuée (en secondes) pour un maillage de $(512 \times 512 \times 64 \times 32 \times 1)$ points.

$(\mathtt{Np}_r, \mathtt{Np}_\theta)$	Comm.	RHS	Gyroavg.
(2,2)	9.38	1.98	2.99
(2,4)	9.97	2.07	1.85
(4,4)	4.36	0.96	0.75
(4,8)	4.50	0.89	0.51
(8,8)	2.18	0.46	0.26

(a) Temps d'exécution pour les processus de la couronne centrale (en secondes).

$(\mathtt{Np}_r, \mathtt{Np}_\theta)$	Comm.	RHS	Gyroavg.
(2,2)	2.34	0.61	1.28
(2,4)	1.44	0.26	0.59
(4,4)	0.96	0.13	0.21
(4,8)	0.80	0.07	0.12
(8,8)	0.74	0.04	0.05

(b) Temps d'exécution pour tous les autres processus (en secondes).

TABLE 4.2 – Détails des temps d'exécution moyens du diagnostic en version globale pour un maillage de $(512 \times 512 \times 64 \times 32 \times 1)$ points.

Cet algorithme de gyromoyenne utilisant un schéma de communication global pour les processus centraux n'est pas satisfaisant. Ses performances, tant en termes de temps d'exé-

cution que de volume de communication, sont en deçà de celles de la version transposée. Des améliorations peuvent être apportées au prix d'un investissement en développement important. Les surcoûts dus aux communications et au calcul des dérivées dans la couronne intérieure sont compressibles en réduisant l'ensemble des points échangés et traités aux seuls points réellement nécessaires (et non pas l'ensemble de la couronne). Cela nécessiterait néanmoins des schémas de communication et une gestion des indices complexes et non réguliers. Il est également possible d'envisager d'autres solutions comme nous le verrons dans la section suivante.

4.1.3 Redistribution locale des données de la couronne centrale

Ce second schéma apporte une alternative au schéma avec communication globale en proposant un algorithme hybride entre les versions transposées et distribuées. Celui-ci sera dénommé «gyromoyenne avec *redistribution locale*». Le principe est de s'inspirer de la version initiale par transposition et de redistribuer les données uniquement pour les processus au centre du plan poloïdal. On aurait alors la couronne centrale de processus dans une distribution

$$\mathbf{D}_2' = (0 : N_{Lr} - 1,\ *,\ iN_{L\varphi} : (i+1)N_{L\varphi},\ jN_{Lv_\parallel} : (j+1)N_{Lv_\parallel},\ \mu) \tag{4.1}$$

et le reste des processus en distribution $\mathbf{D}_1$, et ce pour la durée du calcul de la gyromoyenne de chaque bloc en cours de traitement (voir Figure 4.4). Ainsi, à la fin de la phase de communication, chaque processus du centre aura localement à sa disposition l'ensemble des données de la couronne sur $\frac{b_s}{\mathtt{Np}_r\mathtt{Np}_\theta}$ plans (en supposant b_s au moins supérieur à $\mathtt{Np}_r\mathtt{Np}_\theta$, quitte à procéder à un ajustement de la répartition sur la division n'est pas exacte). Il est toutefois nécessaire de retourner en distribution $\mathbf{D}_1$ pour la suite de la simulation, ce qui génère une deuxième phase de communication.

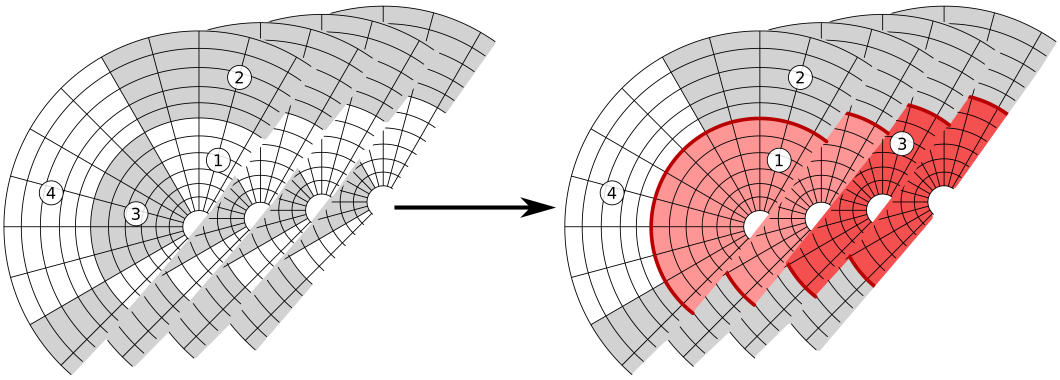


FIGURE 4.4 – Exemple de changement de distribution pour les processus au centre avec un bloc de quatre plans poloïdaux. La partie en blanc et gris est en distribution $\mathbf{D}_1$ et la partie centrale en nuance de rouge en distribution $\mathbf{D}_2'$.

Les données étant simplement redistribuées, la quantité de calculs pour les processus centraux reste la même que dans la gyromoyenne distribuée. Le surcoût engendré par les conditions dues à la périodicité du domaine en θ est néanmoins inévitable. Concernant les communications pour un sous-domaine, le volume est de $b_s \cdot N_{\mathrm{L}r}N_{\mathrm{L}\theta}$ par bloc fractionné en $\mathtt{Np}_\theta - 1$ échanges point à point pour une transposition. Il faut donc en compter deux

pour l'aller-retour. Le volume des échanges reste tout de même moindre qu'en version globale et la multiplication de leur nombre par $\mathtt{Np}_\theta - 1$ est compensée par le fait que ce sont des communications point à point et non des communications globales. La seconde phase de communication nécessaire au retour dans la distribution de données initiale pose toutefois problème en ajoutant un point de synchronisation supplémentaire, potentiellement néfaste aux temps d'exécution pour l'algorithme avec recouvrement. En tout, le volume des communications devrait être fortement réduit par rapport à la version globale. L'empreinte mémoire est également moindre puisque le surcoût est celui d'une duplication lors de chaque transposition des deux blocs en cours de traitement dans le pipeline de recouvrement (contre un ajout de $\mathtt{Np}_\theta - 1$ blocs dans la version globale). Ce schéma devrait théoriquement présenter des performances situées entre celles de la gyromoyenne distribuée sans schéma au centre et celles de la gyromoyenne transposée.

Du fait de l'introduction d'une deuxième phase de communication, il est nécessaire de revoir le schéma de recouvrement. Un exemple de déroulement de l'exécution des *threads* dans ce schéma est donné en Figure 4.5. Les proportions utilisées sont représentatives des rapports de temps d'exécution en gyromoyenne distribuée dans des configurations à grand plan poloïdal (512×512 et 1024×1024, voir en fin de section 3.2.1). Similairement à l'algorithme de recouvrement de la gyromoyenne distribuée (voir section 3.2.2), le pipeline est démarré avec l'initialisation des blocs 1 et 2, la communication des halos et la transposition ($\mathbf{D}_1 \rightarrow \mathbf{D}'_2$) du bloc 1. La première phase de communication du bloc 2 est effectuée en parallèle du calcul de la gyromoyenne du bloc 1. On retourne en distribution $\mathbf{D}_1$ sur le bloc 1 avant d'exécuter son post-traitement afin d'avoir les données des moments fluides dans la bonne distribution. De même pour le bloc 2, alors que l'initialisation du bloc 3 qui a été effectuée en parallèle de la communication précédente permet de lancer la première phase de communication du bloc 3, et ainsi de suite. Avec un ratio calcul – communication tel que présenté en Figure 4.5, on constate que la chaîne de communications devient le chemin critique de ce schéma de l'opérateur de gyromoyenne.

À noter qu'il est possible d'effectuer le post-traitement en distribution $\mathbf{D}'_2$. Cela implique de changer l'algorithme de recomposition des moments fluides à la fin du diagnostic et l'influence sur les performances serait quasiment nulle. En effet, le nouveau schéma de recouvrement ne s'agence pas beaucoup mieux à cause de la prépondérance des temps de communication devant ceux du calcul de la gyromoyenne et des post-traitements. D'autre part, il devient complexe d'agencer les différentes étapes de l'algorithme de recouvrement. Les chemins critiques peuvent changer notoirement en fonction des temps d'exécution de chaque partie. Le modèle de programmation par tâche présenté en introduction (section 1.2.4) pourrait faciliter le travail des développeurs et permettre d'avoir un bon ordonnancement, quelle que soit la répartition des temps d'exécution, et donc potentiellement de meilleures performances.

Ce dernier schéma pourrait permettre d'atteindre de bonnes performances bien que l'ajout d'une deuxième phase de communication dans le schéma de recouvrement risque de diminuer les gains potentiels. Il n'a pas fait l'objet d'une implémentation du fait de sa complexité de mise en œuvre et de l'incertitude quant au gain de temps d'exécution qu'il pourrait apporter. D'autre part, une autre solution plus en adéquation avec les évolutions du code lui a été préférée comme nous allons le voir.

L'utilisation d'un schéma spécifique au centre du plan est nécessaire et entraîne une hausse du temps d'exécution inévitable. Différents schémas sont envisageables dont la complexité de mise en œuvre est significative. Il est néanmoins possible de minimiser leur

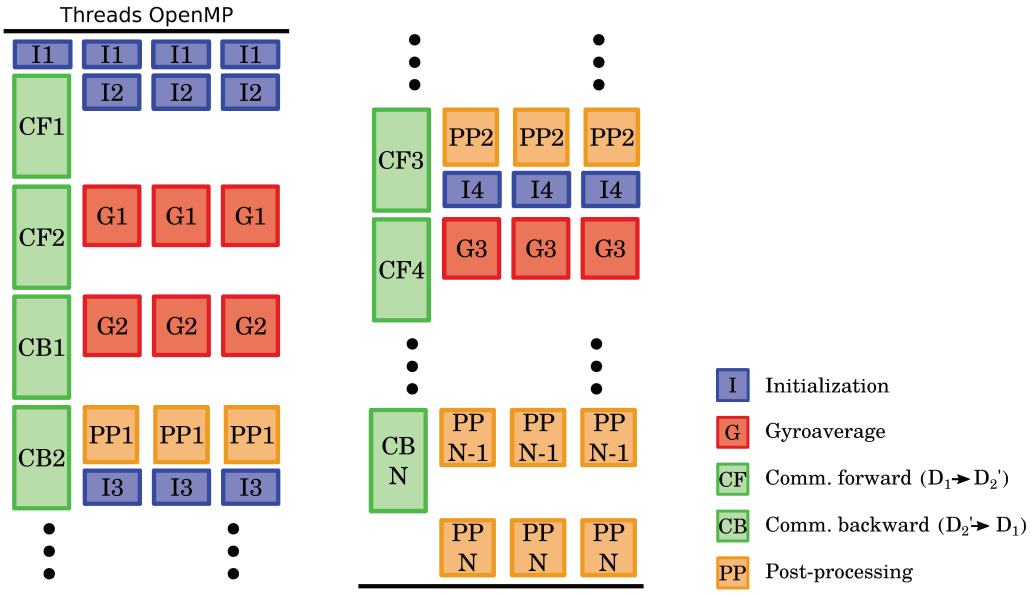


FIGURE 4.5 – Exemple de déroulement attendu du diagnostic pour un processus de la couronne centrale avec le schéma avec redistribution locale. Les étapes sont suffixées par le numéro du bloc traité.

surcoût. Le principe est de diminuer le nombre de points de maillage utilisés pour la gyromoyenne des processus de la couronne centrale. Par exemple, ne considérer qu'un point sur deux dans la direction r ou θ. Cette approche a été retenue et fait l'objet d'une étude complète en section 4.3. Elle permet, sous certaines conditions, de réduire les temps d'exécution sans perte de précision.

4.2 Comparaison des interpolateurs Hermite et Lagrange

En parallèle aux travaux d'extension de l'opérateur de gyromoyenne à des maillages à $r_{\min}$ faible, une étude de comparaison des interpolateurs Hermite et Lagrange a été menée. Il est en effet prévu de faire évoluer GYSELA vers des maillages faiblement structurés dans le plan poloïdal afin de pouvoir simuler des géométries de tores plus complexes. Ce type de maillage permettrait également de réduire grandement la taille de la simulation (90%) dans des cas particuliers (simulation raffinée du bord extérieur) où la connaissance de l'ensemble du tore n'est pas nécessaire. D'autre part, l'interpolation de Lagrange est plus adaptée aux maillages faiblement structurés, car elle est plus directe et ne nécessite pas l'estimation des dérivées contrairement à l'interpolation d'Hermite. Cette section compare les performances et la précision (section 4.2.2) de l'interpolation de Lagrange par rapport à l'interpolation d'Hermite dans le cadre de la gyromoyenne distribuée.

Dans l'implémentation de la gyromoyenne distribuée, l'interpolation de Lagrange vient se substituer à l'interpolation d'Hermite en conservant le même algorithme global (voir section 2.2). Ainsi, pour chaque point de Larmor, les coefficients des points intervenant dans son interpolation sont ajoutés à la matrice M_{coef}. L'étape du calcul de la gyromoyenne (combinaison des coefficients et des valeurs) reste la même. Le seul changement apporté porte sur le calcul des coefficients d'interpolation et sur la suppression du calcul des dérivées.

4.2.1 Polynômes d'interpolation de Lagrange

Les polynômes d'interpolation de Lagrange permettent d'approcher une fonction f définie sur un maillage discretisé afin d'estimer sa valeur entre les points du maillage. Cette reconstruction assure que les valeurs de la fonction initiale et des polynômes d'interpolation sont égales aux points du maillage. Considérons un cas 1D avec une discrétisation régulière en N points $(x_i)_{i\in[\![0,N-1]\!]}$ tel que $\forall i \in [\![0, N-1]\!]$, $x_i = x_0 + i \cdot \Delta x$, avec Δx et x_0 donnés. Le polynôme de Lagrange $L(X)$ est un polynôme de degré $N_L - 1$ où N_L est le nombre de points du maillage pour lesquels on assure que le polynôme aura la même valeur que la fonction interpolée. Le polynôme s'écrit

$$L(X) = \sum_{i=0}^{N_L-1} L_i(X) f(x_i) \tag{4.2}$$

où les polynômes de base de Lagrange L_i sont donnés par

$$L_i(X) = \prod_{j=0, j\neq i}^{N_L-1} \frac{X - x_j}{x_i - x_j} \qquad \text{d'où } L_i(x_j) = \delta_{i,j}.$$

Comme indiqué en section 2.1.2, les polynômes d'interpolation de Lagrange sont sujets au phénomène de Runge. Les interpolations sont donc réalisées localement autour de chaque point et non sur l'ensemble du domaine. Dans le cas d'un maillage 2D, l'interpolation de Lagrange combine les deux dimensions. Elle peut être considérée comme le produit tensoriel de deux interpolations 1D. Ainsi, dans le plan $r \times \theta$, une interpolation 4 points (de degré 3) est une combinaison linéaire des valeurs des seize points du maillage les plus proches du point à interpoler (voir Figure 4.6).

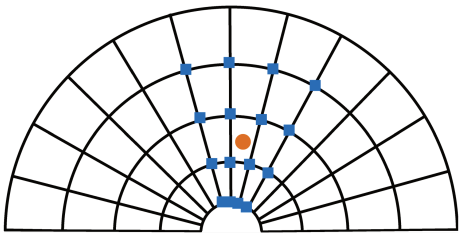


FIGURE 4.6 – Exemple d'une interpolation de Lagrange avec seize points sur un maillage polaire (produit tensoriel de deux polynômes de Lagrange de degré 3).

Par conséquent, pour un point (r, θ) donné tel que $r_h \leq r < r_{h+1}$ et $\theta_k \leq \theta < \theta_{k+1}$ où $h \in [\![0, N_r - 1]\!]$ et $k \in [\![0, N_\theta - 1]\!]$, et pour un *stencil* de p points, la valeur interpolée de f s'écrit

$$L^{(p)}(r, \theta) = \sum_{i=h+l}^{h+u} \sum_{j=k+l}^{k+u} f(r_i, \theta_j) M_i^{(p)}(r) N_j^{(p)}(\theta) \tag{4.3}$$

où $l = -\lfloor \frac{p-1}{2} \rfloor$, $u = \lfloor \frac{p}{2} \rfloor$ et les $M_i^{(p)}$ et $N_j^{(p)}$ sont les polynômes de base de Lagrange pour les dimensions r et θ restreints aux p points autour des coordonnées r_i et θ_j. Ils se formulent

$$M_i^{(p)}(r) = \prod_{m=h+l, m\neq i}^{h+u} \frac{r - r_m}{r_i - r_m} \qquad \text{et} \qquad N_j^{(p)}(\theta) = \prod_{n=k+l, n\neq j}^{k+u} \frac{\theta - \theta_n}{\theta_j - \theta_n}.$$

De même que pour l'interpolation d'Hermite, lorsqu'un indice r sort de l'intervalle $[r_{\min}, r_{\max}]$, une projection radiale sur $r_{\min}$ ou $r_{\max}$ est effectuée afin d'appliquer une condition de Dirichlet.

Erreur d'interpolation

Considérons un intervalle $I = [x_0, x_{N-1}]$ discrétisé par un maillage $(x_i)_{i\in[\![0,N-1]\!]}$, une fonction $f \in \mathcal{C}^p$ définie sur l'intervalle et un polynôme d'interpolation de degré $p-1$ (avec $p \leq N$), alors quel que soit x dans l'intervalle, il est possible d'exprimer l'erreur commise lors de l'interpolation [3] par applications itérées du théorème de Rolle. Celle-ci s'exprime :

$$\exists \xi \in I,\ f(x) - L^{(p)}(x) = \frac{f^{(p)}(\xi)}{p!} \prod_{j=0}^{p-1} (x - x_j). \tag{4.4}$$

Dans le cas d'un maillage régulier de pas Δx, la relation (4.4) peut être bornée par une expression plus simple [65] :

$$\|f(x) - L^{(p)}(x)\| \leq \frac{\Delta x^p}{4p} \|f^{(p)}\|_\infty \tag{4.5}$$

Cette relation nous permettra de vérifier l'implémentation de l'interpolation de Lagrange en comparant les pentes des courbes expérimentales de convergence en espace et en degré avec cette borne.

4.2.2 Comparaison des versions Hermite et Lagrange

Afin de pouvoir utiliser l'interpolation de Lagrange, il faut que celle-ci puisse atteindre la même précision que l'interpolation d'Hermite sur un ensemble de problèmes représentatifs des utilisations. Il faut également que sa vitesse d'exécution reste similaire à celle de l'interpolation d'Hermite, voire qu'elle soit inférieure.

Comparaison numérique

Les fonctions de type Fourier-Bessel sont une classe de fonctions dont la gyromoyenne est connue analytiquement [70]. Il est donc aisé de réaliser des vérifications numériques en s'appuyant sur cette famille de fonctions. Celles-ci s'expriment sous la forme

$$f(r, \theta) = C_m(zr)e^{im\theta} \tag{4.6}$$

où C_m est une fonction de Bessel de première ou seconde espèce, $z \in \mathcal{C}$ et $m \in \mathcal{N}$. Alors, pour $\rho_{\mathcal{L}}$ donné, la gyromoyenne de f vaut

$$\mathcal{J}_{\rho_{\mathcal{L}}}(f)(r, \theta) = J_0(z\rho_{\mathcal{L}})f(r, \theta) \tag{4.7}$$

avec J_0 la fonction de Bessel de première espèce d'ordre 0. Pour la comparaison de la précision numérique des deux gyromoyennes, la fonction utilisée est la suivante :

$$f(r, \theta) = J_1(j_{1,1}\frac{r}{r_{\max}})e^{i\theta} \tag{4.8}$$

où $j_{1,1}$ est le premier zéro positif de la fonction J_1, fonction de Bessel de première espèce d'ordre 1. L'argument est divisé par $r_{\max}$ afin que la fonction s'annule en $r_{\max}$. Elle est représentée en Figure 4.7.

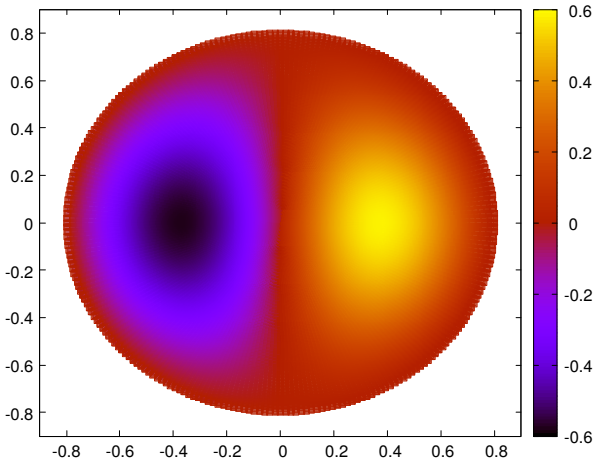


FIGURE 4.7 – Carte couleur représentant la fonction de Bessel de première espèce donnée par l'équation (4.8).

Comme mentionné en section 4.2.1, l'implémentation de la gyromoyenne avec l'interpolation de Lagrange permet de choisir le nombre de points du *stencil* utilisé dans l'interpolation. La précision de l'opérateur est ainsi ajustable. Le Figure 4.8 donne l'erreur moyenne effectuée lors de la gyromoyenne de cette fonction sur l'ensemble d'un plan poloïdal pour différents interpolateurs (quatre Lagrange et un Hermite) en fonction de la discrétisation du maillage. Les courbes de convergence théoriques telles qu'exprimées par la relation 4.5 sont également affichées pour vérification. L'erreur maximale (norme L_∞) étant très proche de l'erreur moyenne (norme L_1) hors des conditions aux bords, seule cette dernière est représentée. Une seule courbe est montrée pour l'interpolateur d'Hermite car l'implémentation utilisée n'est pas paramétrable. L'interpolation d'Hermite utilise 2 points et un *stencil* de 4 points pour le calcul des dérivées. La discrétisation du maillage augmente en doublant alternativement la résolution dans les dimensions r et θ (θ en premier) à partir de la configuration (128×128). On remarque pour l'ensemble des courbes qu'il n'y a pas d'évolution significative de la précision lors de l'augmentation de la résolution en r (des plateaux sont visibles sur les courbes). Cela signifie que la dimension r est sur-résolue pour cette fonction test.

Les courbes d'erreur des gyromoyennes avec les interpolateurs de Lagrange suivent assez fidèlement le taux de convergence théorique $c_p \cdot \Delta\theta^p$ avec c_p constant pour une taille de stencil p donnée. L'*offset* des courbes théorique n'est pas représentatif et est choisi pour être proche des courbes de leur Lagrange respectif entre 2^{19} et 2^{20} point afin de faciliter la lisibilité. La courbe d'erreur de la gyromoyenne avec l'interpolateur d'Hermite a une pente équivalente à celle d'un Lagrange 3 points. Bien qu'un *stencil* de trois points pour l'interpolation de Lagrange puisse ici suffire à égaler, voire surpasser l'interpolation d'Hermite, il est préférable de choisir un nombre de points pair et un *stencil* centré afin d'éviter les problèmes d'instabilité de l'interpolation [59].

Comparaison des complexités

La gyromoyenne basée sur l'interpolation de Lagrange met en œuvre un algorithme similaire à la gyromoyenne basée sur l'interpolation d'Hermite sans la coûteuse étape de

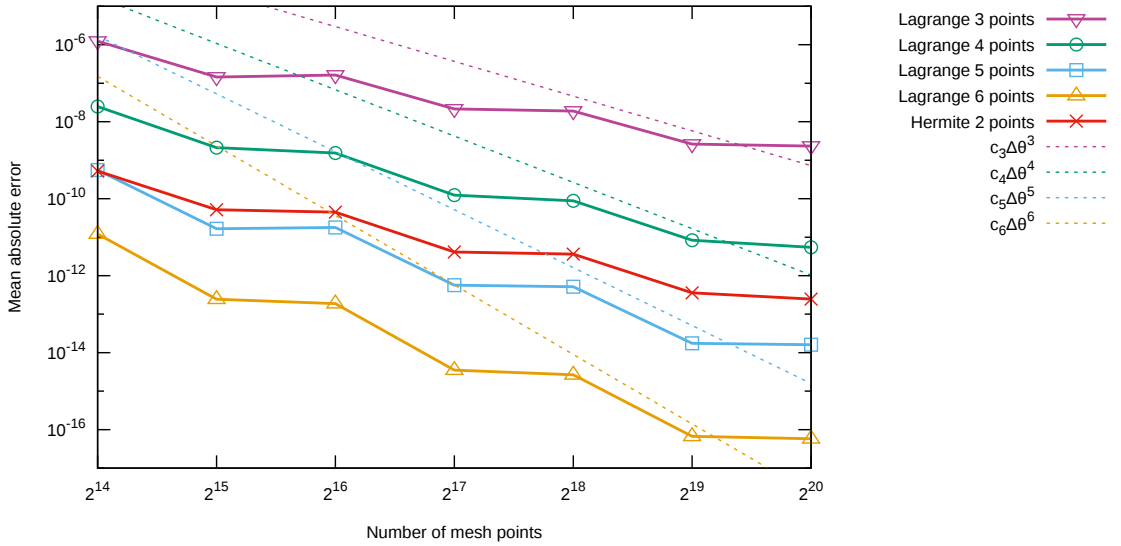


FIGURE 4.8 – Norme L_1 de l'erreur lors de l'estimation de la gyromoyenne de la fonction (4.8) utilisant différents interpolateurs en fonction de la discrétisation du maillage et pente théorique de convergence pour les interpolateurs de Lagrange. Le maillage pour 2^{14} points est le suivant ($128\times128\times64\times32\times1$).

calcul de dérivée. Pour une même précision (entre quatre et cinq points d'après la Figure 4.8), le nombre des points dans la matrice M_{coef} est très proche et l'étape de calcul des dérivées n'est plus nécessaire. La version utilisant l'interpolation de Lagrange est donc plus rapide comme le confirment les temps d'exécution donnés en Figure 4.9. La configuration est semblable aux précédentes (maillage de $(512\times512\times64\times32\times1)$ points). La version avec l'interpolation de Lagrange 6 points est ici environ 20% plus rapide que la version avec l'interpolation d'Hermite (2 points + 4 points pour les dérivées) bien qu'utilisant un *stencil* plus étendu.

L'interpolation par les polynômes de Lagrange présente de meilleurs résultats par rapport à l'interpolation d'Hermite que ce soit vis-à-vis des performances en évitant le calcul des dérivées, ou de la précision. Son implémentation dans le code GYSELA permet d'équilibrer précision et temps de calcul de manière fine en choisissant la taille du *stencil*. Ainsi, l'intégration des coefficients du calcul des dérivées directement dans M_{coef} pour l'interpolation d'Hermite (afin de supprimer la phase de calcul et stockage des dérivées comme évoqué en section 2.2.2) aurait entraîné une amélioration des performances (intensité computationnelle, localité spatiale et temporelle). Finalement, cet interpolateur est mieux adapté aux maillages faiblement structurés comme nous le verrons dans la section 4.3 qui suit.

4.3 Maillage poloïdal réduit

En complément des travaux de comparaison des interpolations d'Hermite et de Lagrange dans le code GYSELA sur le maillage poloïdal classique, une autre étude a été conduite sur un nouveau maillage poloïdal réduit. Ce type de maillage permet de réduire le nombre de points du plan poloïdal (concentration de points au centre dû au maillage polaire) d'au moins 15% tout en conservant la même précision. Il permet également de

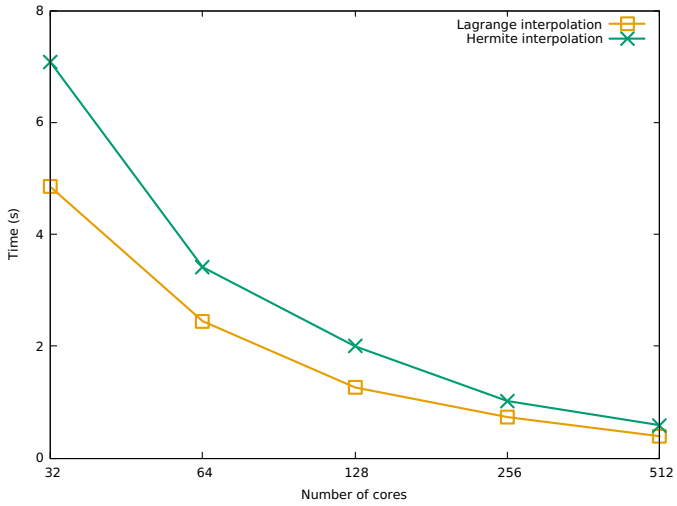


FIGURE 4.9 – Temps d'exécution moyen du diagnostic moment fluides avec la gyromoyenne distribuée utilisant différents interpolateurs. Le maillage est de (512×512×64×32×1) points.

définir de manière flexible des zones précises à étudier (étude physique des bords extérieurs par exemple) qui seront plus finement résolues que le reste du plan. Ces travaux ont été menés sur un prototype 2D dans le cadre du projet TaRGeT et ont fait l'objet d'une publication dans ESAIM [8]. Ils sont centrés autour de la validation numérique du maillage réduit et de l'adaptation des différents opérateurs. Leur performance n'est pas évaluée ici mais sera détaillée dans le cadre d'un prototype 4D dans les chapitres 5, 6 et 7. L'objectif majeur de ces travaux est d'atteindre une réduction de 90% de la taille du maillage lorsque l'on prend en compte la frontière extérieure du tore (zone du piédestal et/ou *scrape-off layer*) ; la résolution au centre du plan serait fortement réduite pour ces simulations. Le maillage réduit est d'abord présenté en section 4.3.1, puis les opérateurs d'interpolation, de gyromoyenne et d'advection sont adaptés dans les sections 4.3.2, 4.3.3 et 4.3.4.

4.3.1 Maillage réduit

Le maillage polaire réduit est défini à partir du maillage polaire régulier classique utilisé dans GYSELA (voir section 1.3.1). Dans le maillage classique, les différents rayons sont définis par $r_i = r_{\min} + i\Delta r$ avec $i \in [\![0, N_r - 1]\!]$ et les angles par $\theta_j = \frac{j\,2\pi}{N_\theta}$ avec $j \in [\![0, N_\theta - 1]\!]$. Le rayon central $r_{\min}$ est considéré large (de l'ordre de $0.1 r_{\max}$) et chaque opérateur possède un schéma spécifique pour traiter le bord du centre du plan. Le détail de ces schémas peut être trouvé dans [36]. Le nouveau maillage poloïdal réduit propose d'avoir un nombre de points en θ variable pour chaque rayon. Par exemple sur la figure 4.10a, le premier rayon r_0 (aussi appelé $r_{\min}$) a six points, le rayon r_1 a douze points, le rayon r_2 vingt-quatre et les rayons r_3 et r_4 quarante-huit. La discrétisation en θ n'est pas nécessairement croissante avec le rayon. D'autre part, le rayon central est fixé de sorte à ne plus avoir de trou central et de conditions de bords artificielles comme c'était le cas auparavant. Cela permet d'étendre les schémas numériques pour qu'ils s'appliquent sans condition de bord au centre avec de simples ajustements mineurs. Des approches similaires ont été étudiées dans un certain nombre de publications [41, 52, 53]. Néanmoins, le cadre

mathématique et physique y est relativement différent de celui de la physique des plasmas.

Le maillage réduit peut-être décrit mathématiquement comme suit. Le rayon central vaut $r_{\min} = \frac{\Delta r}{2}$ et le pas d'espace en r est défini comme $\Delta r = \frac{r_{\max} - r_{\min}}{N_r - 1}$, ce qui s'écrit également

$$\Delta r = \frac{r_{\max}}{N_r - \frac{1}{2}}.$$

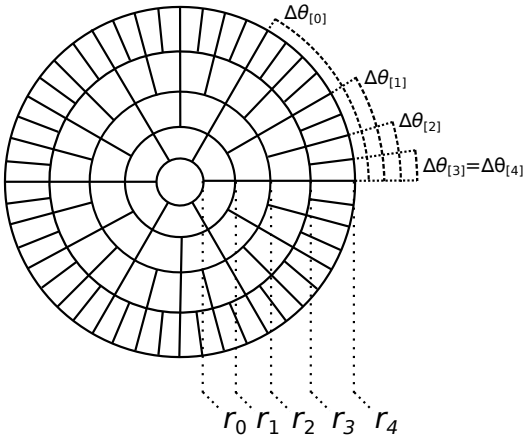
Ainsi les différents rayons valent

$$r_i = (i + \frac{1}{2})\Delta r, \quad i \in [\![0, N_r - 1]\!].$$

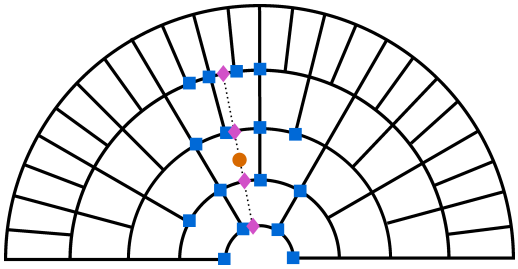
Ensuite, pour chaque rayon r_i, le nombre de points dans la direction θ est donné par $N_{\theta[i]}$ et le pas d'espace en θ par

$$\Delta\theta_{[i]} = \frac{2\pi}{N_{\theta[i]}}.$$

Sur l'exemple donné en Figure 4.10a, la configuration est donc la suivante : $N_r = 5$, $N_{\theta[0]} = 6$, $N_{\theta[1]} = 12$, $N_{\theta[2]} = 24$, $N_{\theta[3]} = 48$, et $N_{\theta[4]} = 48$. Choisir séparément la résolution du maillage à chaque rayon permet de rester au plus près des caractéristiques du plasma en raffinant les régions où les turbulences sont plus fines. Dans les simulations où le plan poloïdal est déjà suffisamment discretisé, cette approche permet de réduire le nombre de points utilisés à précision constante afin d'améliorer les temps d'exécution.



(a) Maillage poloïdal réduit. Le nombre de points dans la direction θ dépend de la position radiale.



(b) Interpolation de Lagrange d'un point du plan poloïdal par un *stencil* de seize points (Lagrange de degré 3).

FIGURE 4.10

4.3.2 Interpolation de Lagrange

Schéma numérique

Partant de l'équation de l'interpolation de Lagrange 2D (4.3) développée en section 4.2.1 sur un maillage poloïdal régulier, il est possible de l'adapter facilement au maillage poloïdal réduit. Le maillage réduit est discontinu dans la direction r, c'est-à-dire qu'à partir d'un point d'un rayon fortement discretisé, il arrive fréquemment que l'on ne trouve pas

de point de même coordonnée θ sur un rayon moins discretisé. Il n'est alors pas possible d'effectuer directement l'interpolation dans la direction r comme c'était le cas pour le maillage uniforme. L'interpolation d'une fonction f en un point $(\tilde{r}, \tilde{\theta})$ est donc effectuée en deux étapes détaillées par l'équation 4.9. Une première étape d'interpolation dans la direction θ est effectuée sur chaque rayon inclus dans le *stencil* afin de construire les valeurs aux points $(r_i, \tilde{\theta})$, puis une seconde étape d'interpolation dans la direction r est réalisée en utilisant les valeurs calculées précédemment. Cette équation est une réécriture de l'équation (4.3). $N_\theta(r)$ étant une fonction du rayon, on a désormais $r_h \leq r < r_{h+1}$ et $\theta_{k[i]} \leq \theta < \theta_{k[i]+1}$ où $h \in [\![0, N_r - 1]\!]$ et $k[i] \in [\![0, N_{\theta[i]} - 1]\!]$.

$$\begin{aligned} f(r_i, \tilde{\theta}) &= \sum_{j=k[i]+l}^{k[i]+u} f(r_i, \theta_j) N_j^{(p)}(\tilde{\theta}) \\ L^{(p)}(\tilde{r}, \tilde{\theta}) &= \sum_{i=h+l}^{h+u} f(r_i, \tilde{\theta}) M_i^{(p)}(\tilde{r}). \end{aligned} \tag{4.9}$$

La Figure 4.10b montre l'exemple d'une interpolation 4×4 points dont le *stencil* est réparti sur quatre rayons avec quatre $\Delta\theta$ différents. Les interpolations en θ des points ♦ d'angle $\tilde{\theta}$ sur les rayons r_0, r_1, r_2 et r_3 sont effectuées à partir des quatre points ■ les plus proches pour chaque rayon afin de maximiser la précision de l'opérateur.

D'autre part, puisque $r_0 = \frac{\Delta r}{2}$, il est possible d'éviter la mise en place de conditions de bord. Le centre se comporte comme un pas de maillage normal en r et son diamètre vaut Δr. L'interpolateur doit toutefois utiliser un schéma spécial puisque les coordonnées des points de l'autre côté du centre ne suivent plus la même indexation. Par exemple sur la Figure 4.11, on souhaite interpoler un point se situant entre les rayons r_0 et r_1. Les rayons du *stencil* seront dans ce cas les rayons r_2, r_1, r_0, r_{-1} et r_{-2}. Les rayons r_{-1} et r_{-2} n'existent pas et correspondent aux rayons r_0 et r_1 ; on modifie alors la cordonnée θ des points du *stencil* appartenant à ces rayons : l'angle θ est incrémentée de π par rapport à celle des points des rayons positifs.

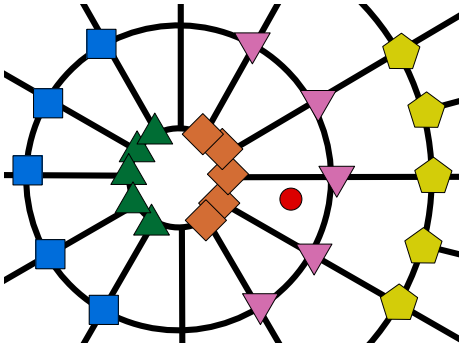


FIGURE 4.11 – Exemple de prolongation d'un *stencil* d'interpolation 5×5 à travers le centre du plan.

Résultats numériques

L'interpolateur est un opérateur central car il sert de fondement à d'autres opérateurs (gyromoyenne et advection). Ainsi, il est essentiel qu'il garde une précision aussi haute que

nécessaire. Comme dans la section 4.2.2 la précision de l'interpolation de Lagrange dépend de trois paramètres : le degré et les pas d'espace dans les directions r et θ.

Afin d'éprouver l'implémentation de l'interpolateur de Lagrange sur le maillage réduit, celui-ci est initialisé avec un produit de fonctions sinusoïdales défini par $f(x, y) = \sin(5x) \times \cos(4y)$. La fonction est donc relativement perturbée sur le disque de rayon $r_{\max}$ et ne présente pas d'alignement avec les directions d'interpolation. Un ensemble d'interpolations sur une grille cartésienne de taille $2N_r \times 2N_r$ et de dimensions $[-r_{\max} : r_{\max}] \times [-r_{\max} : r_{\max}]$ est effectué. Les points en dehors du plan poloïdal sont écartés. Par la suite, les courbes présentées dans les figures donnent les normes L_1, L_2 et L_∞ de l'erreur effectuée dans l'interpolation de la grille cartésienne pour le maillage réduit (*non-uniform*) et le maillage régulier (*uniform*).

La Figure 4.12 présente l'erreur en fonction du degré du polynôme d'interpolation qui varie de un à quinze. Le maillage utilisé pour le plan poloïdal est de 256×256 points pour le maillage régulier. Le maillage réduit possède également 256 rayons et les $N_{\theta[i]}$ sont définis par $N_\theta = (2 : \underline{32}, 8 : \underline{64}, 64 : \underline{128}, 182 : \underline{256})$. Cette notation se lit comme suit : les deux premiers rayons possèdent 32 points ($N_{\theta_{[0]}} = N_{\theta_{[1]}} = 32$), les 8 suivants en possèdent 64 et ainsi de suite. Ce maillage possède 15% de points en moins que le maillage régulier avec 256×256 points. Le comportement de l'interpolateur de Lagrange est le même sur les deux maillages : d'abord une phase de convergence qui suit la pente attendue [3] puis une phase de plateau lorsque l'on atteint la précision machine (10^{-15} en flottants double précision). Le maillage réduit a environ un degré de retard sur le maillage régulier car le jeu de $N_{\theta[i]}$ n'est pas adapté à la fonction de test, mais cela ne remet pas en cause la pertinence de l'approche.

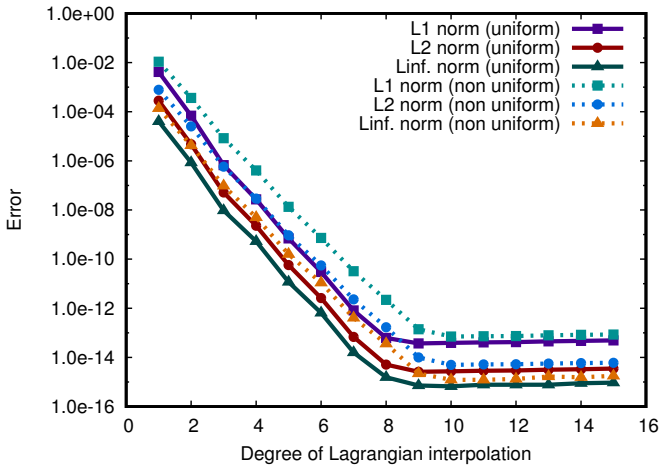
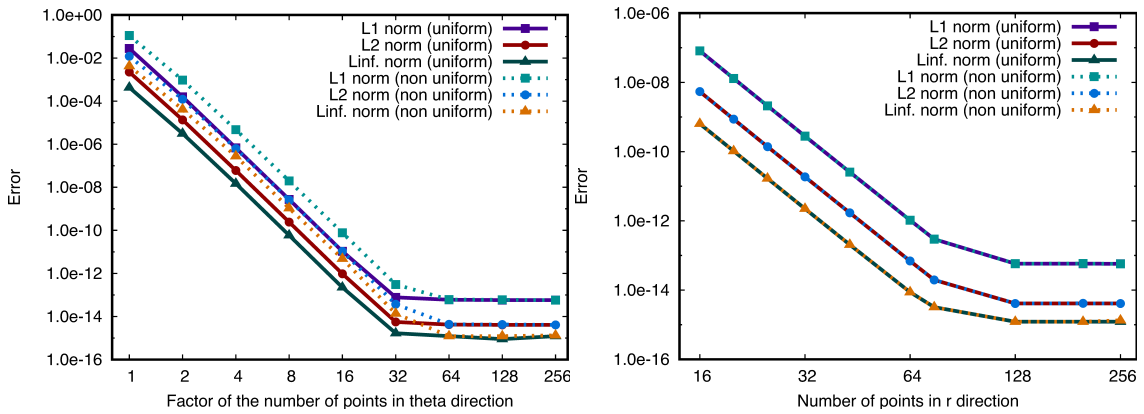


FIGURE 4.12 – Convergence de l'erreur en fonction du degré pour l'interpolation de Lagrange 2D.

La Figure 4.13a représente la convergence de l'erreur en fonction du facteur de discrétisation en θ. Pour le maillage régulier, la configuration en facteur 1 est $N_r = 256$, $N_\theta = 16$, et $N_r = 256$, $N_\theta = 16 \times 2^n$ pour un facteur 2^n variant entre 1 et 256. Pour le maillage réduit la configuration correspondante à un facteur 2^n est $N_r = 256$, $N_\theta = (2 : \underline{2 \times 2^n}, 8 : \underline{4 \times 2^n}, 64 : \underline{8 \times 2^n}, 182 : \underline{16 \times 2^n})$. Le degré des polynômes des Lagrange est fixé à 7. Ici, les comportements et les écarts sur les deux maillages sont les mêmes que pour la convergence en degré. Ceci confirme la bonne adaptation de l'interpolateur au maillage réduit. Fina-



(a) Convergence en θ. (b) Convergence en r.

FIGURE 4.13 – Convergence de l'erreur en espace pour l'interpolation de Lagrange 2D.

lement, la Figure 4.13b donne la convergence de l'erreur en fonction de la discrétisation dans la direction r. Les configurations utilisées ici sont caractérisées par $N_\theta = 2048$ pour le maillage régulier et une progression de $N_{\theta[0]} = 256$ à $N_{\theta[N_r-1]} = 2048$ pour le maillage réduit. La discrétisation en θ est suffisamment importante pour ne pas influer sur la précision de l'interpolation. Le nombre de rayons varie de 16 à 256. On observe encore le même comportement sauf que les deux courbes sont ici confondues. Cela s'explique par le fait que le schéma d'interpolation n'a pas changé pour la direction r ; les résultats sont donc égaux.

Sur ces fonctions simples (produit de sinusoïdes), il est possible de trouver des configurations de $N_{\theta[i]}$ adaptées mais complexes qui permettent d'atteindre une réduction par deux du nombre de points en conservant la même précision. Aucune règle n'a pu être déduite pour générer automatiquement des configurations optimales pour des fonctions de distributions quelconques. On peut néanmoins conclure que l'opérateur d'interpolation Lagrangienne fonctionne de façon satisfaisante sur le maillage réduit.

4.3.3 Opérateur de gyromoyenne

Schéma numérique

L'opérateur de gyromoyenne implémenté ici utilise l'équation discrète (2.5) donnée en section 2.1.1 et rappelée ci-dessous. Aucune optimisation spécifique n'a été mise en œuvre ici, le but de ce prototype n'étant pas l'évaluation des performances. Seul l'opérateur d'interpolation change et est remplacé par l'interpolateur de Lagrange adapté au maillage réduit introduit en section 4.3.2.

$$\mathcal{J} \cdot f(\mathbf{x}, \mathbf{v}) \approx \frac{1}{N_\mathcal{L}} \sum_{k=0}^{N-1} f(\mathbf{x} + \mathbf{x}_{\rho_\mathcal{L}}(\frac{k}{N_\mathcal{L}}), \mathbf{v}). \tag{2.5}$$

Résultats numériques

Comme en section 4.2.2, les fonctions de Fourier-Bessel sont utilisées afin de vérifier la précision de l'opérateur de gyromoyenne. La fonction considérée ici est une fonction de

Bessel de première espèce d'ordre 3. Elle est représentée en Figure 4.14 et est donnée par l'équation suivante :

$$f(r,\theta) = J_3(j_{3,1}\frac{r}{r_{\max}})e^{3i\theta}. \tag{4.10}$$

De même que pour l'interpolation de Lagrange, des tests de convergence de l'erreur ont été menés en fonction des dimensions r et θ et du degré de l'interpolation. Lors de ces travaux et par le passé [62], des études ont également été conduites en prenant en compte l'impact du nombre de points de Larmor, de l'ordre de la fonction de Bessel et du rayon de Larmor. Les résultats présentant des comportements similaires et à des fins de concision, seule l'étude pour les paramètres suivants sera détaillée : f telle que définie par (4.10), $N_{\mathcal{L}} = 128$ et $\rho_{\mathcal{L}} = 0.1$.

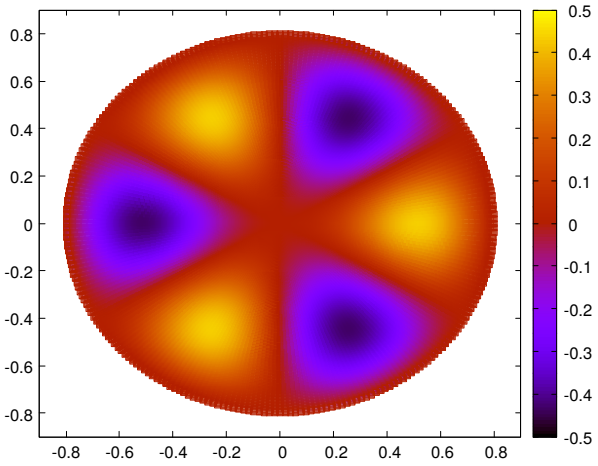
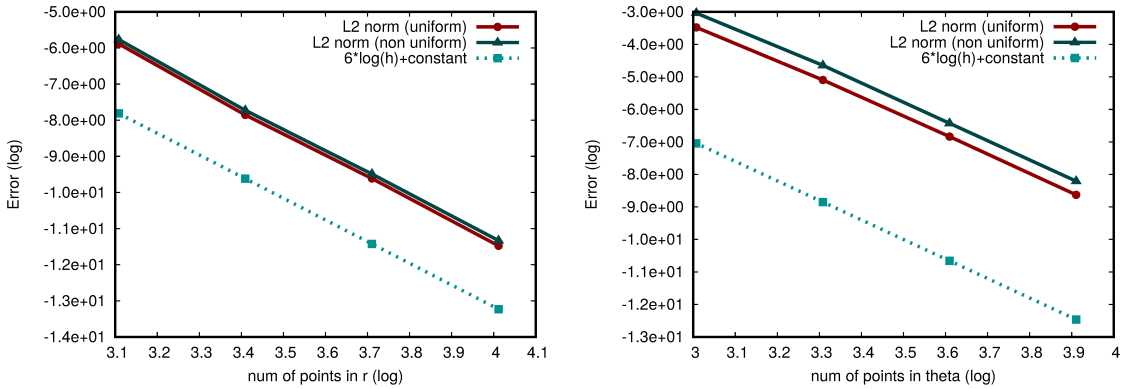


FIGURE 4.14 – Carte de couleurs représentant la fonction de Bessel de première espèce donnée par l'équation (4.10).

Les Figures 4.15 et 4.16 présentent la norme L_2 de l'erreur effectuée lors du calcul de la gyromoyenne de l'ensemble des points du plan poloïdal. La Figure 4.15a montre l'évolution de l'erreur en fonction de la discrétisation en r. Le nombre de rayons varie entre 32 et 256 par puissances de 2. La configuration utilisée en θ est $N_\theta = 512$ pour le maillage régulier et une progression de $N_{\theta[0]} = 32$ à $N_{\theta[N_r-1]} = 512$ pour le maillage réduit. La Figure 4.15b montre l'évolution de l'erreur en fonction de la discrétisation en θ. La configuration est $N_r = 256$ pour les deux maillages et les maillages les moins résolus ont $N_\theta = 32$ pour le maillage régulier et $N_\theta = (10 : \underline{4}, 30 : \underline{8}, 50 : \underline{16}, 166 : \underline{32})$ pour le maillage réduit. Le nombre de points en θ est ensuite multiplié par deux pour chacun des points suivants sur la Figure. Pour les deux Figures 4.15, on observe le même comportement que pour l'interpolation. Les deux maillages suivent la même courbe et le maillage réduit est légèrement en retrait pour la convergence en θ. De plus, la courbe en pointillés donne la courbe théorique du logarithme de la convergence spatiale pour une interpolation de degré 5. Sa pente vaut $p \cdot log(h) + c$ où h est le pas du maillage, c une constante et p la taille du *stencil* pour l'interpolation de Lagrange (le degré étant $p-1$) [3]. La Figure 4.16 permet d'observer les pentes de convergence pour différents degrés d'interpolation (5, 7 et 9) sur une convergence en θ. Les résultats sont conformes aux attentes théoriques.

L'opérateur de gyromoyenne montre des résultats satisfaisants et semble être bien adapté au maillage réduit. Cela est essentiellement dû au bon comportement de l'inter-



(a) Convergence en r. (b) Convergence en θ.

FIGURE 4.15 – Convergence de l'erreur en espace pour l'opérateur de gyromoyenne.

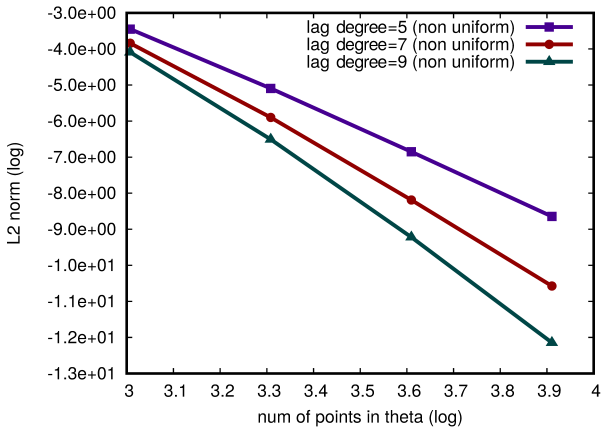


FIGURE 4.16 – Convergence de l'erreur en θ pour l'opérateur de gyromoyenne avec une interpolation de Lagrange de différents degrés.

polation (voir section 4.3.2).

4.3.4 Opérateur d'advection

Schéma numérique

L'opérateur d'advection consiste à transporter des quantités scalaires ou vectorielles dans le domaine de calcul en y appliquant un champ de déplacement. Dans le cas des turbulences du plasma, ce déplacement provient de la vitesse des particules qui est influencée par les variations du champ électromagnétique. Plus de détails concernant l'advection de Vlasov seront donnés dans la section 6.1, dédié à son implémentation dans un prototype 4D. Pour l'instant, l'opérateur est considéré comme l'application simple d'un champ de déplacement sur une grandeur du maillage. La Figure 4.17 présente l'advection d'un point du maillage. Le schéma utilisé est appelé *Backward Semi-Lagrangian*. C'est un schéma inverse, ce qui signifie que partant des points du maillage au temps $t^{N+1}(= t^N + \Delta t)$, l'advection est effectuée à l'envers afin de retrouver le point d'origine au temps t^N. La valeur de la fonction en ce point est ensuite calculée avec l'interpolateur de Lagrange et assignée au

point du maillage à t^{N+1}.

L'équation résolue par l'opérateur d'advection pour une fonction f en un point (r, θ) est donnée par

$$f(r, \theta, t^{N+1}) = f(r - v_r \Delta t, \theta - v_\theta \Delta t, t^N) \tag{4.11}$$

où v_r et v_θ sont les vitesses du point dans les dimensions r et θ et Δt représente le pas de temps. De même que l'opérateur de gyromoyenne, l'opérateur d'advection repose entièrement sur l'interpolation et sa précision lui est donc entièrement liée. Ce test est représentatif de l'évolution de la turbulence dans le code gyrocinétique GYSELA. Les vortex sont transportés dans le plan poloïdal essentiellement sur la direction θ, mais aussi dans la direction r.

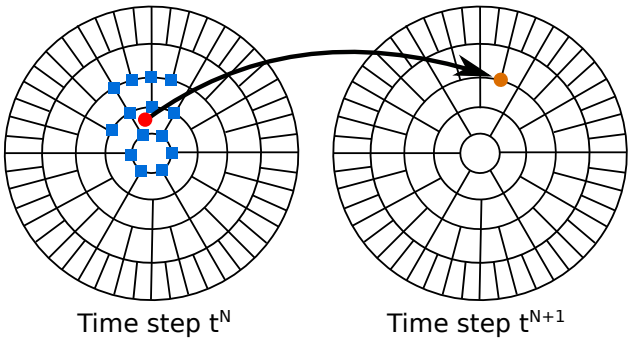


FIGURE 4.17 – Advection d'un point ● du plan poloïdal sur le maillage au temps t^{N+1} et interpolation de Lagrange de degré 3 du point ● originel au temps t^N à partir des points ■.

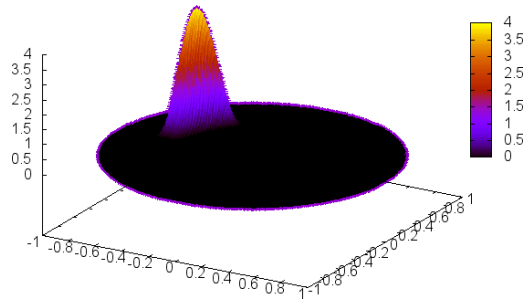
Résultats numériques

Plusieurs tests similaires à ceux étudiés pour l'interpolation et la gyromoyenne ont été menés. Les convergences en fonction des dimensions de l'espace et du degré de l'interpolation montrent les mêmes tendances pour l'advection sur un pas de temps que pour l'interpolation. Elles ne seront pas présentées ici. Il est intéressant d'étudier l'évolution de l'erreur sur plusieurs advections successives et notamment lorsque des structures passent à travers le centre du plan qui est la région la plus critique. Les configurations utilisées pour les maillages réduits et réguliers sont les mêmes qu'en section 4.3.2 : $N_r = 256$, $N_\theta = 256$ pour le maillage régulier et $N_\theta = (2 : \underline{32}, 8 : \underline{64}, 64 : \underline{128}, 182 : \underline{256})$ pour le maillage réduit, c'est-à-dire 15% de points en moins que pour le maillage régulier. La fonction dont l'évolution est étudiée a la forme suivante :

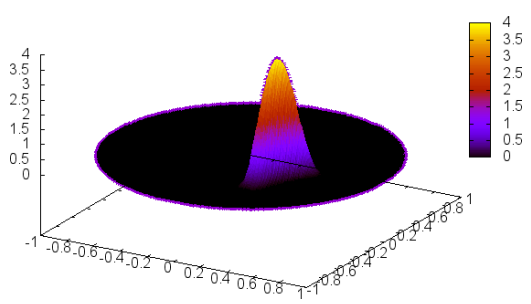
$$f(r, \theta) = \begin{cases} \cos(1 - 2\pi \frac{r - r_2}{r_1 - r_2})(1 - \cos(2\pi \frac{\theta - \theta_2}{\theta_1 - \theta_2})) & \text{si } (r, \theta) \in [r_1, r_2] \times [\theta_1, \theta_2] \\ 0 & \text{sinon} \end{cases} \tag{4.12}$$

où $r_{min} \leq r_1 < r_2 \leq r_{max}$ et $0 \leq \theta_1 < \theta_2 \leq 2\pi$. La fonction est définie par ($r_1 = 0.6$, $r_2 = 0.8$, $\theta_1 = \frac{5\pi}{7}$, $\theta_2 = \pi$) avec $r_{max} = 1.0$. Cette fonction, centrée en $(0.7, \frac{6\pi}{7})$, est représentée en Figure 4.18a. Elle est de classe $\mathcal{C}^1$ tel que nécessaire pour l'interpolation de Lagrange. Pour l'étude de l'évolution de l'erreur, la fonction est advectée à travers le centre du plan poloïdal avec une vitesse $(v_x = \frac{2}{3}, v_y = -\frac{1}{3})$ jusqu'à atteindre $r = 0.7$ du côté opposé. La structure est ensuite ramenée en position initiale en subissant l'advection

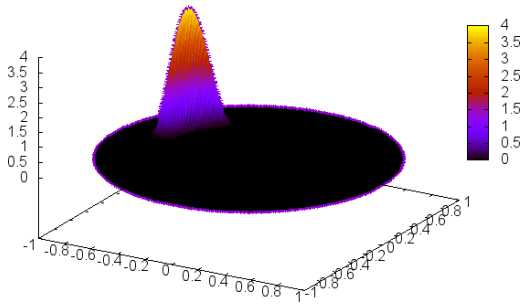
inverse. L'ensemble du déplacement est effectué sur quarante pas de temps. La Figure 4.18 présente la solution exacte de l'advection aux pas de temps 1, 20 et 40. Les Figures 4.19a et 4.19c montrent l'erreur absolue aux pas de temps 20 et 40 sur le maillage régulier et les Figures 4.19b et 4.19d sur le maillage réduit. On observe que l'erreur n'est pas nulle, croît dans le temps et est plus prononcée pour le maillage réduit. Avec l'ensemble des pas de temps, on remarque également que l'évolution la plus conséquente de l'erreur se déroule lors du premier passage de la structure au centre du plan.



(a) Fonction au pas de temps 1.



(b) Fonction au pas de temps 20.

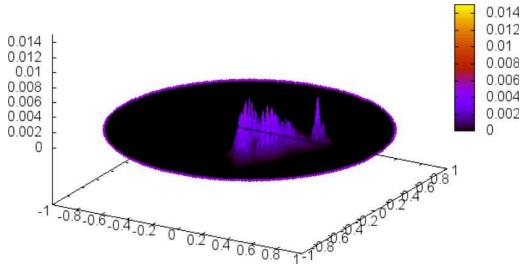


(c) Fonction au pas de temps 40.

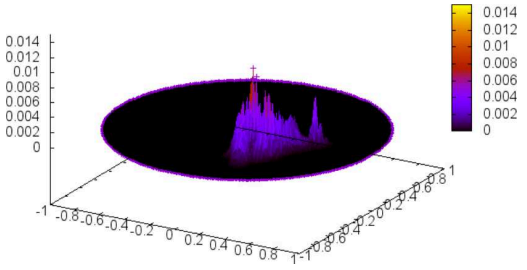
FIGURE 4.18 – Évolution de la fonction de référence (4.12) lors d'une advection aller-retour à travers le centre du plan poloïdal.

Cela se lit plus aisément sur la Figure 4.20 qui donne les normes L_1, L_2 et L_∞ de l'erreur sur l'ensemble du plan à chaque pas de temps pour les deux maillages. On a d'abord une perte de précision qui croît régulièrement avant de subir une accélération vers le centre du plan autour du pas de temps 12 pour le maillage régulier et 8 pour le maillage réduit. Cela signifie qu'un des $N_{\theta[i]}$ dans cette zone du plan réduit n'est pas assez élevé pour atteindre la même précision qu'en régulier. De plus, les rayons inférieurs ne sont également pas assez résolus puisque l'advection sur le maillage régulier a une meilleure précision. Une fois la structure passée à travers la partie la moins raffinée du plan, l'erreur n'augmente plus et stagne à un palier qui correspond à l'erreur minimale de la configuration (maillage + degré d'interpolation). Ces résultats se montrent tout à fait satisfaisants.

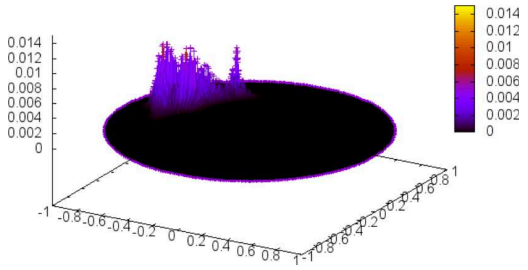
Une fois encore l'opérateur se comporte comme attendu. La précision dépend fortement du choix des $N_{\theta[i]}$, surtout près du centre où la plus grosse réduction du nombre de points était espérée. Il pourrait être utile de disposer d'un outil d'ajustement qui pourrait calculer la distribution des $N_{\theta[i]}$ en fonction de la précision souhaitée, du nombre de rayons et des variations typiques de la fonction (taille des structures à étudier). Dans ce but, une catégorie de maillages réduits ayant une densité de points presque uniforme a été développée, mais ne montre pas de bons résultats sur l'ensemble des opérateurs. Leur construction peut



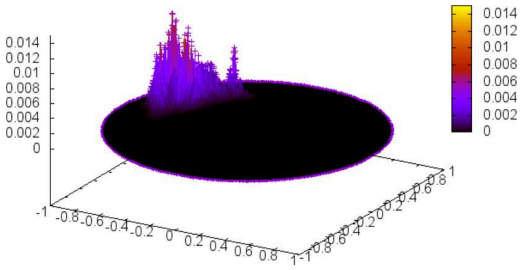
(a) Erreur au pas de temps 20 pour le maillage régulier.



(b) Erreur au pas de temps 20 pour le maillage réduit.



(c) Erreur au pas de temps 40 pour le maillage régulier.



(d) Erreur au pas de temps 40 pour le maillage réduit.

FIGURE 4.19 – Évolution de l'erreur absolue pour une advection aller-retour à travers le centre pour un maillage régulier et un maillage réduit.

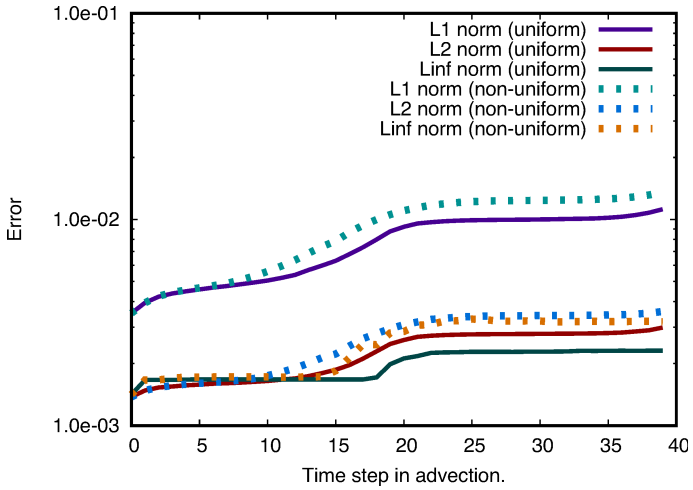


FIGURE 4.20 – Évolution de l'erreur absolue lors d'une advection aller-retour d'une fonction de référence à travers le centre du plan poloïdal.

être trouvée dans l'Annexe A. Le prototype d'évaluation du maillage poloïdal réduit donne néanmoins de bons résultats numériques, ajustable grâce à l'interpolateur de Lagrange, lorsque le maillage est défini spécifiquement pour chaque cas. Cela ouvre la voie à une amélioration des simulations GYSELA, soit pour améliorer les performances des simulations actuelles, soit pour réduire la taille des cas et se focaliser sur des zones particulières du tokamak telles que la zone du piédestal ou la *scrape-off layer*. De plus, un *mapping* en géométrie complexe a également été implémenté dans ce prototype sous la forme d'une

fonction analytique. Les résultats numériques étant similaires, ils ne sont pas présentés ici et le lecteur peut trouver de plus amples informations dans [8].

Les évolutions des paramètres physiques des simulations GYSELA ainsi que de la taille des configurations accessibles sur les machines modernes (grands plans poloïdaux) nécessitent une évolution du schéma numérique de l'opérateur de gyromoyenne. Celui-ci ne peut plus s'appliquer à toutes les modélisations visées par GYSELA. Les solutions proposées dans ce chapitre ne sont pas complètement satisfaisantes. Leurs coûts d'implémentation ou leurs temps d'exécution restent problématiques dans certaines configurations. Un nouveau maillage poloïdal réduit est proposé et permet de réduire fortement le nombre de points au centre du plan. L'interpolation de Lagrange, plus adaptée à ce maillage faiblement structuré donne également de meilleurs résultats que l'interpolation d'Hermite. Le maillage réduit devrait permettre de diminuer la taille des simulations dans certains cas particuliers ; l'objectif de réduire de 90% le nombre de points est ainsi réaliste. Cela ouvre la voie à certaines propositions formulées dans l'optique d'une évolution du code dont une esquisse est donnée en Figure 4.21 (ici en géométrie cartésienne dans le plan). Que le maillage soit cartésien ou polaire, l'ambition serait de raffiner le maillage au bord du domaine afin de s'adapter à des gradients forts et de petites structures se développant localement. L'intégration de ce nouveau maillage dans GYSELA est néanmoins complexe de par l'ampleur des modifications à effectuer. Il faut en effet modifier nombre de structures

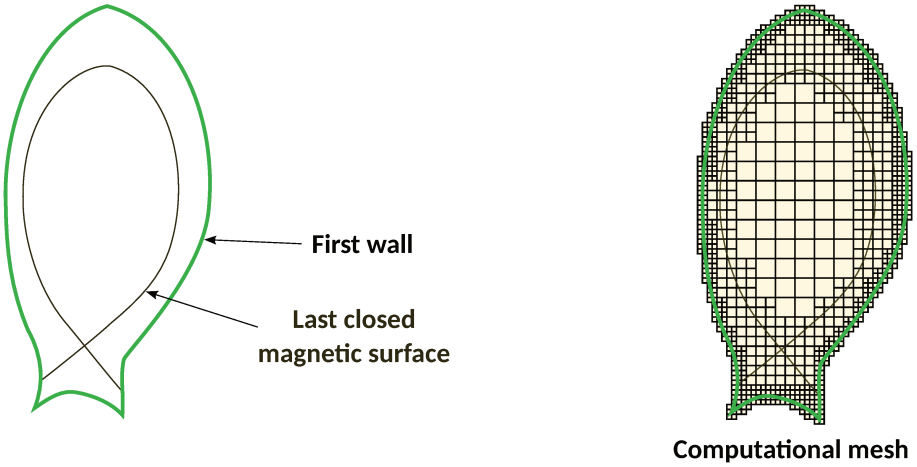


FIGURE 4.21 – Exemple de géométrie complexe dans un tokamak et exemple de maillage adaptatif pour une étude de la *scrape-off layer*.

de données internes et modifier la parallélisation dans l'ensemble du code. La distribution de donnée ne peut plus s'effectuer simplement en découpant la dimension r en puissance de 2 ; cela entraînerait un déséquilibre de charge, les processus ayant des sous-domaines mieux résolus étant alors responsables de plus de points. C'est pourquoi l'écriture d'une nouvelle version du code GYSELA proposant également d'autres améliorations (point X, maillage non-uniforme dans plusieurs dimensions) est envisagée. Afin de préparer le terrain à cette nouvelle mouture et afin d'éprouver le maillage réduit sur des cas proches des conditions de production, un nouveau prototype 4D *drift-kinetic*, GYSELA++, a été développé. Il fait l'objet de la deuxième partie de ces travaux de thèse.

Deuxième partie

Étude du prototype Gysela++

Chapitre 5

Introduction à Gysela++

Sommaire

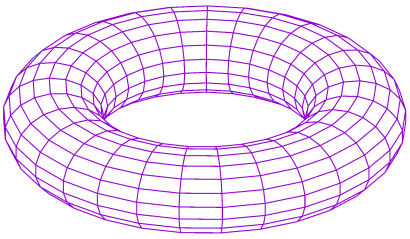
Gysela++ est un prototype de code gyrocinétique 5D de simulation de turbulences dans un plasma de tokamak. Il implémente le modèle 4D *drift-kinetic*, plus simple à mettre en œuvre et nécessaire au modèle gyrocinétique, tout en prévoyant une structure 5D. Il est écrit en C++ et sa finalité est d'expérimenter un certain nombre d'évolutions et d'optimisations vis-à-vis de la structure du code Gysela actuel. Les modifications envisagées n'étaient pas réalisables directement dans le code de production sans en réécrire une partie conséquente. Il a donc fallu envisager l'implémentation d'un prototype dans l'optique du développement d'un nouveau code GyselaX qui viendra en remplacement de Gysela et intégrera dès le début de nombreuses fonctionnalités (maillage adaptatif en espace et en vitesse, géométrie complexe en point X...). L'objectif est d'améliorer le passage à l'échelle sur des calculateurs pré-*exascales*, d'accéder à des simulations en géométries réalistes plus proches des architectures des réacteurs actuels et d'obtenir une plus grande étendue de cas envisageables. Des simulations fines du bord (*scrape-off layer* ou piédestal) ou de certaines régions centrales, d'intérêt pour les physiciens, seont alors possibles sans pour autant avoir à raffiner l'ensemble du maillage.

Gysela++ propose une géométrie adaptative et déformable inspirée des travaux développés en section 4.3. La structure de données mise en place utilisant des blocs structurés permet d'exposer et d'exploiter plus de parallélisme. Elle est introduite en section 5.1.

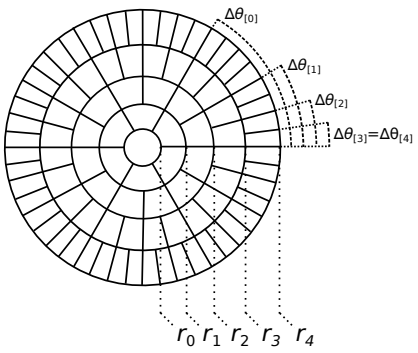
Le code possède une parallélisation basée sur les paradigmes MPI et OpenMP et focalisée sur l'exécution simultanée de séquences de calcul et de séquences de communication. Ce recouvrement calcul – communication est atteint, entre autres, grâce à l'utilisation de communications asynchrones et d'un modèle de programmation par tâches. Le modèle de communication est présenté en section 5.2.1. La parallélisation multicœur utilise des tâches OpenMP afin de créer une encapsulation qui facilite le développement et permet de mettre en évidence différents niveaux de parallélisation. La stratégie de découpage du problème en tâches est détaillée en section 5.2.2. Cette approche doit permettre d'améliorer la régulation de la charge ainsi que de réduire les coûts de communication et le nombre de synchronisations. En effet, GYSELA fait face à des coûts de communication élevés qui compromettent son passage sur des machines exascales. De nombreux points de synchronisation sont également présents, liés à l'enchainement des différents opérateurs du modèle gyrocinétique, et cela entraîne une consommation non négligeable d'heures de calcul par des cœurs inoccupés. Résoudre ces problèmes est un des objectifs de GYSELA++. Ce modèle de parallélisation (tâches et recouvrement asynchrone) a également été expérimenté dans d'autres codes de simulation de plasma [2, 57] pour lesquels il a su montrer ses avantages.

5.1 Géométrie et structure de données

GYSELA++ adopte une géométrie et un maillage logique qui intègre directement le maillage poloïdal réduit introduit en section 4.3.1. Le maillage est rappelé aux Figures 5.1. La géométrie spatiale décrit un tore en utilisant les variables r, θ et φ telles que présentées en section 1.3. L'espace des vitesses est dans un premier temps réduit à sa composante parallèle $v_\parallel$. La dimension μ est toutefois prise en compte, mais se compose d'un unique point et $\mu = 0$. Les directions φ et $v_\parallel$ sont maillées uniformément. Afin d'exacerber le parallélisme et de faciliter le découpage de la simulation en tâches, les calculs sont découpés sous forme de blocs appelés *tuiles*. Bien que données et calculs puissent être structurés séparément, il est plus efficace de lier les deux découpages pour des questions de localité et de lisibilité du code, entre autres. C'est ce qui a été fait pour GYSELA++ et nous ne parlerons plus que de découpage en tuiles. Les arguments soutenant ce choix sont détaillés par la suite.



(a) Maillage d'une surface magnétique (r = constante).



(b) Maillage du plan poloïdal.

FIGURE 5.1 – Représentation du maillage spatial dans GYSELA++.

5.1.1 Structuration du maillage en tuiles

Définition des tuiles

Les calculs et les données sont structurés en blocs 5D appelés «*tuiles*». Le découpage d'un jeu de données sous forme de tuiles afin de faciliter les calculs a déjà fait ses preuves dans d'autres domaines [13, 72]. Comme nous le verrons par la suite, cela permet de faciliter l'utilisation du modèle de programmation par tâches ainsi que la répartition des charges de calcul entre *threads*. En effet, le découpage en tuiles permet de répartir la complexité en implémentant une version de chaque opérateur adaptée à chaque type de tuile (conditions au bord extérieur, spécificité locale du maillage au centre et entre les différents $N_{\theta[i]}$, voir les sous-sections suivantes : «Stockage et halos» et «Interpolations»). La construction des tuiles se fait en deux étapes : d'abord dans les directions r et θ en partitionnant le plan poloïdal et ensuite dans les directions φ et $v_\parallel$ en conservant le découpage poloïdal calculé. Le découpage dans les directions φ et $v_\parallel$ se fait de façon uniforme, c'est-à-dire que la largeur des tuiles dans ces dimensions est la même pour toutes les tuiles. Le découpage dans le plan poloïdal est plus complexe à cause des conditions particulières au centre du plan et du fait du maillage réduit qui génère des changements de $N_{\theta[i]}$ (couronnes). Est définie comme couronne, une succession de rayons ayant la même discrétisation, c'est-à-dire le même nombre de points en θ ($N_{\theta[i]}$). Plusieurs catégories de tuiles sont définies pour l'ensemble du plan comme présenté en Figure 5.2. La tuile «centrale» englobe l'ensemble des points au centre du plan sur 360 degrés et sur un rayon qui dépend des paramètres physiques de la simulation (tuile 0). Cette unique tuile centrale permet de pallier les problèmes de certains opérateurs au centre, notamment la gyromoyenne (la hauteur en r est calculée de sorte que la première rangée de tuiles ne souffre pas des problèmes débattus en section 4.1) et l'advection [1]. Les tuiles «régulières» sont situées à l'intérieur d'une couronne et ont une hauteur et une largeur fixe (tuiles 1 et 2). Ce sont les tuiles les plus simples, sans condition particulière. Les tuiles «jonctions» chevauchent deux couronnes et en effectuent la jonction (tuiles 3 et 4) ; les opérateurs doivent prendre en compte les deux $N_{\theta[i]}$ différents dans leurs calculs, qui sont alors plus complexes. Finalement, les tuiles «bords» situées à l'extérieur du plan sont similaires aux tuiles régulières, mais appliquent des conditions spécifiques au bord extérieur (tuiles 5 à 8).

Il existe également des tuiles dites «interfaces» qui viennent en remplacement des tuiles jonctions. Contrairement aux tuiles jonctions qui chevauchent deux couronnes et encapsulent le changement de discrétisation en θ, les tuiles interfaces vont par deux et se placent face-à-face, une sur chaque couronne. La complexité de la jonction est alors portée sur les échanges de données entre les deux tuiles qui doivent prendre en compte ce changement de discrétisation. Cette approche n'a néanmoins pas été explorée plus en détail, car nous souhaitions à tout prix éviter d'alourdir les phases de communication qui sont un point bloquant pour le passage à l'échelle du code Gysela. Avec les tuiles interfaces, il aurait alors été nécessaire de générer les données à échanger entre les deux tuiles (halos, voir sous-sections suivantes) et d'utiliser un *buffer* supplémentaire pour les communications (alors que les données sont prises directement à leur emplacement de stockage pour les communications des autres tuiles). Ces tuiles n'ont donc pas été utilisées.

Les dimensions des tuiles (n_r, n_θ, n_φ et $n_{v_\parallel}$) sont au choix de l'utilisateur et différentes politiques de découpage en tuiles peuvent être envisagées. La politique actuelle est simple et utilise les tailles de tuiles données en paramètre. L'algorithme 7 présente l'essentiel du déroulement du découpage en tuiles d'un plan poloïdal. L'idée est donc de suivre au

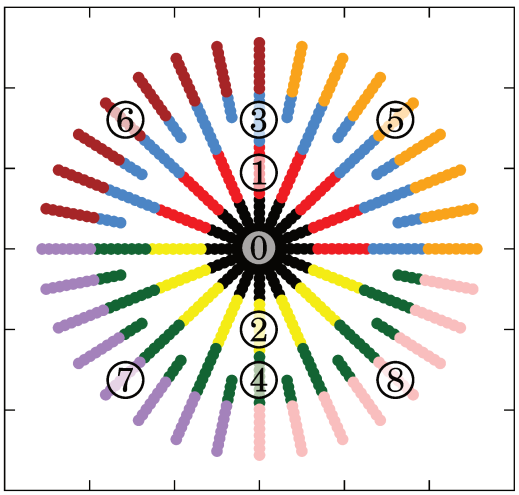


FIGURE 5.2 – Exemple de découpage en neuf tuiles du maillage d'un plan poloïdal à deux couronnes ($N_r = 32$, $N_\theta = (16 : \underline{20}, 32 : \underline{12})$). Tuiles : centrale = $\{0\}$, régulières = $\{1, 2\}$, jonctions = $\{3, 4\}$, bords = $\{5, 6, 7, 8\}$.

maximum les paramètres de dimension des tuiles en partant de r_0 ($=\frac{\Delta r}{2}$). D'abord, la tuile centrale est créée (ligne 1) en prenant essentiellement en compte le paramètre n_r et le rayon de Larmor $\rho_{\mathcal{L}}$ afin que la première rangée de tuiles puisse appliquer la gyromoyenne à l'ensemble de ses points en utilisant des données localisées au plus loin sur les tuiles voisines [1]. Puis des tuiles régulières (ligne 4) sont générées jusqu'à atteindre la seconde couronne moins la moitié de la hauteur en r d'une tuile. On met alors en place des tuiles jonctions (ligne 3) et l'on répète le processus (tuiles régulières puis jonctions éventuelles) jusqu'à $r_{\max}$ où l'extérieur du plan est pavé de tuiles bords (ligne 2). La hauteur des rangées de tuiles est adaptée durant le découpage en fonction des rayons où se situent les jonctions des couronnes de sorte que les rangées de tuiles jonctions fassent la taille spécifiée en paramètre. Les tuiles créées sont de taille n_φ et $n_{v_\parallel}$ dans les dimensions φ et $v_\parallel$. Le découpage poloïdal n'est effectué qu'une fois et cette partition est répétée pour générer le reste des tuiles dans les dimensions φ et $v_\parallel$. Le découpage présenté en Figure 5.2 a été généré par cet algorithme avec $n_r = 8$ et $n_\theta = 8$. Une version optimisée de cet algorithme de découpage en tuiles pourrait tenir compte des poids des différents types de tuiles en termes de temps d'exécution. Les tuiles jonctions sont les plus complexes puisqu'elles nécessitent des algorithmes et des accesseurs [2] adaptés. Puis viennent les tuiles de bord qui n'ont que de simples conditions au bord, et la tuile centrale dont la complexité de la gestion du centre dépend des opérateurs (interpolations et gyromoyennes) et du stockage des données (voir sous-sections suivantes). Les tuiles les plus simples sont les tuiles régulières sur lesquelles les opérateurs doivent s'exécuter rapidement. Cette prise en compte du poids pourrait également être faite à un autre niveau lors de la distribution des tuiles sur les différents processus MPI afin d'équilibrer la charge (voir section 5.1.2). Ce mécanisme n'a pas été implémenté dans le code pour l'instant.

1. Il serait également possible de réviser le schéma de communication des halos de la gyromoyenne, introduit au chapitre 3, de façon à l'adapter à chaque tuile (ne plus avoir un schéma de communication des halos unique), et ce d'autant que cela serait facilité par la structure de données.

2. Un accesseur est une primitive permettant d'accéder à une donnée depuis les coordonnées d'un point sur le maillage.

Algorithme 7 : Algorithme simplifié de découpage du plan en tuiles.

Entrée : N_r, $(N_{\theta[i]})$, n_r, n_θ
Sorties : Liste de tuiles `TL`

```
TL ← liste_vide()
L_couronne ← liste_couronnes((N_θ[i]))        /* P.ex. Figure 5.2 : {20,32} */
i_r ← 0; i_couronne ← 0
r_start ← 0; r_end ← 0
tant que r_end < N_r faire
    r_start ← r_end
    r_end ← min(r_start + n_r, L_couronne[i_couronne] − n_r/2)
1   si i_r = 0 alors                                          /* Centrale */
        TL.append(Tuile_centrale([0, r_end − 1], [0, N_θ[0] − 1]))
    sinon si r_end = r_start alors
2       si r_end = N_r − n_r/2 alors                          /* Bords */
            pour j = 0 .. N_θ[N_r−1] − 1 par n_θ faire
                TL.append(Tuile_Bord([r_start, N_r − 1], [j, j + n_θ − 1]))
            r_end = N_r
3       sinon                                                 /* Jonctions */
            r_mid ← L_couronne[i_couronne]
            r_end ← L_couronne[i_couronne] + n_r/2
            α ← N_θ[r_end] ÷ N_θ[r_start]
            pour j = 0 .. N_θ[N_r−1] − 1 par n_θ faire
                TL.append(Tuile_Jonction(([r_start, r_mid − 1], [r_mid, r_end − 1]),
                                         ([j, j + n_θ − 1], [α·j, α(j + n_θ) − 1])))
            i_couronne = i_couronne + 1
4   sinon                                                     /* Régulières */
        pour j = 0 .. N_θ[N_r−1] − 1 par n_θ faire
            TL.append(Tuile_Reguliere([r_start, r_end], [j, j + n_θ − 1]))
```

Interpolations

L'interpolation est un opérateur fondamental pour le code puisqu'il est utilisé comme composant de base dans d'autres opérateurs. Aussi, il est essentiel qu'il soit optimisé au maximum. Les interpolations sont effectuées en utilisant des polynômes de Lagrange dont le degré est paramétrable à la compilation. L'implémentation pour les dimensions $r\times\theta$ est reprise du prototype 2D développé en section 4.3.2. À cause du maillage poloïdal réduit, les interpolations 2D sont plus complexes pour certains points localisés près des jonctions des couronnes, comme nous le verrons par la suite. Les interpolation φ et $v_\parallel$, quant à elles, ne posent pas de problème, car le maillage est régulier dans ces dimensions et elles sont effectuées avec des interpolations 1D classiques.

La séparation des opérateurs entre les tuiles permet de minimiser les effets néfastes sur les performances du surcoût engendré par le maillage réduit sur l'interpolation poloïdale 2D. Sur les tuiles régulières, bords et centrales (qui ont un stockage régulier), l'interpolation

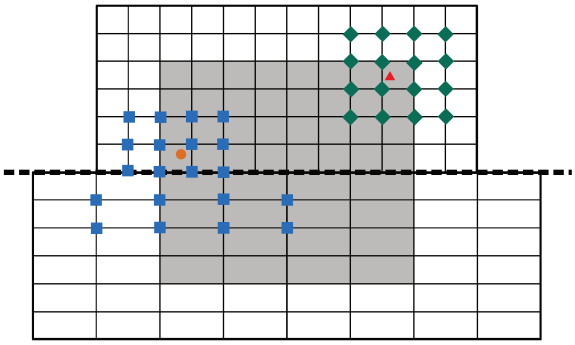


FIGURE 5.3 – Exemple d'interpolations sur une tuile jonction (représentation *géométrique* sans courbure). Les données locales sont en gris. L'interpolation du point ● implique les deux couronnes, celle du point ▲ est intégralement contenue dans la couronne supérieure.

de Lagrange implémentée est un simple produit tensoriel d'interpolations 1D pour les directions désirées parmi r, θ, φ et $v_{\parallel}$ (voir section 4.2.1 pour un exemple en $r{\times}\theta$). Pour les tuiles jonctions, la stratégie pour les interpolations combinant les dimensions r et θ est tirée des travaux développés en section 4.3.2 et est illustrée par la Figure 5.3. Si le *stencil* empiète sur les deux couronnes, l'interpolation dans la dimension θ est d'abord effectuée pour chaque indice r du *stencil* en choisissant le bon $\Delta\theta$. Puis les points ainsi reconstruits sont interpolés dans la direction r pour trouver la valeur finale comme le montre l'équation (4.9). Si le *stencil* est entièrement compris dans une couronne, une interpolation classique est effectuée avec le $\Delta\theta$ adéquat. Le découpage en tuiles permet de limiter les surcoûts aux seules tuiles jonctions.

Les performances de l'opérateur d'interpolation dépendent fortement de la façon dont sont stockées les données. L'interpolation des points situés au bord d'un sous-domaine nécessite également l'utilisation de zones tampons, que nous appellerons halos et qui nécessitent un stockage spécifique.

Stockage et halos

Similairement à la distribution du plan poloïdal pour la gyromoyenne distribuée dans le code GYSELA (voir section 3.1), la structuration en tuiles 5D des données et leur distribution (détaillée en section 5.1.2) imposent la prise en compte de halos (zones tampons) autour des tuiles pour certains opérateurs (l'interpolation et la gyromoyenne). Il est possible d'envisager plusieurs implémentations des halos en fonction de la façon dont seront stockées les données dans les tuiles. Ces halos sont également 5D, car des interpolations sont réalisées dans toutes les directions. Il serait toutefois possible de se limiter à un halo 2D + deux halos 1D, comme expliqué en section 7.2.

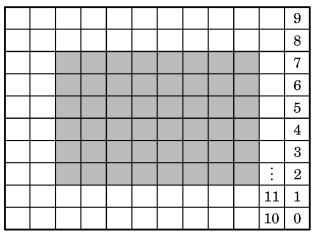
Sur un processus, les données peuvent être stockées de deux façons différentes : soit en utilisant un unique bloc mémoire pour l'ensemble des tuiles du processus (les tuiles d'un processus sont toutes mitoyennes dans notre stratégie de distribution), soit en utilisant autant de blocs qu'il y a de tuiles sur le processus. Ajoutons à cela les halos nécessaires à l'interpolation et à la gyromoyenne pour le stockage des données (nous nous focaliserons pour l'instant sur les halos de l'interpolation). Ceux-ci peuvent également être stockés de deux différentes manières : intégrés dans le stockage des données (par processus ou par tuiles) ou séparés dans un bloc mémoire dédié. Stocker les halos séparément améliore lé-

gèrement la localité spatiale pour les interpolations à l'intérieur d'un domaine (ou d'une tuile), mais la brise totalement pour les points nécessitant des données du halo (en plus d'avoir un système d'indexation séparé pour le halo). Il paraît plus opportun d'unifier le stockage des données et des halos. Considérons alors les différentes possibilités. Pour le stockage en un bloc par processus, le halo ne sera constitué que de données provenant de processus distants. Néanmoins, l'étendue du domaine d'un processus risque de fortement diminuer la localité spatiale. De plus, un système d'indexation complexe peut être nécessaire si le processus possède des tuiles jonctions (voir paragraphes suivants et commentaires des Figures 5.4). En effet, la présence de plusieurs couronnes dans le domaine et le stockage en une unique zone mémoire nécessiterait d'étendre le système d'indexation des tuiles jonctions à l'ensemble du domaine. Pour le stockage en un bloc par tuile, on peut également considérer que chaque tuile nécessite son propre halo constitué de données des tuiles voisines (données locales au processus pour les tuiles au cœur du domaine et données distantes pour les tuiles aux frontières du domaine). Présentant une bonne localité spatiale, le stockage des données tuile par tuile présente l'inconvénient de dupliquer des données sur un processus (dans les halos). Ainsi, il faut une fois encore choisir entre complexité calculatoire (système d'indexation complexe si stockage unique pour le processus et perte de localité spatiale) et empreinte mémoire (duplication des données locales dans les halos).

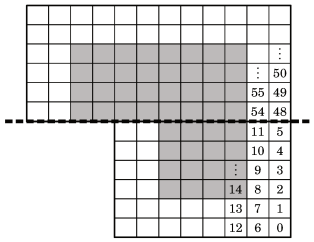
L'implémentation du stockage tuile par tuile étant plus simple et améliorant la lisibilité du code, le choix a été fait d'adapter cette approche. L'encapsulation du stockage dans le code (voir section 5.2) fait qu'il ne serait pas trop complexe de changer l'implémentation afin de pouvoir comparer les deux types de stockage. Il faudra veiller toutefois à trouver une taille de tuile optimale afin de minimiser la proportion prise par les halos dans la part d'empreinte mémoire totale tout en gardant un maximum de localité spatiale sachant que la taille des tuiles intervient également dans la granularité des tâches. Les données sont stockées dans l'ordre r, θ, φ puis $v_\parallel$, la dimension $v_\parallel$ étant la moins contrainte par les différents opérateurs (voir section 6.1) alors que les dimensions poloïdales sont plus complexes à gérer (voir section 6.1 et 6.2). Il aurait également été possible d'utiliser des courbes remplissantes (courbe de Hilbert, de Morton...) afin de stocker les données. Ces courbes présentent une bonne localité [54] des données mais ont un coût non négligeable. De plus, utiliser ces courbes sur les quatre dimensions réduirait la localité dans les dimensions les plus critiques : r et θ. Limitées à ces deux dimensions et étant donné la taille des tuiles dans ces dimensions, il n'est pas certain que ces courbes offrent un gain de localité.

Les Figures 5.4a et 5.4b donnent les représentations en mémoire de certaines tuiles telles qu'implantées dans GYSELA++. Les tuiles régulières (Figure 5.4a) ont une indexation classique simple. Les tuiles jonctions (Figure 5.4b) présentent une séparation (en pointillés) correspondant à la représentation des deux couronnes impliquées. Les tuiles bords et centrales ont un stockage similaire aux tuiles régulières, les halos sont simplement remplis différemment (conditions de bord pour les tuiles bords et rotation des données locales de la tuile pour la tuile centrale). À noter que les données de la tuile centrale sont recopiées intentionnellement dans le halo afin d'avoir une exécution plus rapide ; cela évite la rotation de π et le modulo $N_{\theta[0]}$ à chaque indice r négatif.

Afin de réaliser l'intégration en temps, il est nécessaire de posséder au moins deux versions de la fonction de distribution 5D : une au pas de temps t et une au pas de temps $t + \Delta t$ (notamment lors des advections). Cette duplication a été faite au niveau de l'encapsulation des données dans la tuile afin de limiter les duplications de métadonnées. Ainsi, le stockage des données dans une tuile a un jeu d'accesseurs pour chaque version de



(a) Représentation *logique* d'une tuile régulière.



(b) Représentation *logique* d'une tuile jonction.

FIGURE 5.4 – Représentations de la tranche poloïdale d'une tuile régulière et d'une tuile jonction. La zone grisée représente le domaine de la tuile et la zone blanche le halo (ici de taille 2). Les pointillés représentent le changement de couronne pour la tuile jonction.

la fonction et ils sont intervertis à la demande, lorsqu'une étape de calcul est terminée.

Le stockage des données tuile par tuile permet une exécution rapide et efficace de l'interpolation grâce à la localité spatiale accrue et à la présence des halos. Bien qu'ils mettent à disposition localement les données nécessaires, les halos engendrent un surcoût mémoire non négligeable qui peut être quantifié en fonction du nombre de tuiles, de leur taille et du degré des polynômes d'interpolation de Lagrange.

5.1.2 Distribution de données MPI

Distribution des fonctions de distributions

Une fois les tuiles créées, elles sont distribuées entre les processus MPI selon la configuration indiquée en paramètre. Contrairement au code GYSELA où la distribution ne pouvait se faire que dans trois dimensions à la fois (essentiellement $r\times\theta\times\mu$ et $\varphi\times v_\parallel\times\mu$), les données peuvent ici être distribuées dans toutes les dimensions. Cette stratégie augmente le potentiel de parallélisation du code. De plus, le nombre de processus dans les dimensions r et θ ne peut plus être choisi séparément ; le découpage sous forme de tuiles complexifiant le procédé, il a été décidé d'imposer la distribution des tuiles dans le plan poloïdal. Le nombre de processus MPI peut être sélectionné grâce aux paramètres $\texttt{Np}_{r\theta}$, $\texttt{Np}_{\varphi}$ et $\texttt{Np}_{v_\parallel}$ [3]. Comme pour l'algorithme de découpage des tuiles, différentes politiques de distribution de tuiles dans le plan poloïdal sont envisageables. Dans les dimensions φ et $v_\parallel$, la distribution se fait de façon homogène en respectant les paramètres d'entrée (en supposant qu'il y ait suffisamment de tuiles dans chaque dimension).

Par exemple, pour une configuration $\texttt{Np}_{r\theta} = 2$, $\texttt{Np}_{\varphi} = 2$ et $\texttt{Np}_{v_\parallel} = 2$ et pour un ensemble de tuiles dont le découpage poloïdal serait celui de la Figure 5.2 et le nombre de tuiles dans chaque dimension φ et $v_\parallel$ égal à quatre, les tuiles seraient réparties comme indiqué en Table 5.1. Une fois distribuées, les tuiles locales à chaque processus MPI sont allouées. L'ensemble des processus MPI dispose des métadonnées de toutes les tuiles afin de faciliter les communications et les éventuels échanges de tuiles entre processus (redistribution de données 3D pour Poisson ou possibilité de mettre en œuvre un équilibrage de charge dynamique).

3. $\texttt{Np}_{\mu}$ est également présent dans ce prototype mais n'est pour l'instant pas utilisé car $N_\mu = 1$. Cela rajoutera une dimension de parallélisation lorsque plusieurs points seront considérés en μ.

Rang MPI ($r\times\theta$, φ, $v_\parallel$)	Tuiles
0 (0,0,0)	0,1,3,5,6
1 (1,0,0)	2,4,7,8
2 (0,1,0)	9,10,12,14,15
3 (1,1,0)	11,13,16,17
4 (0,0,1)	18,19,21,23,24
5 (1,0,1)	20,22,25,26
6 (0,1,1)	27,28,30,32,33
7 (1,1,1)	29,31,34,35

TABLE 5.1 – Exemple de distribution de tuiles pour un découpage poloïdal tel que celui de la Figure 5.2 et l'ensemble de processus suivant : $\mathtt{Np}_{r\theta}$=2, $\mathtt{Np}_{\varphi}$=2, $\mathtt{Np}_{v_\parallel}$=2.

Distribution des champs 3D

En parallèle de la fonction de distribution, il est nécessaire de stocker plusieurs champs 3D, essentiellement le potentiel et les champs de déplacement des particules. Ces champs 3D sont également représentés sous forme de tuiles. Il n'y a qu'un unique jeu de tuiles 3D pour l'ensemble des champs, c'est-à-dire que chaque tuile possède autant de structures de stockage qu'il y a de champs. Les métadonnées ne sont ainsi pas dupliquées. Une fonction de correspondance permet de passer d'une tuile 5D à la tuile 3D associée. La distribution des tuiles 3D reste similaire à celle des tuiles 5D ; celles-ci sont donc répliquées sur chaque groupe de processus $v_\parallel$. Ainsi, chaque processus dispose des champs 3D sur son sous-domaine, évitant d'ajouter des communications à chaque utilisation des champs s'ils devaient être redistribués sur l'ensemble des processus. Cela permet de dissocier les *clusters* $v_\parallel$ (s'il y en a plusieurs) lors des opérations 3D afin de gagner un peu en asynchronicité. La Table 5.2 donne la distribution de tuiles 3D associée à la distribution des tuiles 5D présentée précédemment en Table 5.1.

Rang MPI ($r\times\theta$, φ, $v_\parallel$)	Tuiles
0 (0,0,0)	0,1,3,5,6
1 (1,0,0)	2,4,7,8
2 (0,1,0)	9,10,12,14,15
3 (1,1,0)	11,13,16,17
4 (0,0,1)	0,1,3,5,6
5 (1,0,1)	2,4,7,8
6 (0,1,1)	9,10,12,14,15
7 (1,1,1)	11,13,16,17

TABLE 5.2 – Distribution de tuiles 3D pour une distribution de tuiles 5D telle que présenté en table 5.1 et pour l'ensemble de processus suivant : $\mathtt{Np}_{r\theta}$=2, $\mathtt{Np}_{\varphi}$=2, $\mathtt{Np}_{v_\parallel}$=2.

En plus de la distribution de données standard en 3D, il existe actuellement des distributions de données alternatives qui, comme dans GYSELA, permettent d'avoir localement certaines dimensions en entier. À terme, l'objectif est de se passer de ces distributions alternatives en adaptant les opérateurs à une distribution de données dans toutes les dimensions, mais elles restent pour l'instant nécessaires pour réutiliser certains opérateurs

existants (essentiellement l'opérateur de Poisson qui est le plus complexe, voir section 6.2). L'opération de redistribution des données associée (voir section 5.2.1) est coûteuse en temps de calcul du fait des quantités de données impliquées dans les communications.

5.2 Structure logique du code

Le code GYSELA++ implémente le modèle 4D *drift-kinetic* (voir section 1.1.5) avec un moment magnétique $\mu = 0$. Ainsi, l'opérateur de gyromoyenne est réduit à l'identité (voir section 2.1.1) et n'est pour l'instant pas nécessaire. Une simulation 4D *drift-kinetic* calcule dans un premier temps le potentiel électromagnétique d'une fonction de distribution de particules grâce à l'opérateur de Poisson (section 6.2). Ce potentiel est ensuite transformé pour calculer des champs de déplacement et faire évoluer la distribution de particules avec l'opérateur d'advection de Vlasov (section 6.1). Cela génère une nouvelle fonction de distribution dont le potentiel est recalculé et ainsi de suite. Poisson et Vlasov sont les deux opérateurs centraux du code GYSELA++. Une phase de diagnostic durant laquelle des données utiles aux physiciens sont extraites est également nécessaire. La parallélisation s'effectue en MPI pour les communications inter-processus (section 5.2.1) et des tâches OpenMP sont utilisées pour la parallélisation multicœur (section 5.2.2). Les communications MPI sont effectuées par un *thread* OpenMP dédié afin d'assurer l'asynchronicité. De plus, le déroulement de la simulation repose sur le *thread ordonnanceur* qui génère les tâches de calcul. Cet *thread ordonnanceur* génère les tâches de calcul pour les *threads de calcul* et déclenche les phases de communication avec le *thread communiquant*. La Figure 5.5 illustre le déroulement d'un pas de temps d'une simulation de GYSELA++ pour un processus avec quatre *threads*. Le *thread* ordonnanceur T_1 distribue le calcul aux *threads* T_2 et T_3 et délègue la gestion des communications au *thread* T_0.

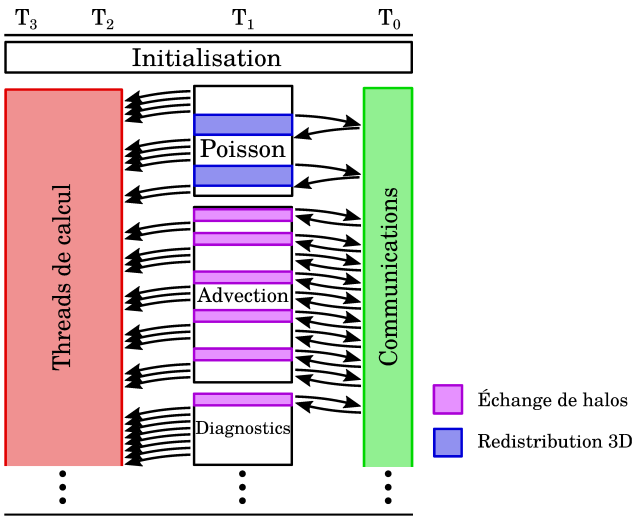


FIGURE 5.5 – Diagramme simplifié de déroulement du code GYSELA++ avec quatre *threads*.

Le C++ a été utilisé pour certaines de ses fonctionnalités afin de simplifier le développement. Notamment, le système de classes permet d'organiser de façon lisible les différentes

parties du programme et facilite la mise en place d'une encapsulation des mécanismes internes (structuration en tuiles, accesseurs...) qui permettra à des physiciens non expert du code GYSELA++ de développer de nouveaux opérateurs. La Figure 5.6 présente les principales classes de GYSELA++ :

- la classe `Gysela` est la classe principale : elle implémente les opérateurs ainsi que la routine de communication principale ;
- la classe `Distrib` encapsule toute la partie technique des communications MPI (lancement des requêtes préétablies, traduction des *tags*, routines de redistribution...) et stocke les différentes distributions de données (voir section 5.2.1 – «Communications inter-processus») ;
- la classe `Mesh` stocke l'ensemble des données relatives à la géométrie physique (dimensions) et logique (maillage) du problème (voir section 5.1) : elle implémente les traductions de coordonnées logiques vers géométriques et inversement ;
- la classe `Omp_parallelization` implémente tout ce qui a attrait à la parallélisation multicœurs, notamment le système de communication entre *threads* (voir section 5.2.1 – «Système de notifications intra-processus») ;
- les classes `Tile5d_map` et `Tile3d_map` stockent l'ensemble des tuiles ainsi qu'un réseau de voisinage des tuiles pour faciliter les communications ;
- les classes `Tile5d` et `Tile3d` contiennent les métadonnées des tuiles ainsi qu'un pointeur vers les différentes fonctions de distribution ou les différents champs lorsqu'ils sont alloués (tuiles locales) : elles implémentent les opérateurs d'interpolation, essentiels dans la simulation et stockent les pré-requêtes pour l'échange de halos (voir section 5.2.1) ;
- les classes `Function5d` et `Field3d` implémentent le système de stockage présenté précédemment (voir section 5.1.1) ; les données sont stockées de manière linéaire dans l'ordre $r \times \theta \times \varphi \times v_{\|}$ et des accesseurs sont mis à disposition.

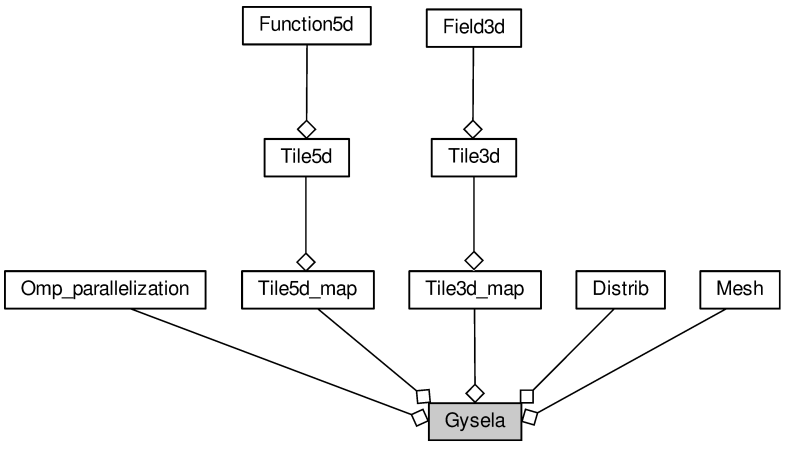


FIGURE 5.6 – Diagramme de classe principal de GYSELA++. Les losanges indiquent une association.

Les différentes implémentations des classes `Tile5d` et `Tile3d`, correspondant à chaque type de tuiles présentées en section 5.1.1, illustrent le rôle important de l'hérédité du C++ dans ce code. La Figure 5.7 présente le graphe d'héritage de la classe `Tuile`.

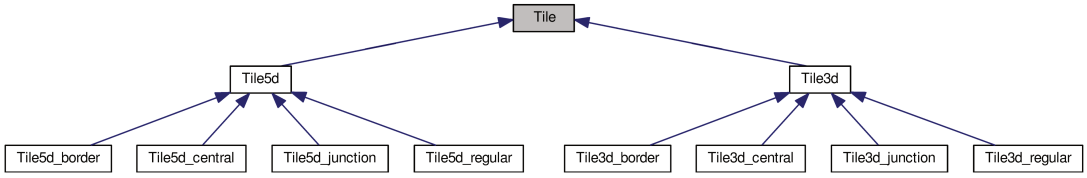


FIGURE 5.7 – Diagramme d'héritage des classes tuiles de GYSELA++. Les flèches indiquent l'hérédité.

5.2.1 Modèle de communication

Le code GYSELA++ utilise un paradigme de parallélisation hybride OpenMP pour le calcul multicœur (intra-processus) et MPI pour le calcul distribué (inter-processus). Il se base sur un modèle de communication asynchrone afin de minimiser les phases durant lesquelles des cœurs sont inoccupés, en attente des données provenant d'autres processus MPI. La norme MPI spécifie des primitives de communication non bloquantes. Implémentées dans les bibliothèques les plus usitées (OpenMPI, IntelMPI ou encore MVAPich), ces primitives ne sont toutefois pas réellement asynchrones (voir section 3.2.2) ; en pratique, les communications n'avancent que lorsque le programme réalise des appels à une des fonctions de la bibliothèque. Il existe toutefois des implémentations dans lesquelles les communications non bloquantes sont également réellement asynchrones. C'est le cas, par exemple, de la bibliothèque MPI New Madeleine [21] développée par l'équipe Taadam à Inria Bordeaux. Pour des raisons de portabilité du futur code, nous avons choisi de ne pas utiliser de bibliothèques absentes des systèmes HPC utilisés en production pour l'implémentation de ce prototype. Il a donc été décidé de mettre en œuvre nous-mêmes la progression asynchrone des communications par le biais d'un *thread* OpenMP dédié. Dans la zone parallèle, le *thread* communiquant parcourt une routine qui reste à l'écoute de l'ensemble des communications entrantes jusqu'à la fin de la zone parallèle (qui correspond à la fin de la simulation). Afin de permettre au processus d'envoyer des données, il faut également que le *thread* communiquant puisse être informé des données à transmettre. Pour cela un système de notification entre *threads* d'un même processus est mis en place.

Communication inter-processus

Les communications entrantes et sortantes sont uniquement réalisées par le *thread* communiquant. Celui-ci est défini comme étant le *thread* maître (*thread* 0) car c'est la situation qui est la mieux supportée par la plupart des bibliothèques MPI. Pendant toute la durée de la simulation, la routine de communication reste à l'écoute des communications MPI entrantes et des communications sortantes demandées par les autres *threads* (détails dans la sous-section suivante). Elle consiste essentiellement en un appel répété à la primitive `MPI_Iprobe` et à la routine de parcours des notifications `notification_probe`. Plusieurs types de communications peuvent être déclenchées dont : les communications invariantes dont les paramètres et le nombre d'appels ne changent pas d'une itération temporelle à l'autre (échange de halos, redistribution) et les communications ponctuelles dont les paramètres et le nombre par itération varie (interpolations).

Les échanges de halos sont optimisés par un système de requêtes préétablies. Le maillage, le découpage des tuiles ainsi que leur distribution étant fixés au début du programme, les halos et les communications nécessaires pour leur mise à jour le sont également. Ces *pré-*

requêtes consistent en un triplet (destinataire, taille de bloc, indice de départ), créé à l'initialisation par chaque tuile pour chaque halo auquel elle contribue. Ainsi, lors d'une phase d'échange de halos, le processus n'a plus qu'à parcourir, pour chaque tuile, la liste de ces *pré-requêtes* et à les lancer par une communication non bloquante par le biais de la classe `Distrib` (`MPI_Isend`). Une même pré-requête (par exemple mise à jour du halo $r\times\theta$ supérieur droit, de la tuile t) peut servir pour mettre à jour différents champs 3D ou fonctions 5D. De plus, il est possible d'effectuer la mise à jour des halos de différents champs simultanément. À noter que si des mises à jour de halos s'effectuent vers des tuiles locales, des pré-requêtes sont également créées, mais l'appel MPI est remplacé par une recopie du bloc de données dans la tuile de destination.

Le procédé pour les redistributions 3D est similaire, mais n'utilise pas le système de pré-requêtes car les différentes distributions de données, calculées à l'initialisation et stockées dans la classe `Distrib`, font office de requêtes. Une redistribution se déroule donc comme suit :

— allocation de la mémoire des tuiles locales dans la distribution cible (simple appel de type `tile.allocate()` qui est ignoré si la tuile est déjà allouée) ;
— parcours des tuiles locales de la distribution cible et postage des réceptions asynchrones de leurs données ;
— parcours des tuiles locales de l'ancienne distribution et envoi asynchrone de leurs données vers leur nouveau processus ;
— mise à jour des variables locales afférentes et mise à jour des pré-requêtes des halos avec les nouvelles localisations des tuiles ;
— terminaison des communications sortantes et déallocation des anciennes tuiles qui ne sont plus présentes dans la distribution cible ;
— terminaison des communications entrantes.

Les redistributions sont une exception dans les communications, car elles ne sont pas traitées dans la boucle principale, mais de façon synchrone dans une routine dédiée par le *thread* communiquant. Une implémentation totalement asynchrone dans la boucle principale aurait été possible, mais plus complexe (voir section 7.2). Ces redistributions ont été mises en place afin de réutiliser facilement une version existante de l'opérateur de Poisson.

Les interpolations à distance consistent à envoyer les coordonnées d'un point, ainsi que le champ ou la fonction pour lequel la valeur en ce point est désirée, au processus distant où il est situé. Les *threads* de calcul soumettent d'abord des requêtes d'interpolation au *thread* communiquant *via* un système de notification interne (détaillé en sous-section suivante). Le *thread* communiquant prend acte de chaque requête et l'expédie au *thread* communiquant du processus disposant de la donnée par une communication asynchrone. Celui-ci reçoit cette demande d'interpolation, l'effectue et la renvoie au processus initial par une communication asynchrone. À la réception, le processus initial transmet la valeur interpolée au *thread* de calcul correspondant et le débloque. Cette implémentation point à point de l'interpolation souffre de plusieurs problèmes :

— une interpolation par *thread* à la fois et donc un nombre conséquent de communications ;
— le blocage fréquent des *threads* de calcul en attente du retour d'une valeur interpolée ;
— de fait, un asynchronisme limité et un recouvrement non optimal.

Le mécanisme d'interpolation à distance fait l'objet de plusieurs optimisations dans le cadre de l'opérateur d'advection (section 6.1.2).

D'autres types de communications ponctuelles servant à notifier et synchroniser des

Algorithme 8 : Algorithme global de la routine de communication.

```
Entrée : Nombre de processus Nproc
tant que "condition d'arrêt" faire
    tant que notif = {∅} et comm = {∅} faire
1       MPI_Iprobe(comm)            /* Scan des communications entrantes */
2       notification_probe(notif)   /* Scan des notifications internes */
3   pour c ∈ comm faire
        si c = COMM_INTERP alors
            c2 ← interpolate(c.data)
            MPI_Send(c2)
        si c = COMM_INTERP_RESPONSE alors
            notify_thread_interp(c.data)
        si c = COMM_HALO_UPDATE alors
            store_halo(c.location, c.data)
        ⋮
4   pour n ∈ notif faire
        si n = NOTIF_STAGE alors        /* Halos, transpo., réduc.... */
            start_corresponding_set_of_comms(n)
        si n = NOTIF_INTERP_REQUIRED alors
            MPI_Isend(n.data)
            MPI_Irecv(n.data_destination)
        ⋮
```

changements de phases de l'algorithme entre processus sont également utilisés. Pour résumer, l'algorithme 8 récapitule le déroulement global de la routine de communication du code Gysela++. La partie notifications est détaillée dans la sous-section suivante. Les communications et notifications dont la gestion est simple ne sont pas représentées pour faciliter la lisibilité (changement d'étape, fin de la simulation...). À noter que l'appel répété à la primitive `MPI_Iprobe` permet de faire avancer les communications MPI, car celles-ci n'évoluent que lors d'un appel à une fonction de la bibliothèque MPI.

Système de notifications intra-processus

Afin que le recouvrement calcul – communication et le modèle de calcul par tâches (section 5.2.2) fonctionnent correctement, il est nécessaire de mettre en place un système de communication entre les différents types de *threads* : le *thread* communiquant, le *thread* ordonnanceur et les *threads* de calcul. Celui-ci doit permettre une transmission efficace des données et informations tout en générant un minimum de perturbations (pas de blocage des *threads* et peu d'*overhead*).

Ce système est implémenté au sein de la classe `Omp_parallelization` et dispose d'un

certain nombre de *buffers*, dont le nombre et la taille sont fixés à l'avance, pour faire transiter les informations entre les *threads*. Son utilité principale est de permettre au *thread* ordonnanceur de notifier au *thread* communiquant les différentes phases de la simulation ainsi que de permettre aux *threads* de calcul de demander des interpolations distantes au *thread* communiquant. Différentes demandes peuvent être signifiées au *thread* communiquant :

— entrée dans une phase de communication qui prend plusieurs paramètres (échange de halos, redistribution ou réduction puis champ ou fonction) et déclenche la phase de communication associée ;
— changement d'étape qui synchronise tous les processus, typiquement lors du passage du pas de temps t au temps $t + \Delta t$ ou entre les différentes advections (échange des deux fonctions de distributions) ;
— signal de fin qui stipule que les calculs de la simulation sont terminés pour le processus local ; le *thread* communiquant envoie alors un message de fin à l'ensemble des processus et attend un message similaire de leur part pour s'arrêter tout en continuant de traiter les requêtes qui lui sont adressées.

Ces notifications sont effectuées par le *thread* ordonnanceur et sont bloquantes pour lui, sauf pour l'échange de halos qui dispose d'une notification de début et d'une notification de terminaison. Il est donc possible au *thread* ordonnanceur d'effectuer des opérations et de continuer à distribuer des tâches de calcul lors de la mise à jour de halos jusqu'à ce que leurs données soient bloquantes.

Concernant l'interpolation, les *threads* de calcul disposent chacun d'un *buffer* leur permettant d'enregistrer une demande d'interpolation auprès du *thread* communiquant. Ce dernier "surveille" ces *buffers* dans la routine `notification_probe` permettant de vérifier la présence de notifications (ligne 2 de l'algorithme 8). Si des demandes sont présentes, le mécanisme de communication présenté précédemment est enclenché jusqu'à ce que le *thread* communiquant retourne la valeur interpolée aux *threads* de calcul et les débloque. Une implémentation améliorée est proposée en section 6.1.2.

5.2.2 Modèle de programmation par tâches

Gysela++ utilise un modèle de programmation par tâches (voir section 1.2.4), c'est-à-dire que l'ensemble des calculs effectués dans la simulation seront encapsulés dans des tâches et que leur exécution sera gérée par un ordonnanceur en fonction de leurs dépendances. Ce modèle doit permettre d'améliorer l'occupation des cœurs en optimisant l'équilibrage pour minimiser les temps durant lesquels ceux-ci sont inactifs en attente de travail.

Le modèle de tâches OpenMP est moins intrusif que celui de StarPU. De plus, il permet de tester différents ordonnanceurs puisque le *runtime* est remplaçable sans avoir à changer de code, simplement en changeant de bibliothèque. Il a donc été préféré pour ce prototype. Ainsi, les ordonnanceurs OpenMP de GNU (gcc) et d'Intel (KOMP) sont comparés dans l'étude présentée en section 7.1.1. Bien qu'en pleine expansion, le modèle de tâches d'OpenMP présente toutefois quelques lacunes. La gestion des dépendances est par exemple limitée, car OpenMP ne permet de placer des dépendances que sur des variables simples et des sections de tableaux. De plus, les tâches ne sont pas étiquetables, ce qui empêche de créer des groupes de dépendance. Ainsi, les dépendances de données sont faites tuile par tuile (les données d'une tuile ne sont pas séparables pour les dépendances). Il est toutefois possible de séparer le halo des données locales d'une tuile en utilisant par exemple la première case du halo comme étiquette de dépendance pour le halo et la première case

du domaine de la tuile pour les données locales. Il est possible de spécifier des dépendances en entrée (lecture) et en sortie (écriture) et dans GYSELA++, les *buffers* à t et $t + \Delta t$ peuvent être spécifiés séparément. Le coût de gestion des tâches et surtout de gestion des dépendances ne doit pas être négligé comme nous le verrons en section 7.1.1.

Algorithme 9 : Pseudo-code d'un extrait de l'opérateur d'advection exécuté par le *thread* ordonnanceur.

```
   pour tile ∈ tile3d_map faire                          /* Champs de vitesse */
       data_in ← tile.potential
       data_out ← tile.velocity
1      OMP task depend(in:data_in) depend(out:data_out)
           tile.compute_displacement_field()
2  notify(HALO_5D_VPAR_END, DISTRIBUTION_FUNCTION)
   pour tile5d ∈ tile5d_map faire                        /* Advection v_|| */
       tile3d ← corresponding_3d_tile(tile5d)
       data_in ← tile3d.velocity
       data_out ← tile5d.distrib
3      OMP task depend(in:data_in) depend(out:data_out)
           tile.vpar_advection()
4  notify(HALO_5D_PHI_START, DISTRIBUTION_FUNCTION)
5  notify(HALO_5D_PHI_END, DISTRIBUTION_FUNCTION)
   pour tile5d ∈ tile5d_map faire                        /* Advection φ */
       ...
   ...
```

Ainsi, l'ensemble des calculs intensifs de GYSELA++ sont soumis à l'ordonnanceur et effectués sous forme de tâches par les *threads* de calcul. L'algorithme 9 donne un exemple des actions du *thread* ordonnanceur pour l'opérateur d'advection. Une tâche par tuile est créée aux lignes 1 et 3. La granularité des tâches peut être modifiée (plusieurs tâches par tuile ou plusieurs tuiles par tâche) et fait l'objet d'une étude en section 7.1.1. Les lignes 2, 4 et 5 notifient le *thread* communiquant des phases de communication : aux lignes 2 et 5, attente de la fin de la transmission des halos en $v_\parallel$ et φ et à la ligne 4, déclenchement de la communication des halos en φ[4]. Différents aspects de l'implémentation du modèle de programmation par tâches sont évalués en section 7.1.1, dont la granularité des tâches et la comparaison de différents ordonnanceurs (OpenMP, Intel et StarPU).

Pour conclure, la Figure 5.8 récapitule l'ensemble des mécanismes mis en œuvre dans le code GYSELA++ et présentés dans ce chapitre en se basant sur un extrait de l'opérateur d'advection. La Figure présente le déroulement d'une simulation à deux processus, chacun possédant 4 *threads* (le *thread* T_0 est le *thread* communiquant en vert, T_1 le *thread* ordonnanceur en blanc et T_2 et T_3 les *threads* de calcul en rouge). Les notifications et actions initiées par le *thread* ordonnanceur sont indiquées par des flèches en trait plein. Les notifications envoyées par les *threads* de calcul sont représentées par des flèches en

4. À noter qu'il serait possible de gagner en recouvrement si les communications de halos pouvaient être notifiées tuile par tuile et non plus seulement pour le processus entier, ce qui est possible avec un effort de développement modéré.

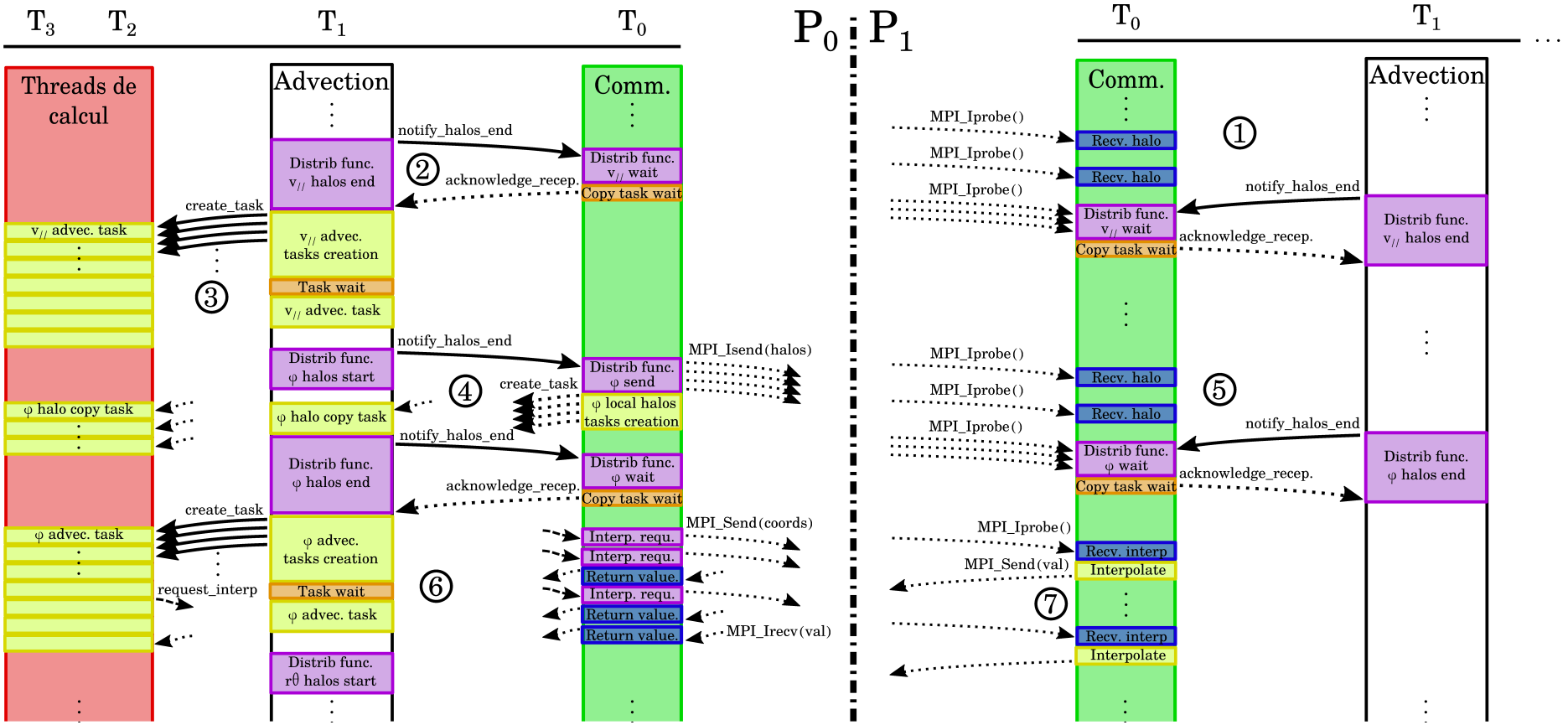


FIGURE 5.8 – Détail du déroulement d'une simulation à deux processus avec quatre *threads* sur un extrait de l'opérateur d'advection. Les blocs jaunes représentent des phases de création et d'exécution de tâches, les blocs bleus des phases réception de communications, les blocs oranges des phases d'attente de complétion de tâches et les blocs violets des phases de notifications internes.

trait “tirets”. Les notifications, actions et communications initiées par le *thread* communiquant sont en trait pointillé. Des numéros indiquent différentes étapes clefs du code de simulation. Pour des raisons de lisibilité, celles-ci sont réparties sur les deux processus ; dans les faits, chaque processus passe par l'ensemble des états représentés en Figure 5.8. L'étape 1 représente l'état standard du *thread* communiquant : celui-ci est à l'écoute de communications entrantes (`MPI_Iprobe`), ici des réceptions de halos en bleu, et en attente de notifications internes. Une notification de terminaison des échanges de halos en $v_{\|}$ lui est envoyée par le *thread* ordonnanceur à l'étape 2 en prévision de l'advection en $v_{\|}$. Le *thread* communiquant passe donc en terminaison prioritaire de ces communications et vérifie que toutes les tâches de recopie locale des halos (`Copy task wait`) sont terminées avant de confirmer la terminaison. L'étape 3 consiste en la création des tâches d'advection $v_{\|}$ par le *thread* ordonnanceur et leur exécution par les *threads* de calcul et le *thread* ordonnanceur. Leur terminaison est assurée avant l'étape 4 de mise à jour des halos en φ où, après avoir été notifié par le *thread* ordonnanceur, le *thread* communiquant envoie toutes ses mises à jour de halos distants et crée des tâches de mise à jour des halos locaux (effectuées par des *thread* inactifs) avant de rendre la main. Après une nouvelle étape 5 de réception de halos et de terminaison des échanges de halos, l'advection en φ est lancée par le *thread* ordonnanceur à l'étape 6. Des interpolations distantes sont demandées par les *threads* de calcul au *thread* communiquant qui les relaie au bon processus. Ce dernier reçoit les requêtes en 7, effectue les interpolations et renvoie les valeurs demandées au processus initial qui les réceptionne et les transmet au *thread* de calcul correspondant. Puis l'advection en φ se termine et la simulation continue, ainsi de suite.

Le code Gysela++ propose différentes solutions afin de pallier les problèmes rencontrés par Gysela lors du passage à l'échelle. Le volume de données à traiter est réduit significativement avec l'introduction du maillage réduit, diminuant à la fois l'empreinte mémoire, la quantité de calculs et le volume de communication. Un modèle de programmation par tâches est mis en place afin d'optimiser l'occupation des cœurs de calcul. Le déploiement de ce modèle est soutenu par un stockage de données structuré ajustable. L'usage de communications asynchrones vise à réduire l'impact des communications sur le temps de calcul en superposant calculs et communications. Les deux opérateurs principaux, Poisson et Vlasov, et leur implémentation font l'objet d'une présentation détaillée au chapitre 6. L'impact sur les performances des différentes solutions d'implémentation et d'optimisation est évalué au chapitre 7. Une évaluation des différents coûts mémoire (halos, *buffers* de communication...) est également proposée.

Chapitre 6

Opérateurs pour un modèle *drift-kinetic*

Sommaire

Le modèle 4D *drift-kinetic* à $\mu = 0$ (voir section 1.1.5) est un cas particulier du modèle gyrocinétique dont la mise en œuvre, plus concise, permet de tester aisément différentes solutions d'implémentation, que ce soit pour les mécanismes de parallélisation ou pour les deux opérateurs fondamentaux : l'opérateur d'advection de Vlasov et l'opérateur de Poisson. Dans ce modèle, lorsque l'on considère la géométrie slab (cylindre périodique dans la longueur), le champ magnétique est supposé uniforme et parallèle et l'opérateur de gyromoyenne est réduit à l'identité. Les équations s'en retrouvent grandement simplifiées et permettent de tester de nouveaux schémas numériques sans avoir à mettre en œuvre l'opérateur de gyromoyenne. Les opérateurs de Vlasov et de Poisson, présentés ci-après, sont basés sur les équations posées dans l'article [36] qui présente les fondements du code Gysela. Les sections suivantes introduisent ces deux opérateurs ainsi que leur implémentation et adaptation au modèle de tâches et au maillage réduit avant de détailler les algorithmes et mises en œuvre originales qui ont été développées. La section 6.1 se concentre sur l'opérateur d'advection et la section 6.2 sur l'opérateur de Poisson.

6.1 Opérateur d'advection

L'opérateur d'advection permet de faire évoluer la fonction de distribution des particules en fonction des forces qui lui sont appliquées. Pour rappel, l'équation gyrocinétique générale de l'advection de Vlasov est donnée par :

$$\frac{\partial \bar{f}}{\partial t} + \mathbf{v}_{\mathrm{cg}} \cdot \nabla_{\perp} \bar{f} + v_{\parallel} \frac{\partial \bar{f}}{\partial \varphi} + \dot{v_{\parallel}} \frac{\partial \bar{f}}{\partial v_{\parallel}} = 0$$

où $\mathbf{v}_{\text{cg}}$ désigne la vitesse des centres-guides, et $\nabla_\perp = (\frac{\partial \cdot}{\partial r}, \frac{1}{r}\frac{\partial \cdot}{\partial \theta})$ représente le gradient dans le plan poloïdal. Les différentes hypothèses du modèle 4D *drift-kinetic* permettent d'en simplifier l'expression.

6.1.1 Algorithme et implémentation

Dans le cadre gyrocinétique, l'équation d'advection de Vlasov peut être reformulée, sous certaines conditions, de façon à être exprimée en un point (**x**,**v**) de l'espace des phases par les deux équations (6.1) et (6.2) avec $\mathbf{x} = (x_r, x_\theta, x_\varphi)$. Leur calcul ne sera pas explicité ici et le lecteur peut se référer à [36] pour plus de détails. Ces équations sont quasiment celles utilisées dans le code GYSELA (ici l'hypothèse $\mathbf{B} = \mathbf{B}^*$ est faite).

$$\frac{\mathrm{d}x_i}{\mathrm{d}t} = (v_\parallel \mathbf{b} + \mathbf{v}_{E\times B} + \mathbf{v}_D) \cdot \nabla x_i \tag{6.1}$$

$$m \cdot \frac{\mathrm{d}v_\parallel}{\mathrm{d}t} = -\mu \mathbf{b} \cdot \nabla B - q\mathbf{b} \cdot \nabla \Phi + \frac{m \cdot v_\parallel}{B} \mathbf{v}_{E\times B} \cdot \nabla B \tag{6.2}$$

$B = ||\mathbf{B}||$ est la norme du champ magnétique, $\mathbf{b} = \frac{\mathbf{B}}{B}$ représente le vecteur unité du champ magnétique et Φ le potentiel électrostatique. L'influence du champ électromagnétique sur le déplacement des particules est donnée par le terme $\mathbf{v}_{E\times B}$ et la vitesse de dérive, $\mathbf{v}_D$, représente l'autre composante perpendiculaire de la vitesse (due à la courbure du champ magnétique et au gradient de sa norme). Les hypothèses prises dans le cadre du modèle 4D *drift-kinetic* permettent de simplifier ces expressions. Dans un premier temps, la géométrie slab (géométrie de type cylindrique plutôt que torique) supprime la courbure géométrique du maillage. D'autre part, le champ magnétique est considéré comme stationnaire. Ainsi la vitesse de dérive est nulle et l'équation (6.1) est simplifiée en (6.3). On considère également que le champ magnétique est uniquement parallèle et n'a pas de composante en r et en θ ($\mathbf{B} = B_\varphi \mathbf{e}_\varphi$). $\mathbf{v}_{E\times B}$ n'ayant pas de composante parallèle par définition, son produit scalaire avec ∇B est nul. D'autre part, on a $\mu = 0$ par hypothèse. L'équation (6.2) se simplifie donc en (6.4).

$$\frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t} = (v_\parallel \mathbf{b} + \mathbf{v}_{E\times B}) \cdot \nabla \mathbf{x} \tag{6.3}$$

$$\frac{\mathrm{d}v_\parallel}{\mathrm{d}t} = -\frac{q}{m} \mathbf{b} \cdot \nabla \Phi \tag{6.4}$$

Dans GYSELA et GYSELA++, un algorithme avec *splitting* de Strang (présenté en section 1.3.1) est mis en place. Il consiste à diviser l'advection 4D en plusieurs advections 1D (deux en φ et deux en $v_\parallel$) et une advection 2D (en $r\times\theta$). Ce découpage spatial nécessite donc de connaître dimension par dimension les champs de déplacement des particules. À partir des équations (6.3) et (6.4) et des hypothèses présentées précédemment, ceux-ci sont donnés pour un point (r,θ,φ,$v_\parallel$,μ=0) par les équations (6.5).

$$\begin{aligned}
\frac{\mathrm{d}r}{\mathrm{d}t} &= -\frac{1}{rB}\frac{\partial \Phi}{\partial \theta} \\
\frac{\mathrm{d}\theta}{\mathrm{d}t} &= \frac{1}{rB}\frac{\partial \Phi}{\partial r} \\
\frac{\mathrm{d}\varphi}{\mathrm{d}t} &= v_\parallel \\
\frac{\mathrm{d}v_\parallel}{\mathrm{d}t} &= -\frac{\partial \Phi}{\partial \varphi}
\end{aligned} \tag{6.5}$$

L'algorithme 10 présente l'implémentation globale de l'opérateur d'advection avec *splitting* de Strang dans GYSELA++. Pour chaque partie de l'opérateur, une tâche est générée pour chaque tuile. Les dépendances sont positionnées en entrée et en sortie pour chaque tuile sur les champs utilisés comme expliqué en section 5.2.2 et comme montré par l'algorithme 9. Pour l'échange des halos 5D en $v_\parallel$, seul le début des communications se fait des la fin de l'advection précédente puisque seul l'opérateur d'advection modifie la fonction de distribution. Le calcul des champs de déplacement est effectué en partie en amont (premières lignes de l'algorithme 10), car bien que simple pour l'instant, il est relativement complexe dans le modèle gyrocinétique. Les champs de déplacement sont pour l'instant essentiellement 3D et seul le déplacement dans la dimension φ dépend de $v_\parallel$. Le calcul de ces champs se fait donc en deux temps : une première partie 3D, calculée dans les dimensions spatiales en amont et stockée, et une seconde partie ajoutant la composante en vitesse parallèle.

Algorithme 10 : Pseudo-code l'opérateur d'advection avec *splitting* de Strang.

```
pour tile3d ∈ tile3d_map faire                          /* Champs de vitesse */
    OMP task depend(in:...) depend(out:...)
        tile3d.compute_displacement_field()
notify(HALO_5D_VPAR_END, DISTRIB_FUNCTION)   /* Terminaison halos v_∥ */
pour tile5d ∈ tile5d_map faire                          /* Advection v_∥ */
    OMP task depend(in:...) depend(out:...)
        tile.vpar_advection(Δt/2)
notify(HALO_5D_PHI_DO, DISTRIB_FUNCTION)          /* Échange halos φ */
pour tile5d ∈ tile5d_map faire                           /* Advection φ */
    OMP task depend(in:...) depend(out:...)
        tile.phi_advection(Δt/2)
notify(HALO_5D_RTHETA_DO, DISTRIB_FUNCTION)     /* Échange halos r×θ */
pour tile5d ∈ tile5d_map faire                        /* Advection r×θ */
    OMP task depend(in:...) depend(out:...)
        tile.rtheta_advection(Δt)
notify(HALO_5D_PHI_DO, DISTRIB_FUNCTION)          /* Échange halos φ */
pour tile5d ∈ tile5d_map faire                           /* Advection φ */
    OMP task depend(in:...) depend(out:...)
        tile.phi_advection(Δt/2)
notify(HALO_5D_VPAR_DO, DISTRIB_FUNCTION)         /* Échange halos v_∥ */
pour tile5d ∈ tile5d_map faire                          /* Advection v_∥ */
    OMP task depend(in:...) depend(out:...)
        tile.vpar_advection(Δt/2)
notify(HALO_5D_VPAR_START, DISTRIB_FUNCTION)        /* Début halos v_∥ */
```

Chaque advection 1D, φ et $v_\parallel$, applique les équations d'advection (6.5) associées à chaque point de chaque tuile en utilisant une discrétisation du premier ordre : le déplace-

ment est donné par le champ de déplacement multiplié par le pas de temps (par exemple $\mathrm{d}v_{\parallel} = -\frac{\partial \Phi}{\partial \varphi} \cdot \Delta t$). L'advection 2D $r{\times}\theta$ nécessite un calcul plus précis du fait des déplacements rapides et complexes des particules dans la direction perpendiculaire. De plus, ces déplacements dans le plan poloïdal sont bien plus incurvés que dans la direction parallèle où ils sont essentiellement parallèles aux lignes de champ.

Advection $r{\times}\theta$

Il est nécessaire d'utiliser un schéma numérique d'ordre deux en temps pour l'advection dans le plan poloïdal afin de conserver une précision du même ordre pour l'ensemble du schéma d'advection avec *splitting* de Strang [68]. Deux solutions sont alors possibles : une approche utilisant les dérivées temporelles des champs de déplacement, dite de Taylor, et une approche en deux temps utilisant un double déplacement de pas de temps $\frac{\Delta t}{2}$ puis Δt, dite de Newton.

L'approche de Taylor consiste simplement à remplacer la discrétisation du premier ordre pour les équations r et θ de (6.5) par un développement de Taylor du second ordre combinant les deux dimensions [35]. Ainsi, en notant $u_r = -\frac{1}{rB} \cdot \frac{\partial \Phi}{\partial \theta}$ et $u_\theta = \frac{1}{rB} \cdot \frac{\partial \Phi}{\partial r}$, les champs de déplacement dans ces deux dimensions, les déplacements associés sont donnés par l'équation (6.7) :

$$\begin{aligned} \mathrm{d}r =& u_r \cdot \Delta t + (u_r \frac{\partial u_r}{\partial r} + u_\theta \frac{\partial u_r}{\partial \theta}) \cdot \frac{\Delta t^2}{2} + O(\Delta t^3) \\ \mathrm{d}\theta =& u_\theta \cdot \Delta t + (u_r \frac{\partial u_\theta}{\partial r} + u_\theta \frac{\partial u_\theta}{\partial \theta}) \cdot \frac{\Delta t^2}{2} + O(\Delta t^3). \end{aligned} \tag{6.6}$$

Cette méthode nécessite de calculer les dérivées du champ de déplacement $r{\times}\theta$ en tout points du domaine. Pour Gysela++, cela signifie avoir des halos pour les champs de déplacement, donc également pour les dérivées du potentiel avec lesquels ils sont calculés. Nous verrons que cela ne pose pas de problème avec l'opérateur de Poisson actuel, puisque les dérivées peuvent être calculées avec un minimum de communications. Le champ magnétique est supposé invariant en φ et consiste donc en un champ vectoriel 2D poloïdal. Il est intégralement connu de tous les processus.

L'approche de Newton consiste à scinder le calcul d'un déplacement de pas Δt en deux temps. En notant $u = (u_r, u_\theta)^{\mathsf{T}}$, le vecteur de champ de déplacement poloïdal, la position d'un point $\mathbf{x}$ au temps précédent s'exprime par :

$$\mathbf{x}_{t-\Delta t} = \mathbf{x}_t - \Delta t \cdot u(\mathbf{x}_t - \frac{\Delta t}{2} \cdot u(\mathbf{x})). \tag{6.7}$$

Cette méthode nécessite d'aller chercher les valeurs du champ de déplacement en un point intermédiaire. La distribution des données de Gysela++ fait que ce point peut se trouver sur une autre tuile, potentiellement distante. Il faudrait alors mettre en place un système de communication spécifique ou bien rapatrier localement sur chaque processus l'ensemble du champ de déplacement $r{\times}\theta$ du plan poloïdal. Ainsi, l'approche de Taylor utilisant les dérivées des champs de déplacement présente des avantages que nous allons exploiter Cela pourrait ne plus être le cas si l'opérateur de Poisson était amené à changer (voir section 6.2).

Après le calcul des champs de déplacement, l'autre composante principale de l'opérateur d'advection est l'interpolation de la valeur finale. Celle-ci peut se situer sur un processus distant et peut alors engendrer des communications. La section suivante propose un schéma d'interpolation distante amélioré.

6.1.2 Optimisation des communications point à point

Le système d'interpolation à distance présenté en section 5.2.1 présente un défaut majeur comme nous l'avons vu (une communication par demande d'interpolation, blocage du *thread* de calcul en attente du retour de la valeur interpolée). Le mécanisme introduit ici propose une solution paramétrable pour pallier ce problème. Un système de mémoire tampon (*bufferisation*) est mis en place pour les *threads* de calcul afin qu'ils ne soient plus bloqués par la communication MPI. D'autre part, les demandes d'interpolation à un processus distant sont groupées pour réduire le nombre de communications.

Le mécanisme est détaillé ci-après. Chaque *thread* de calcul dispose d'un *buffer* lui permettant de stocker ses demandes d'interpolation. Si lors de la localisation d'un point à interpoler celui-ci se trouve sur une tuile distante, le *thread* va enregistrer les coordonnées de ce point, la tuile où il est situé, le processus MPI où elle est localisée et l'adresse mémoire où la valeur à interpoler doit finalement être stockée (dans la fonction de distribution). Le *thread* tient à jour, en parallèle, un compteur pour indiquer le nombre de points qu'il a enregistré pour chaque processus distant. Le *thread* communiquant, lui, surveille l'ensemble de ces compteurs lors de son appel à la routine de vérification des notifications. Si le nombre de requêtes d'interpolation pour un processus dépasse un seuil fixé à l'avance, le *thread* va débuter l'envoie de ces demandes d'interpolation. *Thread* par *thread*, il va collecter les demandes destinées à ce processus dans son *buffer* de communication (coordonnées, indice de la tuile et adresse mémoire finale). Les accès aux *buffers* de demande des *threads* se font en exclusion mutuelle afin d'éviter les accès concurrents. Une fois le nombre de points souhaité récupéré, le *buffer* de communication est envoyé au processus distant qui le reçoit de façon classique dans sa routine de communication (`MPI_Iprobe`) et le copie dans un *buffer* dédié. Une tâche est alors créée afin que les interpolations soient effectuées par un *thread* de calcul, qui une fois terminé, notifie le *thread* de communication par un simple booléen qui lui aura été fourni et que le *thread* de communication surveille. Une fois la tâche terminée, le *thread* communiquant renvoie le *buffer* (valeur interpolée et adresse mémoire finale) qui est réceptionné de façon classique par le processus initial. Celui-ci recopie alors directement les valeurs interpolées à leur destination. Le *thread* communiquant du processus initial garde une trace des demandes envoyées afin de savoir s'il faut encore attendre des réponses à la terminaison d'une advection.

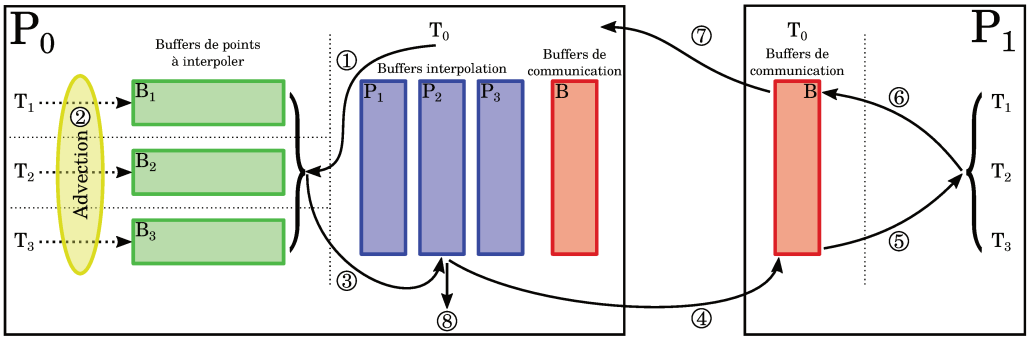


FIGURE 6.1 – Diagramme d'interpolation sur un processus distant.

La Figure 6.1 illustre ce mécanisme avec deux processus distants. Huit grandes étapes peuvent être distinguées et sont décrites ci-dessous :

1 - le *thread* maître surveille le nombre de points stockés dans les *buffers* de chaque

thread pour chaque processus ;
2 - les *threads* de calcul exécutent des tâches d'advection nécessitant des interpolations et stockent des requêtes à des processus distants dans leur *buffer* dédié ;
3 - le *thread* maître détecte suffisamment de points à destination d'un processus et extrait ces points des *buffers* des *threads* vers le *buffer* destiné au processus cible ;
4 - il poste de façon asynchrone l'envoi des points à interpoler ;
5 - le processus distant réceptionne les demandes d'interpolation et le *thread* maître distribue une tâche d'interpolation aux *threads* de calcul ;
6 - le *thread* maître est notifié de la terminaison de calculs ;
7 - les valeurs interpolées sont renvoyées au processus initial de façon asynchrone ; à réception, elles sont recopiées directement au bon emplacement dans la fonction de distribution ;
8 - à la fin de l'advection, le processus attend le retour de l'ensemble des demandes d'interpolations effectuées.

L'ensemble du mécanisme s'effectue de manière asynchrone et s'intègre au déroulement de l'exécution des tâches sur les différents processus en bloquant au minimum les *threads* de calcul. La section 7.1 présente les performances du code avec ce mécanisme. Cette optimisation s'applique également dans le cas d'un opérateur d'advection effectuant le déplacement 4D sans *splitting* directionnel [45, p.64]. Un tel opérateur est envisagé pour le futur du code GYSELA.

6.2 Opérateur de Poisson

L'opérateur de Poisson permet de calculer le potentiel électrostatique d'une distribution de particules. Dans le code GYSELA, celui-ci est exprimé sous la forme d'une équation de quasineutralité [36]. La densité ionique n_i est alors supposée constamment égale à la densité électronique n_e et s'exprime

$$n_\mathrm{i}(r,\theta,\varphi) = \iint f(r,\theta,\varphi,v_\parallel,\mu)\mathrm{d}v_\parallel\mathrm{d}\mu.$$

L'équation de quasineutralité pour un code 4D *drift-kinetic* [35] peut s'écrire :

$$-\operatorname{div}_\perp\left[\frac{n_0(r)}{B_0\Omega_\mathrm{c}}\nabla_\perp\Phi\right] + \frac{n_0(r)}{T_\mathrm{e}(r)}\left(\Phi - \langle\Phi\rangle_{\theta,\varphi}\right) = n_\mathrm{i}(r,\theta,\varphi) - n_0(r) \tag{6.8}$$

où T_e représente la température électronique, B_0 le champ magnétique à l'axe magnétique et Ω_c la vitesse cyclotronique de rotation des particules autour des lignes de champ. Cette équation permet de calculer le potentiel électrostatique à partir de la densité des particules, issue de la fonction de distribution. Néanmoins, elle forme un système 3D coûteux à résoudre. Un algorithme permettant de séparer les dimensions est mis en place et accélère la résolution du système.

6.2.1 Algorithme et implémentation

Il est possible de transformer ce système 3D en la combinaison d'un problème 1D et d'un système 2D, en considérant une représentation dans l'espace de Fourier [48]. Néanmoins, la solution ne sera pas entièrement redétaillée ici. Tout d'abord, en moyennant l'équation

(6.8) dans les dimensions θ et φ, on obtient l'équation (6.9) :

$$-\frac{\partial^2\langle\Phi\rangle_{\theta,\varphi}(r)}{\partial r^2}-[\frac{1}{r}+\frac{1}{n_0(r)}\frac{\partial n_0(r)}{\partial r}]\frac{\partial\langle\Phi\rangle_{\theta,\varphi}(r)}{\partial r}=\frac{\langle n_\mathrm{i}\rangle_{\theta,\varphi}(r)}{n_0(r)}-1. \tag{6.9}$$

Cette équation est ensuite discretisée en l'équation (6.10) afin d'être résolue par le solveur linéaire présenté en Annexe B. Soit un maillage de N_r points et de pas Δr, notons $p_i = \langle\Phi\rangle_{\theta,\varphi}(r_i),\ \forall i \in [\![0, N_r-1]\!]$ avec $r_i = \frac{\Delta r}{2} + i\Delta r$. L'équation forme alors un système tridiagonal 1D dont les inconnues sont les (p_i), les valeurs des différentes densités étant connues :

$$\frac{2p_i-p_{i-1}-p_{i+1}}{\Delta r^2}-[\frac{1}{r_i}+\frac{1}{n_0(r_i)}\frac{\partial n_0(r_i)}{\partial r}]\cdot\frac{p_{i+1}-p_{i-1}}{\Delta r}=\frac{\langle n_\mathrm{i}\rangle_{\theta,\varphi}(r_i)}{n_0(r_i)}-1. \tag{6.10}$$

Résoudre ce système permet d'obtenir le potentiel moyenné en θ et φ qui sera utilisé pour résoudre le système 2D par la suite. L'équation (6.9) est ensuite transcrite dans l'espace de Fourier dans la direction θ et après quelques transformations qui ne seront pas détaillées ici, cela forme un système 2D qui peut être résolu comme un ensemble de systèmes 1D avec le solveur présenté en Annexe B adapté aux nombres complexes. Le spectre du potentiel ainsi calculé est finalement retransformé dans l'espace réel.

Dans la discrétisation de l'équation de Poisson, des conditions au bord sont nécessaires pour les rayons intérieurs et extérieurs, r_0 et r_{N_r-1}. Pour le rayon extérieur, une condition de Dirichlet est appliquée et le potentiel est forcé à zéro (les particules restent confinées au cœur du champ magnétique). Pour le rayon intérieur $r_0 = \frac{\Delta r}{2}$, le maillage est défini pour que les opérateurs puissent s'étendre à travers le centre du plan (voir l'exemple de l'interpolation en section 4.3.2). Ainsi, le point directement en dessous de (r_0, θ_0) dans la direction r est le point $(r_{-1}, \theta_0) = (r_0, \theta_{N_{\theta[0]}/2})$. Dans le cas du potentiel moyenné en θ, l'indice p_{-1} pour le rayon r_0 correspond à p_0. L'équation (6.11) donnent l'expression de l'équation (6.10) dans ce cas particulier :

$$\frac{p_0-p_1}{\Delta r^2}-[\frac{1}{r_0}+\frac{1}{n_0(r_0)}\frac{\partial n_0(r_0)}{\partial r}]\cdot\frac{p_1-p_0}{\Delta r}=\frac{\langle n_0\rangle_{\theta,\varphi}(r_0)}{n_0(r_0)}-1, \qquad p_{N_r-1}=0. \tag{6.11}$$

Cette méthode [45, p.45] est implémentée dans le code Gysela++. L'algorithme 11 décrit le déroulement du calcul du potentiel. Dans un premier temps, la densité des particules est calculée à partir de la fonction de distribution 5D répartie sur l'ensemble des processus (ligne 1). Les différentes densités locales sont combinées afin que chaque processus dispose de la densité pour son sous-domaine 3D (ligne 2). La densité est ensuite moyennée sur les directions θ et φ et est répliquée sur l'ensemble des processus (lignes 3 et 4). Ceux-ci peuvent alors calculer le potentiel moyen pour tout r en résolvant l'équation (6.10) (ligne 5) avec la densité moyenne comme membre de droite et la matrice des coefficients du système calculée à l'initialisation. Afin de pouvoir effectuer la transformée de Fourier dans la direction θ et afin de pouvoir résoudre les systèmes 1D dans la direction r, il est nécessaire de posséder localement des plans poloïdaux de densité. C'est pourquoi une redistribution des tuiles 3D est effectuée (ligne 6). Le calcul du potentiel est ensuite effectué pour chaque plan poloïdal indicé par φ (boucle `for` ligne 7). À cause du maillage réduit, la transformée de Fourier en θ puis la résolution en r ne sont pas triviaux. En effet, chaque rayon r_i ayant potentiellement un $N_{\theta[i]}$ différent, la transformée de Fourier discrète générée aurait également $N_{\theta[i]}$ fréquences. À une fréquence m donnée, il ne serait donc

pas possible de résoudre un système 1D en r car celle-ci pourrait ne pas exister sur les rayons moins résolus. C'est pourquoi un plan poloïdal uniforme de taille (N_r,max($N_{\theta[i]}$)) est interpolé depuis le plan poloïdal réduit (ligne 8). Ce plan uniforme peut être manipulé puis transformé sans problème pour être résolu radialement (ligne 9) avant d'être manipulé de nouveau et de subir une transformation de Fourier inverse. Il est ensuite réinséré dans le maillage réduit en ne gardant que les points nécessaires. Les dérivées du potentiel sont calculées tant que les données sont dans cette distribution (ligne 11) car il y a alors moins de halos à communiquer (ligne 10). Le potentiel est finalement redistribué en 3D (ligne 12) [1].

Algorithme 11 : Pseudo-code de l'opérateur de Poisson.

```
1  pour tile5d ∈ tile5d_map faire                    /* Densité de particules */
       tile3d = tile5d.get_tile3d()
       OMP task depend(in:...) depend(out:...)
         tile3d.density = tile5d.compute_density()
2  notify(VPAR_MU_REDUCTION, DENSITY)    /* Agglomération de la densité */
3  density_avg = tile3d_map.compute_density_avg()    /* Densité moyenne */
4  notify(3D_BROADCAST, DENSITY_AVG)   /* Partage de la densité moyenne */
5  potential_avg = solve_1D(coeffs_avg,density_avg)  /* Potentiel moyen */
6  notify(REDISTR_3D_RTHETA, DENSITY)  /* Redistribution 3D → poloïdal */
7  pour φ ∈ [[φ_start, φ_end]] faire                  /* Plan poloïdal */
       OMP task depend(in:...) depend(out:...)
8        2Dplane = get_uniform_poloidal_plane(φ)
         2Dplane = 2Dplane - density_avg(r)
         2Dplane = FFT(2Dplane,θ)
9        pour m ∈ [[0, N_θ − 1]] faire                    /* Solveur */
           solve_1D(coeffs,2Dplane(:,m))
         2Dplane = 2Dplane + potential_avg(r)
         2Dplane = FFT^{-1}(2Dplane,θ)
         set_uniform_poloidal_plane(φ,2Dplane)
10 notify(HALO_3D_PHI_DO, POTENTIAL)              /* Échange des halos */
11 pour tile3d ∈ tile3d_map faire                 /* Dérivées du potentiel */
       OMP task depend(in:...) depend(out:...)
         tile3d.compute_pot_derivatives()
12 notify(REDISTR_3D_3D, POTENTIAL)          /* Redistribution 3D → 3D */
```

Cette méthode de résolution impliquant des transformées de Fourier n'est pas idéale pour le maillage réduit et engendre redistributions de données et interpolations en maillage uniforme (supprimant les bénéfices du maillage réduit). Dans l'optique du développement d'un code gyrocinétique en distribution 5D, il sera nécessaire de repenser l'algorithme de

1. Cette solution présente un bon passage à l'échelle jusqu'à environ 8.000 cœurs, au-delà de quoi la redistribution devient problématique. Celle-ci peut toutefois être évitée en utilisant une solution complexe à mettre en œuvre [48]

l'opérateur de Poisson pour qu'il s'adapte à cette distribution de données [8]. De nombreux autres solveurs existent, pour la plupart itératifs ou multi-grilles.

Les opérateurs de Poisson et de Vlasov, centraux au modèle gyrocinétique et *drift-kinetic*, reprennent dans leur implémentation dans le code GYSELA++ des schémas numériques éprouvés. Ils subissent une adaptation au modèle de programmation par tâches et avec communications asynchrones. L'opérateur d'advection est bien adapté au maillage grâce à l'interpolation spécifique que nous avons mis en œuvre. Le schéma numérique de Poisson devra être révisé afin d'exploiter au mieux les propriétés du maillage réduit. Le chapitre suivant porte sur l'évaluation des performances numériques et en temps d'exécution des différents composants de GYSELA++.

Chapitre 7

Étude et validation des performances

Sommaire

Le code Gysela++ implémente le modèle 4D *drift-kinetic* avec les opérateurs de Vlasov et de Poisson optimisés présentés au chapitre 6. Il utilise un modèle de programmation par tâches et des communications asynchrones ainsi qu'une structure de données mettant en œuvre un maillage non uniforme (voir chapitre 5). L'objectif de ce prototype est d'étudier la viabilité des différents paradigmes, utilisés de manière combinée, dans le cadre d'une évolution profonde du code Gysela. Bien que les optimisations déjà intégrées ne soient pas suffisantes pour un code de production, il est pertinent d'observer dès à présent le comportement du code lors d'un passage à l'échelle et de l'évaluer selon plusieurs critères de performances.

La section 7.1 propose une étude des performances du code selon différents aspects : étude de la parallélisation par tâches (section 7.1.1) et étude de la parallélisation MPI (section 7.1.2). Un cas d'étude en *weak scaling* jusqu'à 6144 cœurs sera présenté en section 7.1.3 en comparaison avec le code Gysela actuel. Pour conclure, une discussion autour du modèle, des améliorations et évolutions pouvant lui être apportées, ainsi que de ses limitations, sera menée en section 7.2.

7.1 Évaluation des performances

L'analyse des performances de Gysela++ est l'objet de cette section. Bien que moins optimisé qu'un code de production, Gysela++ propose déjà un certain nombre de paramètres permettant de jouer sur l'efficacité de différentes optimisations.

Le choix des dimensions des tuiles (illustration Figure 5.2) est un élément central dans les performances du code comme nous le verrons en section 7.1.1. Le choix du maillage poloïdal réduit (chaque rayon peut avoir une résolution différente en θ) conditionne la répartition des points en différentes tuiles. Afin d'avoir un ensemble représentatif de tuiles, un maillage réduit type, à trois couronnes (trois $N_{\theta[i]}$ différents) et n'engendrant pas une baisse de précision significative, est choisi. Celui-ci est donné, avec les conventions de notations introduites en section 4.3.2, par : $N_r = 128$, $N_\theta = (24 : \underline{32}, 48 : \underline{64}, 56 : \underline{128})$. Ce maillage sera référé par la suite comme maillage à $(128{\times}128{\times}N_\varphi{\times}N_{v_\parallel})$ points (réduit en $r{\times}\theta$). Les autres maillages seront dérivés de ce maillage de référence en multipliant ou divisant l'ensemble des valeurs par une puissance de deux. Le nombre minimal de points en θ sur une couronne est fixé à 32. Ainsi, pour un maillage avec $Nr = 256$, on aura $N_\theta = (48 : \underline{64}, 96 : \underline{128}, 112 : \underline{256})$. Afin d'améliorer la lisibilité, le détail du découpage des plans poloïdaux réduits ne sera pas répété (sauf pour des cas avec un maillage différent) et seuls N_r et $N_{\theta[N_r-1]}$ seront explicités.

Les performances sont évaluées selon plusieurs critères. Le modèle de programmation par tâches est éprouvé selon différents critères en section 7.1.1. Les différentes possibilités de parallélisation multi-processus (MPI) sont présentées en section 7.1.2. Puis une évaluation globale est faite en section 7.1.3.

7.1.1 Étude du modèle de programmation par tâches

Le modèle de programmation par tâches offre de nombreuses possibilités d'optimisation et de paramétrage du code. La taille des tuiles, la granularité des tâches ainsi que le choix de l'ordonnanceur influent sur ses performances. Cette section propose un examen de l'ensemble de ces paramètres ainsi que des visualisations des traces d'exécution du code, grâce à l'outil ViTE [1]. Ces traces sont générées pour chaque *thread* directement par le *runtime* (Intel KOMP et GNU) dans les versions les plus récentes.

Le choix de la taille des tuiles est crucial. Si celles-ci sont trop petites, le code perd en densité calculatoire et son empreinte mémoire augmente fortement à cause des halos (voir section 5.1.1). Si elles sont trop grandes, il perd en localité mémoire et l'ordonnancement a moins de flexibilité pour le placement des tâches. Après expérimentation, les dimensions minimales permettant un bon déroulement de l'exécution sont les suivantes : $n_r = 16$, $n_\theta = 16$, $n_\varphi = 8$, $n_{v_\parallel} = 16$. Cela représente 256KiO de données, ce qui correspond à la taille des caches L2 des processeurs actuels. Cela permet également d'avoir suffisamment de tuiles à distribuer entre un grand nombre de processus MPI, même pour un petit maillage, en gardant le surcoût mémoire inférieur à deux fois l'espace de stockage des données. Il est possible d'augmenter la taille des tuiles en fonction de l'architecture tout en restant dans la limite du cache L3 (partagé entre tous les cœurs).

La granularité des tâches doit être adaptée à l'ordonnanceur et à la machine. Elle ne doit pas être trop grosse, afin de pouvoir laisser une marge de manœuvre à l'ordonnanceur et de ne pas générer de temps d'attente où certains cœurs se retrouveraient sans tâche à exécuter. Elle ne doit pas non plus être trop fine, ce qui entraînerait un trop grand nombre de tâches, au risque que l'ordonnancement prenne autant de temps que l'exécution. La Figure 7.1 présente un extrait de la visualisation ViTE d'une exécution du code sur un maillage de $(256{\times}256{\times}32{\times}32)$ avec des tuiles de taille $n_r = 8$, $n_\theta = 8$, $n_\varphi = 4$, $n_{v_\parallel} = 4$ (8KiO). Chaque ligne représente les tâches (en couleur) exécutées par un *thread*, le *thread*

1. `https://vite.gforge.inria.fr`

1 étant le *thread* ordonnanceur. L'extrait montre l'advection en $v_\parallel$ pour certains *threads*. On observe entre chaque tâche (en vert et violet) une phase grise qui consiste à un temps d'inactivité du *thread* en attente de l'attribution de sa prochaine tâche. Ce temps est de quelques dizaines de microsecondes sur les machines que nous utilisons pour ces *benchmarks* (Poincaré et Occigen), et ce aussi bien pour l'ordonnanceur Intel que GNU. Des tâches plus lourdes sont nécessaires pour minimiser ces surcoûts. Après expérimentations, avec les dimensions de tuile données au paragraphe précédent, générer une tâche par tuile donne une bonne granularité.

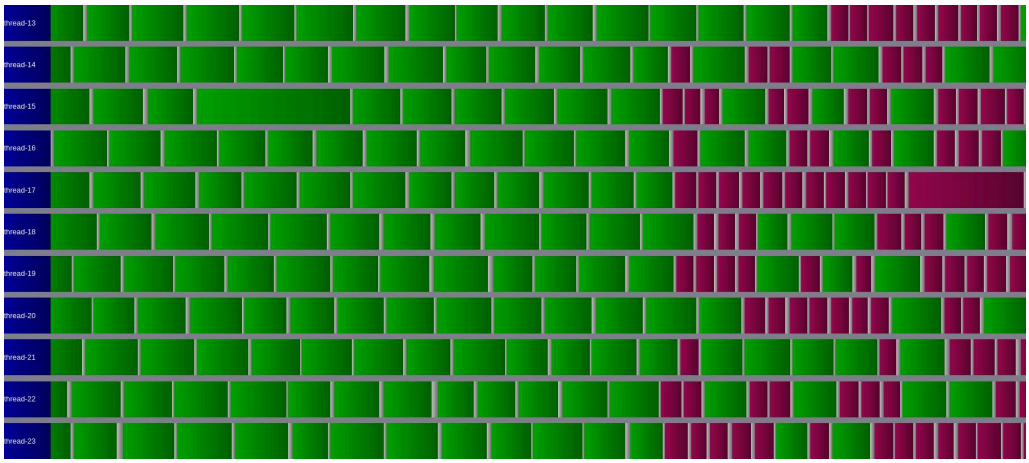


FIGURE 7.1 – Extrait de la trace d'une exécution sur un maillage de (256×256×32×32) points (réduit en $r\times\theta$) avec de petites tuiles et un processus MPI. Zoom sur l'opérateur d'advection $v_\parallel$ (en vert) et des copies de halos (en violet) suivantes. *Runtime* Intel.

La Figure 7.2 montre la trace d'une itération complète sur le même maillage que précédemment avec les dimensions de tuile conseillées en début de section. Le cas est exécuté avec un seul processus MPI sur un nœud du calculateur Occigen (deux processeurs de 12 cœurs). Les tâches violettes, jaunes et rouges correspondent respectivement aux advections en $v_\parallel$, φ et $r\times\theta$. Les tâches turquoises entre elles correspondent aux recopies de halos. Les détails de l'opérateur de Poisson (à gauche avant la première advection en $v_\parallel$) sont mieux visibles sur la Figure 7.3. Les tâches vertes correspondent au calcul de la densité, les tâches cyan au calcul du potentiel et les tâches orange au calcul des dérivées du potentiel. Les zones grises entre elles sont dues à des barrières et à des calculs séquentiels intermédiaires non tracés par l'outil. On observe globalement sur les deux figures un bon ordonnancement des tâches. De plus, leur granularité est assez régulière, ce qui montre le faible impact en matière de déséquilibrage de charge des différents traitements pour chaque type de tuile (régulière, jonction, centre et bord). Seules les tuiles centrales sont plus grosses par construction (elles prennent tous les points en θ et sont donc ici huit fois plus grosses que les autres tuiles). Les tâches relatives à ces tuiles sont créées en premier afin qu'elles soient ordonnancées pour exécution en premier.

En effet les tâches d'une advection peuvent être simplement créées dans l'ordre de numérotation des tuiles. Mais comme le montre la Figure 7.4, une phase d'attente impromptue peut être générée par l'ordonnanceur à la fin de l'exécution des tâches de l'advection. En effet, la tâche associée à une des tuiles centrale est systématiquement ordonnancée à la fin (entourée en jaune). Afin d'éviter ce problème, il serait intéressant que des priorités puissent

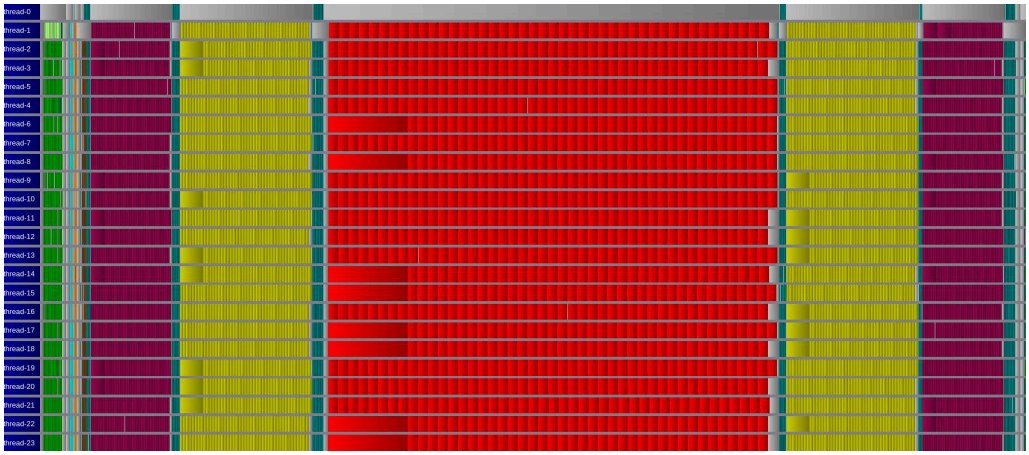


FIGURE 7.2 – Extrait de la trace d'une exécution sur un maillage de (256×256×32×32) points (réduit en $r\times\theta$) avec un processus MPI. Zoom sur une itération. *Runtime* GNU.

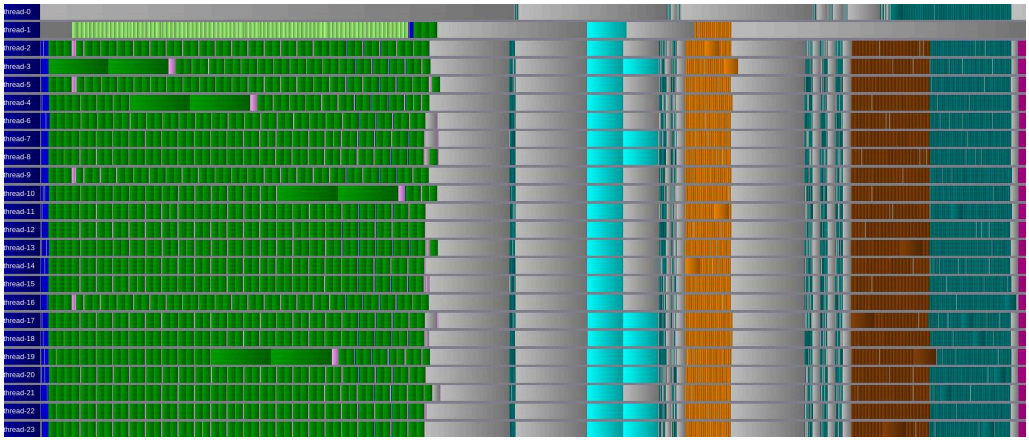


FIGURE 7.3 – Extrait de la trace d'une exécution sur un maillage de (256×256×32×32) points (réduit en $r\times\theta$) avec un processus MPI. Zoom sur l'opérateur de Poisson. *Runtime* GNU.

être spécifiées au *runtime* lors de la création des tâches. Cela est techniquement possible dans le standard OpenMP mais en pratique, très peu d'implémentations les prennent en compte.

La Figure 7.5 donne la trace d'une itération sur le même cas qu'en Figure 7.2 mais avec le *runtime* Intel. L'ordonnancement des tâches y est très similaire. Il est important de noter que la création de l'ensemble des tâches pour chaque opérateur est relativement rapide. Pour l'advection poloïdale par exemple, la génération de toutes les tâches représente un dixième du temps d'exécution d'une tâche (non tuile centrale).

La parallélisation par tâches offre comme désiré une bonne occupation des cœurs de calcul lorsque les bons paramètres sont choisis pour la taille des tâches. On pourrait néanmoins opter pour des tailles plus petites si le *runtime* était en mesure de réduire ses coûts de gestion. L'utilisation des caches et les performances seraient ainsi améliorées. Néanmoins, certaines fonctionnalités manquent encore dans les *runtimes*, et elles nécessitent d'être remplacées manuellement.

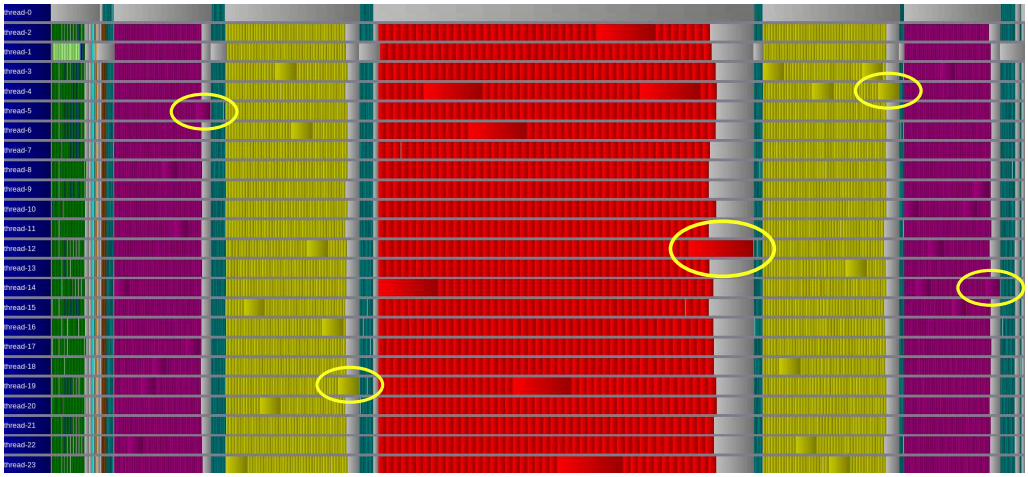


FIGURE 7.4 – Extrait de la trace d'une exécution sur un maillage de (256×256×32×32) points (réduit en $r\times\theta$) avec un processus MPI. Zoom sur une itération. *Runtime* GNU sans ordonnancement manuel.

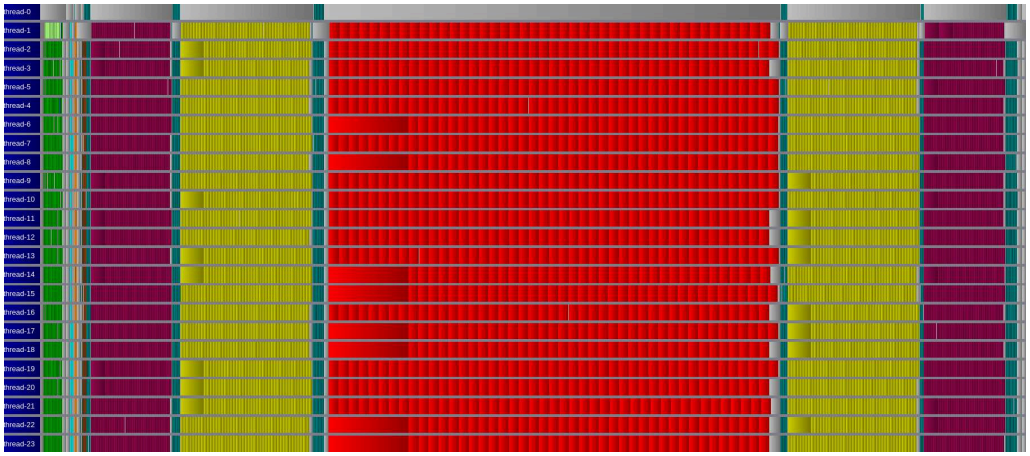


FIGURE 7.5 – Extrait de la trace d'une exécution sur un maillage de (256×256×32×32) points (réduit en $r\times\theta$) avec un processus MPI. Zoom sur une itération. *Runtime* INTEL.

7.1.2 Parallélisation MPI

Contrairement au code GYSELA où la parallélisation MPI ne peut être faite que dans les dimensions r et θ simultanément (auxquelles il faut ajouter μ en 5D), GYSELA++ propose une distribution de données et de calculs dans l'ensemble des dimensions (voir section 5.1.2). Les dimensions r et θ étant liées de manière complexe par le maillage réduit, les paramètres effectifs de la parallélisation MPI sont le nombre de processus dans chaque dimension : $\mathtt{Np}_{r\theta}$, $\mathtt{Np}_{\varphi}$ et $\mathtt{Np}_{v_\parallel}$. Les études présentées dans cette section ont été effectuées sur le calculateur Occigen [2] (CINES). Elles caractérisent le passage à l'échelle des opérateurs en fonction des différentes décompositions de domaines.

La Figure 7.6 donne le détail des temps d'exécution en *strong scaling* pour une parallé-

2. Un nœud de calcul est composé deux Haswell E5-2690 dodecacore associés à 32GiO de mémoire vive chacun.

lisation le long de la dimension $r\times\theta$. Le maillage utilisé est de (1024×1024×32×32) points (réduit en $r\times\theta$) et chaque processus utilise 12 *threads*. Les courbes donnent les temps exécution moyens des parties les plus importantes des opérateurs de Poisson et de Vlasov sur une trentaine d'itérations. Les temps totaux de chaque opérateur sont en traits pleins et les détails (sous parties interne des opérateurs) en trait pointillés. L'opérateur de Poisson utilise des carrés, celui d'advection des croix. La prépondérance des temps de calcul pour l'advection comparé au solveur de Poisson observée en section 7.1.1 est confirmée. On remarque que l'ensemble *scale* de façon régulière sauf pour deux courbes : les transpositions et l'advection $r\times\theta$. La transposition des données 3D avant et après le calcul du potentiel subit une forte envolée pour les cas à 2 et 4 processus, car le calcul du potentiel nécessite un plan poloïdal complet localement (voir section 6.2). Il adopte toutefois par la suite un scaling *faible* mais notable. Le temps d'exécution de l'advection $r\times\theta$ entre le cas à 1 processus et celui à 2 processus a un *scaling* légèrement inférieur au reste de la courbe du fait de l'introduction des communications MPI dans l'advection. Le faible déplacement des particules dans ces dimensions [3] fait que seuls peu de points nécessitent des interpolations distantes, ce qui permet à l'opérateur d'avoir un bon *scaling*. D'autre part, le calcul du potentiel et de la densité ont des temps d'exécution très faibles qui ne leur permettent pas de *scaler* aussi bien que les advections. À 64 nœuds, l'efficacité de la parallélisation MPI en $r\times\theta$ est d'environ 40% pour les advections et de 50% pour Poisson. Une étude globale est présentée en section 7.1.3 qui permet de mettre ces efficacités en perspective.

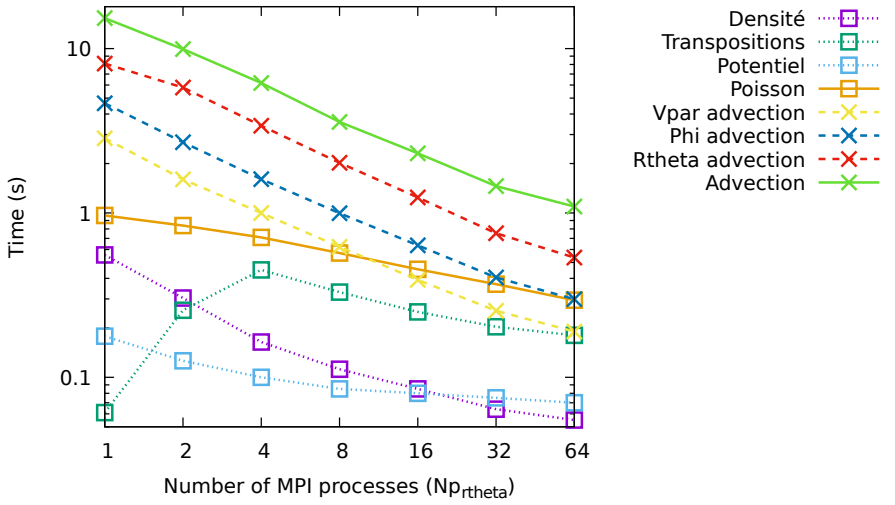


FIGURE 7.6 – Temps d'exécution moyen des différentes parties des opérateurs d'advection et de Poisson sur 30 itérations en fonction du nombre de processus dans la direction $r\times\theta$. Le maillage est de (1024×1024×32×32) points (réduit en $r\times\theta$).

La Figure 7.7 donne le détail des temps d'exécution en *strong scaling* dans la dimension φ. Le maillage utilisé est de (128×128×512×32) points (réduit en $r\times\theta$). Dans un premier temps, l'opérateur de Poisson se comporte désormais bien puisque les données sont dans la bonne distribution et il n'y a pas de transposition nécessaire (on observe d'ailleurs que celles-ci et le calcul du potentiel atteignent un palier). Les advections en $r\times\theta$ et $v_\parallel$

3. Des cas tests différents peuvent donner des déplacements plus significatifs générant plus de communications. La scalabilité de l'opérateur serait alors modifiée.

présentent également un bon *scaling*. L'advection en φ est ici plus problématique puisque le déplacement des particules dans cette direction est en moyenne relativement important et une grande majorité d'entre elles nécessite des communications pour une interpolation sur un processus distant. L'apparition des communications MPI (cas à 1 et 2 processus) est ici plus marquée que pour l'advection $r{\times}\theta$. Le surcoût de la gestion des interpolations distantes est fort mais relativement bien absorbé par le système de *bufferisation*[4] présenté en section 6.1.2. L'efficacité pour Poisson à 16 nœuds (avant le palier) est de 80% ; il faut noter que son coût reste négligeable par rapport à l'advection. L'efficacité à 64 nœuds pour l'advection φ est d'environ 40%, 51% pour l'advection $r{\times}\theta$, et 52% pour $v_\parallel$.

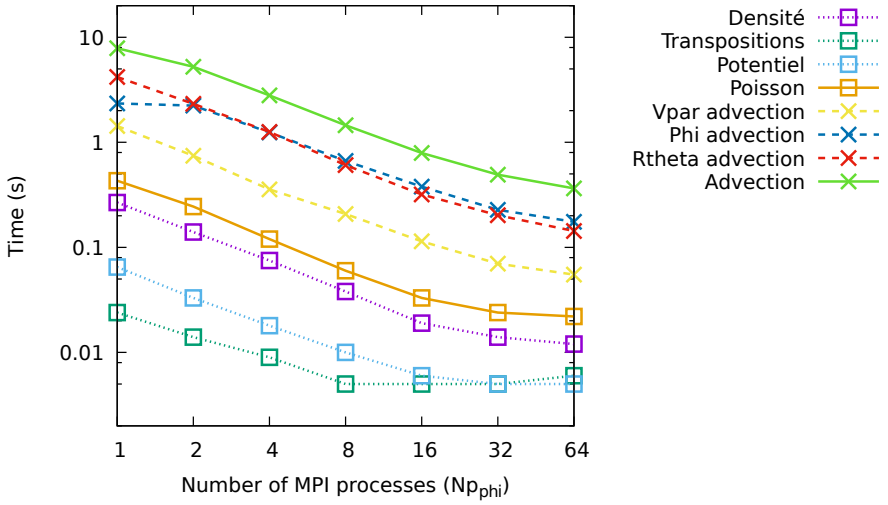


FIGURE 7.7 – Temps d'exécution moyen des différentes parties des opérateurs d'advection et de Poisson sur 30 itérations en fonction du nombre de processus dans la direction φ. Le maillage est de (128×128×512×32) points (réduit en $r{\times}\theta$).

La Figure 7.8 donne le détail des temps d'exécution en *strong scaling* dans la dimension $v_\parallel$. Le maillage utilisé est de 128×128×32×512 points (réduit en $r{\times}\theta$). Ici, l'ensemble des conditions sont réunies pour un bon *scaling* de l'ensemble des composantes puisque le déplacement en $v_\parallel$ est très faible et le nombre de particules ayant besoin d'interpolation distante est quasiment nul. L'opérateur de Poisson atteint vite un seuil du fait de la faible quantité de points du plan poloïdal. On observe même une remontée sur les cas à 32 et 64 nœuds, due à la présence de communications globales dans la direction $v_\parallel$ pour le calcul de la densité[5]. L'efficacité à 64 nœuds pour les advections $r{\times}\theta$ et φ est d'environ 60% et de 45% pour l'advection $v_\parallel$ (elle est à 55% à 32 nœuds mais chute à cause des communications de halos).

Les différentes possibilités de parallélisation MPI montrent des efficacités variables : la direction $v_\parallel$ a une bonne efficacité sur les cas présentés, la direction $r{\times}\theta$ une efficacité légèrement inférieure. La direction φ est ici la moins favorable à cause d'un surplus de

4. Comparativement à l'advection en φ sans *bufferisation* où le temps d'exécution était multiplié par dix entre le cas à 1 processus et celui à 2. Chaque interpolation distante générait une communication bloquante pour le *thread* appelant.

5. Une solution à ce problème a déjà été développée et éprouvée dans GYSELA [47], mais n'a pas été implémentée ici.

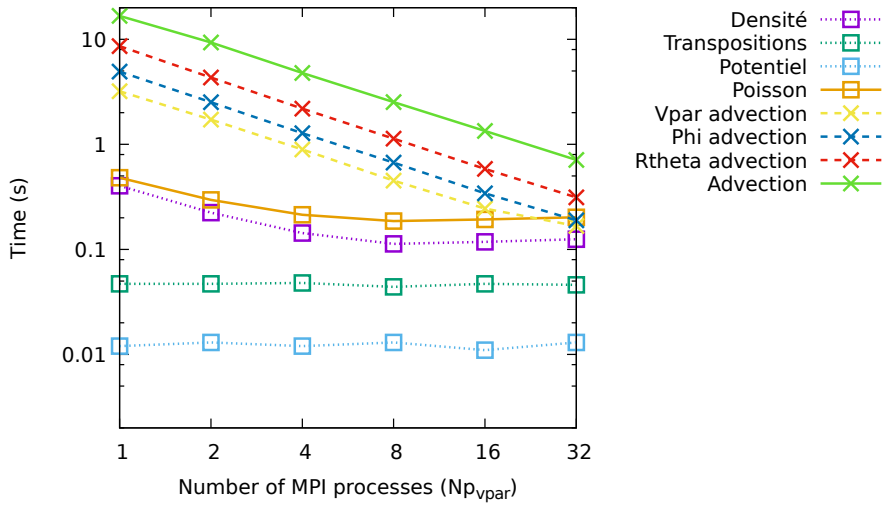


FIGURE 7.8 – Temps d'exécution moyen des différentes parties des opérateurs d'advection et de Poisson sur 30 itérations en fonction du nombre de processus dans la direction $v_\parallel$. Le maillage est de (128×128×32×512) points (réduit en $r\times\theta$).

communications. D'autres optimisations discutées en section 7.2 peuvent être apportées afin de se rapprocher d'une efficacité optimale. Il faut également noter, comme les courbes le montrent déjà, que la scalabilité dans les différentes dimensions dépend fortement de l'opérateur. La dimension de parallélisation la moins efficace peut être amenée à changer lors de l'introduction de nouveaux opérateurs (la gyromoyenne pèse un peu plus sur la direction $r\times\theta$ et les opérateurs de collisions impliqueraient fortement les dimensions $v_\parallel$ et μ dans un code gyrocinétique complet). Différents cas peuvent également amener à différentes distributions de charge entre les opérateurs. Le choix de la parallélisation MPI optimale dépend donc de nombreux facteurs. Pouvoir combiner la parallélisation des calculs dans différentes dimensions en ayant un bon *scaling* pour chaque est donc un avantage : GYSELA++ en offre quatre quand GYSELA n'en dispose que de deux (respectivement cinq et trois en gyrocinétique).

7.1.3 Cas dimensionnant

Cette section présente une étude des performances du code dans des configurations plus grandes. Celles-ci sont comparées aux performances du code GYSELA. Sur des maillages plus gros que ceux utilisés en section 7.1.2, l'advection en φ pose des problèmes notoires de scalabilité des temps d'exécution. Le grand nombre d'interpolations distantes requises éprouve fortement le système de *bufferisation* tel qu'il est conçu. Plus de détails sur les raisons de ces limites et sur les solutions envisageables afin de pouvoir accepter plus d'interpolations distantes sont donnés en section 7.2. Ces nouvelles approches n'ayant pas été mises en œuvre dans le code, la parallélisation en φ sera évitée pour les cas tests qui suivent.

La Figure 7.9 donne les résultats d'une analyse en *weak scaling* à partir d'un maillage de (256×256×64×64) points (réduit en $r\times\theta$) sur le plus petit cas. Ce maillage est constitué de 3648 tuiles ($(n_{r\theta}, np, nv) = (57, 8, 8)$) et représente 9.5GiO de données. Chaque processus MPI utilise 12 cœurs. L'étude a été réalisée sur la machine Occigen (CINES). Le *scaling*

a été effectué dans la même configuration pour le code GYSELA (même maillage mais non réduit en $r \times \theta$ et même nombre de processus et de *threads*). La Table 7.1 donne pour chaque cas du *weak scaling* de GYSELA++, le maillage, le nombre de tuiles, la distribution des processus MPI et l'empreinte mémoire moyenne. La distribution des tuiles entre les processus offre une bonne distribution des données puisque l'empreinte mémoire ne varie que de quelques pourcents entre les différents processus (9% pour le cas à 4 processus et 3% pour le cas à 512 processus). Pour l'étude de GYSELA, la distribution des processus MPI ne pouvant se faire que dans les dimensions r et θ, le nombre de processus MPI est alternativement doublé dans ces directions en commençant par r (N_r=2, N_θ=1 pour 2 processus ; N_r=2, N_θ=2 pour 4 processus ; N_r=4, N_θ=2 pour 8 processus...).

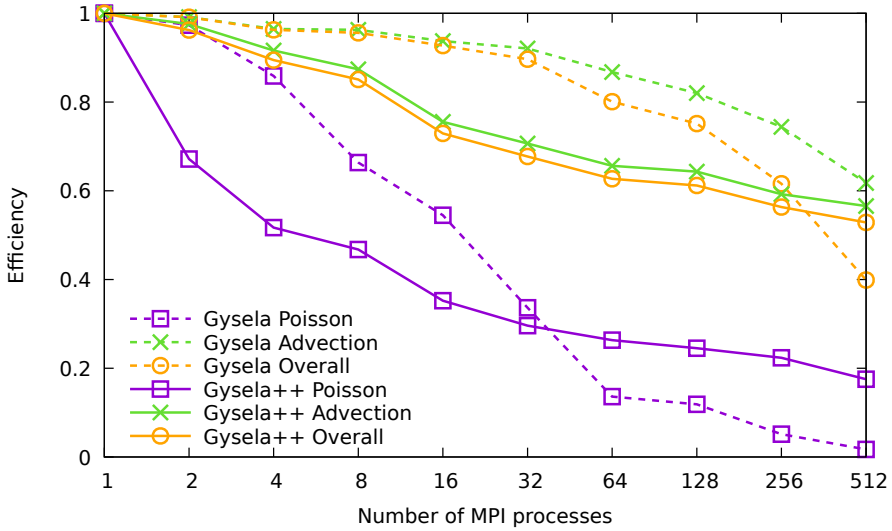


FIGURE 7.9 – Efficacité moyenne sur 30 itérations en fonction du nombre de processus MPI en *weak scaling* pour les codes GYSELA++ (en traits pleins) et GYSELA (en traits pointillés). Le maillage de base est de (256×256×64×64) points (réduit en $r \times \theta$ et représentant 9.5GiO de données pour GYSELA++).

On observe sur la Figure 7.9, pour le code GYSELA++ que l'opérateur de Poisson montre une efficacité inférieure à l'advection, due aux communications 3D globales comme évoqué en section précédente. D'autre part, certaines parties de cet opérateur ne sont pas scalables en $v_\parallel$. Le comportement global de l'application reste tout de même largement dominé par celui de l'advection. Cette étude montre des résultats cohérents avec les *strongs scaling* unidimensionnels. Pour le code GYSELA, on observe un meilleur *scaling* sur les cas à peu de processus, mais qui décroît plus fortement pour les cas les plus gros jusqu'à être moins efficace que GYSELA++ à 512 processus. Cette différence est due à une meilleure répartition des charges et aux possibilités de parallélisation accrues de GYSELA++. En effet, la possibilité de paralléliser dans plusieurs dimensions permet d'atteindre un nombre de processus bien plus élevé avec une meilleure efficacité ; il n'y a plus de dimension "sur-parallélisée". Ces bons résultats se retrouveraient également sur une extension réellement 5D de GYSELA++ avec l'utilisation de plusieurs valeurs de μ, voir 6D avec l'ajout d'une seconde espèce de particules. La parallélisation dans ces dimensions est en effet très peu coûteuse de par le peu d'interactions [6] entre les différents μ et entre les différentes espèces.

6. La forme des interactions dépend fortement du type d'opérateur de collision utilisé.

Cas	Maillage	Tuiles $(r\times\theta,\varphi,v_\parallel)$	MPI $(r\times\theta,\varphi,v_\parallel)$	Mémoire
1	256×256×64×64	(57,8,8)	(1,1,1)	9.5GiO
2	512×256×64×64	(121,8,8)	(2,1,1)	9.4GiO
4	512×512×64×64	(241,8,8)	(4,1,1)	9.6GiO
8	512×512×64×128	(241,8,16)	(4,1,2)	9.6GiO
16	512×512×128×128	(241,16,16)	(8,1,2)	9.5GiO
32	512×512×128×256	(241,16,32)	(8,1,4)	9.6GiO
64	1024×512×128×256	(497,16,32)	(16,1,4)	9.7GiO
128	1024×512×128×512	(497,16,64)	(16,1,8)	9.7GiO
256	1024×1024×128×512	(993,16,64)	(32,1,8)	9.6GiO
512	1024×1024×256×512	(993,32,64)	(32,1,16)	9.7GiO

TABLE 7.1 – Récapitulatif des paramètres des différents cas de l'étude. Les dimensions du maillage, le nombre de tuiles et le nombre de processus MPI dans chaque dimension ainsi que l'empreinte mémoire moyenne par processus sont donnés.

Dans un cas similaire à un cas type de production de GYSELA (deux espèces et soixante-quatre valeurs de μ), la scalabilité serait comparable (c'est-à-dire au dessus de 50%) et porterait jusqu'à 786k cœurs. On voit donc que l'approche du code GYSELA++ est un atout pour s'exécuter sur les architectures *manycore* et sur les futures machines *exascale*.

7.2 Conclusion et discussion sur le prototype GYSELA++

De nombreuses optimisations restent possibles à mettre en œuvre dans le code GYSELA++ et certaines méritent d'être mentionnées, car importantes pour prendre la mesure des atouts potentiels de cette approche. D'autres remarques sur les algorithmes sont également mentionnées.

7.2.1 Entrelacement des opérateurs

À terme, il peut être envisagé que la gestion des tâches et de leurs dépendances puisse être suffisamment fine pour permettre à différents opérateurs de s'exécuter en parallèle sur différentes tuiles. On peut imaginer une tuile régulière qui aurait le temps d'effectuer une advection, puis de récupérer ses halos depuis d'autres tuiles au même stade, puis d'effectuer l'advection suivante pendant qu'une tuile centrale ferait sa première advection. Il pourrait alors être intéressant d'avoir plus de deux copies de la fonction de distribution pour augmenter un parallélisme temporel. À noter que le code GYSELA++ possède déjà les mécanismes permettant ce recouvrement, mais la gestion des dépendances des tâches de communication n'est pas présente [7] (ni dans OpenMP, ni manuellement dans le code). Rajouter une fonction de distribution présente toutefois un coût mémoire non négligeable. Mais une fonction de distribution supplémentaire serait très utile lors des diagnostics afin de permettre à la simulation de continuer pendant que les écritures disque se déroulent (les

7. De plus, les communications des échanges de halos ne sont pas séparées. L'ensemble des halos d'un processus est communiqué d'un bloc, ce qui engendre une synchronisation au début de cette phase de communication. L'introduction d'une mise à jour de halos tuile à tuile dépend de la gestion des dépendances distantes.

I/O représentent une part importante du temps d'exécution de GYSELA). Cette solution est celle mise en place dans le code GYSELA de production.

7.2.2 Priorité d'exécution et poids des tâches

Comme mentionné en section 7.1.1, la gestion des priorités d'exécution entre les tâches n'est pas encore entièrement présente dans les implémentations d'OpenMP. Cela pose un problème lors de l'ordonnancement des tâches des tuiles qui doivent être créées manuellement dans un ordre établi. Cet ordre dépend de plusieurs facteurs, par exemple la charge de calcul que l'on peut représenter par un poids. Ce poids peut être calculé pour chaque tuile à l'initialisation. Il dépend de leur taille et de leur type (différence de traitement dans les opérateurs). Il y a donc autant de poids que de triplet (taille, type, opérateur). Il peut être intéressant de *scheduler* en premier les tuiles à l'intérieur d'un sous-domaine afin de laisser aux communications de halos provenant de processus distants de terminer pendant que l'ensemble des cœurs effectue des tâches. Ce schéma fonctionne pour la gyromoyenne qui utilise des halos, mais pas pour les advections dont les différents champs de déplacement sont difficilement connus à l'avance.

7.2.3 Amélioration de la *bufferisation*

Les résultats ont montré que le système de *bufferisation* tel qu'il est actuellement conçu ne *scale* pas aussi bien qu'on pourrait le souhaiter. Lors de la réception de demandes d'interpolations distantes, les tâches créées afin d'effectuer ces calculs viennent en concurrence avec les tâches locales d'advection. En l'absence de gestion de priorités ou de poids dans le graphe des tâches, l'ordonnancement est majoritairement chronologique. Or, la soumission de l'ensemble des tâches d'advection est très rapide par rapport à leur exécution comme nous l'avons vu en section 7.1.1. Si un grand nombre de points nécessitent une interpolation distante, les *buffers* locaux se remplissent très vite (parfois avant même que la première demande d'interpolation distante n'arrive) et les *threads* de calcul sont bloqués dans une tâche en attente que les *buffers* se libèrent. Le *thread* de communication doit donc effectuer lui-même les interpolations afin de débloquer la situation. Plusieurs solutions sont envisageables.

Une première approche est d'augmenter la taille des *buffers* locaux. Mais l'ordonnancement chronologique fait que toutes les demandes d'interpolation distantes doivent être enregistrées avant que les *threads* de calcul ne se voient assigner des tâches d'interpolations distantes. Une autre solution serait de remplacer les phases d'attentes des *threads* de calcul (verrous OpenMP) par des changements de contexte (*task yield*) afin que d'autres tâches soient exécutées en attendant. En pratique, les tâches ordonnancées en remplacement sont majoritairement des advections et se retrouvent également en attente des *buffers* locaux. Il faudrait alors envisager d'avoir un moyen autre que les priorités pour influencer l'ordonnanceur. Par exemple créer deux types de tâches – interpolation locale et interpolation distante – et lorsque les *buffers* locaux sont remplis, demander à l'ordonnanceur de ne plus exécuter que des tâches de type interpolation distante jusqu'à ce que les *buffers* locaux se vident. Dans l'idéal, une combinaison de toutes ces solutions permettrait une exécution de la *bufferisation* tel qu'envisagé au départ. Néanmoins, une correction pérenne de l'algorithme de *bufferisation* n'est pas possible avec les implémentations OpenMP actuelles.

7.2.4 Stockage des halos

Actuellement, les halos sont les mêmes dans toutes les dimensions : une tuile 4D de taille $(n_r \times n_\theta \times n_\varphi \times n_{v_\parallel})$ est représentée avec son halo par un bloc de taille $(\texttt{ld}_r \times \texttt{ld}_\theta \times \texttt{ld}_\varphi \times \texttt{ld}_{v_\parallel})$ avec $\texttt{ld}_x = n_x + \frac{N_{\text{deriv}}}{2}$ où N_{deriv} est la taille du *stencil* d'interpolation. De nombreux points du halo ne sont en fait jamais utilisés. Par exemple si l'on considère l'opérateur d'advection, les sous-opérateurs sont des advections 2D $r \times \theta$, 1D φ et 1D $v_\parallel$. Les points de halo dans les "coins" de ces quatre dimensions ne sont pas utiles. Sur l'exemple d'une tuile 3D, tous les points du halo sont utiles pour les plans poloïdaux de 0 à $n_\varphi - 1$. Mais pour les $\frac{N_{\text{deriv}}}{2}$ plans avant et après dans le halo φ, seuls les points du cœur sont utiles ; les halos $r \times \theta$ ne sont pas utilisés dans les halos φ.

En 4D, la taille actuelle d'une tuile vaut $\texttt{ld}_r\texttt{ld}_\theta\texttt{ld}_\varphi\texttt{ld}_{v_\parallel}$ quand elle pourrait être au minimum de

$$\texttt{ld}_r\texttt{ld}_\theta n_\varphi n_{v_\parallel} + 2 n_r n_\theta (\overbrace{n_{v_\parallel} \frac{N_{\text{deriv}}}{2}}^{\varphi \text{ halos}} + \overbrace{n_\varphi \frac{N_{\text{deriv}}}{2}}^{v_\parallel \text{ halos}}).$$

Par exemple, pour un maillage de $(256 \times 256 \times 64 \times 128)$ avec des tailles de tuile de $(16 \times 16 \times 16 \times 16)$, on a alors 491 tuiles en $r \times \theta$, 4 en φ et 8 en $v_\parallel$ et la mémoire consommée est de 64GiO quand elle pourrait être de 33GiO si seul le minimum des points était alloué. C'est bien évidemment différent si on passe à une interpolation 4D. À ce moment-là, tous les points du halo auraient une utilité potentielle.

D'autre part, il faut éviter d'avoir des halos "locaux", c'est-à-dire de dupliquer des données locales afin de remplir un halo sur chaque tuile du processus. Le gain en termes de temps de calcul n'est potentiellement pas viable au regard du surcoût mémoire engendré. Par exemple, sur le cas précédent, la mémoire consommée est de 64GiO quand la taille effective du maillage est de 4GiO. Il serait plus judicieux de garder un stockage séparé de chaque tuile (pour la localité spatiale et temporelle dans les calculs), mais de mettre en place des indirections vers les tuiles correspondantes pour les accès aux halos (interpolations au bord d'une tuile par exemple).

De nombreuses optimisations sont également possibles pour les communications MPI qui sont un problème central dans la parallélisation sur un supercalculateur. Par exemple, envisager d'avoir un *thread* de communication par processus distant afin de fluidifier les communications (sous réserve que les communications multi-*threadées* fonctionnent et soient performantes), ou encore ne plus utiliser de `MPI_Iprobe` mais poster directement tous les `MPI_Irecv` et faire des `MPI_Wait` afin que les communications puissent avancer dès leur envoi. Nous ne les détaillerons toutefois pas ici, car elles ne sont pas spécifiques à l'application GYSELA++. Toutes ces remarques et améliorations doivent être gardées à l'esprit, car la marge de progrès est grande dans cette direction.

Le code GYSELA++ montre globalement des performances correctes lors du passage à l'échelle. L'ordonnancement des tâches et l'entrelacement avec les communications se montrent efficaces et la possibilité de distribuer données et calculs dans différentes dimensions est un atout non négligeable. Bien que des optimisations soient encore possibles, les techniques utilisées ont déjà su montrer qu'elles avaient un potentiel intéressant. Les travaux effectués sur ce prototype ont permis de clarifier certains choix de conception pour le futur code GYSELA.

Conclusion et perspectives

La simulation de plasmas turbulents dans le cadre de la fusion nucléaire magnétiquement confinée est un domaine interdisciplinaire complexe. La physique des tokamaks met à l'épreuve des modèles numériques, des algorithmes et des architectures de calcul toujours plus évoluées par le biais de nombreux paradigmes de programmation parallèle et distribuée. La génération de supercalculateur pré-*exascale* propose des calculateurs aux nœuds de calcul plus nombreux et disposant de plus de cœurs que ceux des générations précédentes. Le code Gysela possède une parallélisation efficace, mais qui peut être améliorée au regard de ces évolutions comme c'est aussi le cas de nombreux codes de production. L'objectif de cette thèse s'inscrivait dans ce contexte et visait à proposer de nouveaux schémas et algorithmes parallèles pour certaines parties du code actuel ainsi qu'à étudier des solutions nouvelles en vue d'évolutions physiques envisagées pour le code. L'opérateur de gyromoyenne montrait de bonnes performances, mais s'adaptait mal à la décomposition de domaine parallèle. Un algorithme distribué a donc été proposé et se montrait efficace sauf pour certaines configurations. Des travaux ont été menés pour adapter l'algorithme à ces situations. D'autre part, de nombreuses demandes d'évolutions du code de la part des physiciens, tant au niveau des types de particules simulées que de la géométrie du problème, ont amené à envisager une réécriture de l'application Gysela. Ces nouvelles capacités de modélisation amélioreront le réalisme des simulations en permettant de représenter cœur et bord du plasma simultanément. Un prototype appelé Gysela++, proposant plusieurs modifications structurelles et utilisant des paradigmes de parallélisation différents du code actuel a été conçu et développé dans le cadre de cette thèse. Il constitue la base de la prochaine mouture du code Gysela qui vise à demeurer maintenable et performante sur les architectures de calcul haute performance présentes et à venir.

Synthèse des travaux

La première étude de cette thèse a porté sur l'implémentation et l'optimisation dans le code Gysela de l'opérateur de gyromoyenne distribuée. Le coûteux changement de distribution de données utilisé auparavant (redistribution de la structure de donnée 5D) a été remplacé par un échange de zones tampons (halos) entre processus voisins. Un système de recouvrement de calcul et de communication a également été mis en place grâce à un entrelacement fin de la parallélisation multicœur (OpenMP) et des communications entre nœuds de calcul (MPI). Ces modifications ont permis d'accélérer les temps d'exécution de l'opérateur de gyromoyenne distribuée par un facteur deux, tout en diminuant son empreinte mémoire d'autant. Ces améliorations ont toutefois été remises en cause par des géométries plus réalistes mises en place dans l'application (suppression du trou central au niveau de l'axe magnétique). Des études plus approfondies ont été nécessaires afin d'étendre l'algo-

rithme distribué à ces nouveaux cas. Des adaptations ont été proposées, mais ne se sont pas montrées entièrement satisfaisantes, car nécessitant des communications ne passant pas à l'échelle. C'est pourquoi une réflexion plus globale sur le problème a été menée afin de proposer une approche nouvelle tout en prenant en compte les évolutions prévues du code et des machines dans les années à venir. L'introduction du maillage poloïdal réduit, dans lequel la répartition des points de maillage est adaptative en espace, a permis de diminuer le surcoût engendré par l'ajustement de l'algorithme de gyromoyenne en abaissant le volume des communications ajoutées. La réduction globale, sans perte de précision, du nombre de points de maillage et la possibilité de raffiner uniquement certaines zones du plan procurent également d'autres avantages. Ce maillage réduit s'est montré numériquement stable pour les opérateurs centraux de la théorie gyrocinétique (équations de Vlasov, équation de Poisson, gyromoyenne). L'introduction de ce maillage dans le code existant étant trop complexe à mettre en œuvre et de nombreuses autres évolutions du code étant aussi en préparation, la conception d'une nouvelle version du code a été initiée. Le prototype Gysela++ expérimente différentes techniques de parallélisation afin d'évaluer si elles s'adaptent bien au problème considéré et si leur mise en place dans un code de production est pertinente.

La seconde partie de cette thèse porte donc sur la mise en œuvre et l'évaluation du prototype Gysela++. Celui-ci met en œuvre une version réduite du modèle gyrocinétique afin de limiter les coûts de développement au minimum nécessaire pour pouvoir poser les bases d'une future version du code Gysela. Il repose sur la combinaison d'un modèle de programmation par tâches et d'un modèle de communication asynchrone, le tout utilisant une structure de données adaptative sur un maillage réduit. Cette association de paradigmes forme déjà un code conséquent et complexe (plus de quinze mille lignes). L'opérateur de Poisson utilise un algorithme parallèle éprouvé et l'opérateur de Vlasov a fait l'objet d'une adaptation particulière au modèle de calcul et de communication. L'ensemble se montre efficace et passe à l'échelle sur plusieurs milliers de cœurs. On observe notamment que l'ordonnancement des tâches, très similaire entre ordonnanceurs, maximise bien l'occupation des cœurs ainsi que cela était envisagé. Comme pour le code Gysela, l'efficacité de la parallélisation MPI dépend fortement des différents opérateurs, mais la possibilité de paralléliser dans quatre dimensions au lieu de deux augmente a priori les gains en scalabilité. De nombreuses améliorations peuvent encore être effectuées à tous les niveaux du code, mais celui-ci fournit déjà suffisamment d'informations pour servir de modèle à une nouvelle mouture de Gysela, adaptée aux architectures *manycore*. La programmation par tâches s'avère une voie prometteuse pour combiner schémas adaptatifs, régulation dynamique de la charge et recouvrement calcul – communication.

Les travaux de cette thèse ont donc abouti à une amélioration des performances du code Gysela actuel et proposent des solutions concrètes pour mieux exploiter les architectures de calcul pré-*exascale*. Ces propositions ont été implémentées dans le prototype Gysela++ et des études de performances qui envisageaient de multiples paramètres ont été effectuées et permettent de constater un bon passage à l'échelle. Ces solutions pourraient également être bénéfiques pour de nombreux autres codes HPC qui visent des schémas adaptatifs en espace pour lesquels des problèmes de déséquilibrage de charge sont prévisibles. En effet, le gain de la programmation par tâches en termes de temps de calcul est moindre dans un code régulier où un parallélisme de boucle classique devrait être suffisant.

Perspectives

Le modèle de programmation par tâches et la mise en œuvre de communications asynchrones devaient permettre de réduire les phases de synchronisations inter-processus dans le code GYSELA, durant lesquels la plupart des cœurs étaient inactifs. Dans cette perspective, le prototype montre qu'il est possible de tirer entièrement profit du recouvrement calcul – communication, cela au prix d'un effort de développement d'autant plus important que les outils ne sont pas encore totalement mûrs. La génération de tâches en parallèle avec OpenMP ne permet pas encore un ordonnancement global et l'implémentation de la gestion des priorités (pour améliorer la prise en compte de tâches de différents poids) se fait attendre. Il n'est également pas possible de spécifier une liste de dépendances de taille variable (ce qui est possible avec un framework plus intrusif comme StarPU). Il semble toutefois pertinent d'envisager une utilisation analogue au modèle proposé par le prototype dans le futur code. En effet, le modèle de tâche est au cœur des travaux du consortium OpenMP et est en constante évolution, et ce d'autant que le modèle OpenMP reste peu intrusif et accessible. On pourrait également imaginer pour compléter ce modèle, une forme d'apprentissage de l'ordonnanceur pour des codes itératifs (itérations temporelles) qui génère à chaque fois des graphes de tâches similaires. L'ordonnanceur essaierait différents ordres de soumission à chaque tour afin de converger vers un ordre optimal. Quant aux communications asynchrones, il est également possible d'envisager, à la place d'un unique *thread* dédié, l'utilisation de communications multi-*threadées*. Le recouvrement calcul – communication pourrait alors être assuré par une bibliothèque MPI réellement asynchrone (NewMadeleine par exemple). L'efficacité du passage à l'échelle de ces communications sur un très grand nombre de nœuds reste une inconnue majeure.

Le maillage réduit ainsi que la structure de données en tuiles et sa distribution en cinq dimensions se montrent, eux, suffisamment pertinents pour envisager d'être introduits tels quels dans le futur code. Le maillage réduit a fait preuve de son efficacité au regard de la précision numérique et des temps d'exécution, ainsi que pour sa capacité à se focaliser sur des zones spécifiques. Sa complexité est bien répartie et encapsulée par la structuration des données sous forme de tuiles. Cette solution sera d'autant plus pertinente qu'il est envisagé d'avoir également un maillage non uniforme dans les directions de vitesses. D'autre part, les communications engendrées par la décomposition de domaine 5D passent mieux à l'échelle sur des cas conséquents que les changements de distribution de données utilisés auparavant. La possibilité de revenir à une interpolation 4D, envisagée au début de ces travaux, reste un point qui n'a pas été abordé ici et pour lequel il conviendrait d'effectuer une implémentation dans GYSELA++ afin de s'assurer de la compatibilité avec la distribution de données 4D.

D'autre part, il pourrait être envisagé d'utiliser des architectures hétérogènes, comme des accélérateurs, des cartes graphiques ou des GPGPUs, ce modèle par tâche se prêtant assez bien à cet exercice puisque le découpage en sous-ensemble de calcul à répartir entre les ressources de calcul est bien avancé. Il faut toutefois prévoir le développement de noyaux adaptés à chaque architecture pour chaque opérateur du code. La difficulté de la gestion et de la maintenance d'un tel code n'est pour l'heure pas évidente à estimer, sauf à se comparer à d'autres applications ayant franchi le pas. La bibliothèque StarPU, qui fournit un environnement complet pour mettre en place ce type de système, pourrait dans ce cas s'avérer une bonne alternative.

Bibliographie

[1] J. Abiteboul, G. Latu, V. Grandgirard, A. Ratnani, E. Sonnendrücker, and A. Strugarek. Solving the vlasov equation in complex geometries. In *ESAIM : Proceedings*, volume 32, pages 103–117. EDP Sciences, 2011.

[2] D. Akhmetova, R. Iakymchuk, O. Ekeberg, and E. Laure. Performance study of multithreaded mpi and openmp tasking in a large scientific code. In *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International*, pages 756–765. IEEE, 2017.

[3] K. E. Atkinson. *An introduction to numerical analysis*. John Wiley & Sons, 2008.

[4] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. Starpu : a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation : Practice and Experience*, 23(2) :187–198, 2011.

[5] O. Aumage, E. Brunet, N. Furmento, and R. Namyst. NewMadeleine : a Fast Communication Scheduling Engine for High Performance Networks. *Workshop on Communication Architecture for Clusters (CAC 2007), workshop held in conjunction with IPDPS 2007*, Mar. 2007.

[6] L. Bergstrom. Measuring numa effects with the stream benchmark. *arXiv preprint arXiv :1103.3225*, 2011.

[7] M. S. Birrittella, M. Debbage, R. Huggahalli, J. Kunz, T. Lovett, T. Rimmer, K. D. Underwood, and R. C. Zak. Intel® omni-path architecture : Enabling scalable, high performance fabrics. In *High-Performance Interconnects (HOTI), 2015 IEEE 23rd Annual Symposium on*, pages 1–9. IEEE, 2015.

[8] N. Bouzat, C. Bressan, V. Grandgirard, G. Latu, and M. Mehrenberger. Targeting realistic geometry in tokamak code GYSELA. *ESAIM, CEMRACS '16 proceedings*, 2016. submitted.

[9] N. Bouzat, F. Rozar, G. Latu, and J. Roman. A new parallelization scheme for the hermite interpolation based gyroaverage operator. In *16th International Symposium on Parallel and Distributted Computing (ISPDC)*, pages 70–77. IEEE, 2017.

[10] A. Brizard. Nonlinear gyrokinetic Maxwell-Vlasov equations using magnetic coordinates. *Journal of plasma physics*, 41(03) :541–559, 1989.

[11] A. J. Brizard and T. S. Hahm. Foundations of nonlinear gyrokinetic theory. *Reviews of modern physics*, 79(2) :421, 2007.

[12] K. Burrell. Effects of e× b velocity shear and magnetic shear on turbulence and transport in magnetic confinement devices. *Physics of Plasmas*, 4(5) :1499–1518, 1997.

[13] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. Parallel tiled qr factorization for multicore architectures. *Concurrency and Computation : Practice and Experience*, 20(13) :1573–1590, 2008.

[14] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir. Toward exascale resilience. *The International Journal of High Performance Computing Applications*, 23(4) :374–388, 2009.

[15] E. W. Cheney and W. A. Light. *A course in approximation theory*, volume 101. American Mathematical Soc., 2009.

[16] D. Clarke, Z. Zhong, V. Rychkov, and A. Lastovetsky. Fupermod : A framework for optimal data partitioning for parallel scientific applications on dedicated heterogeneous hpc platforms. In *International Conference on Parallel Computing Technologies*, pages 182–196. Springer, 2013.

[17] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Journal of parallel and distributed computing*, 7(2) :279–301, 1989.

[18] L. Dagum and R. Enon. OpenMP : an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1) :46–55, 1998.

[19] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 4. IEEE Press, 2008.

[20] R. C. Davidson. Theory of nonneutral plasmas. In *Reasearch supported by the National Science Foundation and US Navy. Reading, Mass., WA Benjamin, Inc.(Frontiers in Physics. Volume 43), 1974. 213 P.*, volume 43, 1974.

[21] A. Denis. pioman : a pthread-based Multithreaded Communication Engine. In *Euromicro International Conference on Parallel, Distributed and Network-based Processing*, Turku, Finland, Mar. 2015.

[22] F. Desprez. *Basic routines for scientific computing on distributed memory parallel computers.* Theses, Institut National Polytechnique de Grenoble - INPG, Jan. 1994.

[23] F. Desprez, P. Ramet, and J. Roman. Optimal grain size computation for pipelined algorithms. In *Euro-Par'96 Parallel Processing*, pages 165–172. Springer, 1996.

[24] F. Desprez and B. Tourancheau. Loccs : Low overhead communication and computation subroutines. *Future Generation Computer Systems*, 10(2) :279–284, 1994.

[25] R. Eigenmann and B. R. de Supinski. Openmp in a new era of parallelism. In *4th International Workshop, IWOMP*, volume 10, pages 100–110. Springer, 2008.

[26] Exascale computing research. `https://www.exascale-computing.eu`.

[27] Exascale computing project. `https://www.exascaleproject.org`.

[28] M. Fivaz, S. Brunner, G. De Ridder, O. Sauter, T. M. Tran, J. Vaclavik, L. Villard, and K. Appert. Finite element approach to global gyrokinetic particle-in-cell simulations using magnetic coordinates. *Computer physics communications*, 111(1) :27–47, 1998.

[29] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, et al. Open mpi : Goals, concept, and design of a next generation mpi implementation. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 97–104. Springer, 2004.

[30] X. Garbet, Y. Idomura, L. Villard, and T. Watanabe. Gyrokinetic simulations of turbulent transport. *Nuclear Fusion*, 50(4) :043002, 2010.

[31] A. Geist, W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, W. Saphir, T. Skjellum, and M. Snir. Mpi-2 : Extending the message-passing interface. In *European Conference on Parallel Processing*, pages 128–135. Springer, 1996.

[32] P. Ghendrih. La gazette du cines. `https://www.cines.fr/wp-content/uploads/2017/11/CINES_GAZETTE_GD2017.pdf`, pages 72-79, Décembre 2017.

[33] T. Görler, X. Lapillonne, S. Brunner, T. Dannert, F. Jenko, F. Merz, and D. Told. The global version of the gyrokinetic turbulence code GENE. *Journal of Computational Physics*, 230(18) :7053–7071, 2011.

[34] V. Grandgirard, J. Abiteboul, J. Bigot, T. Cartier-Michaud, N. Crouseilles, C. Erhlacher, D. Esteve, G. Dif-Pradalier, X. Garbet, P. Ghendrih, G. Latu, M. Mehrenberger, C. Norscini, C. Passeron, F. Rozar, Y. Sarazin, A. Strugarek, E. Sonnendrücker, and D. Zarzoso. A 5D gyrokinetic full-f global semi-Lagrangian code for flux-driven ion turbulence simulations. Submitted to CPC. Available online at `http://hal-cea.archives-ouvertes.fr/cea-01153011/file/article_GYSELA_2015.pdf`, May 2015.

[35] V. Grandgirard, M. Brunetti, P. Bertrand, N. Besse, X. Garbet, P. Ghendrih, G. Manfredi, Y. Sarazin, O. Sauter, E. Sonnendrücker, et al. A drift-kinetic semi-lagrangian 4D code for ion turbulence simulation. *Journal of Computational Physics*, 217(2) :395–423, 2006.

[36] V. Grandgirard et al. A 5D gyrokinetic full-f global semi-Lagrangian code for flux-driven ion turbulence simulations. *Computer Physics Communications*, 207 :35 – 68, 2016.

[37] V. Grandgirard, Y. Sarazin, P. Angelino, A. Bottino, N. Crouseilles, G. Darmet, G. Dif-Pradalier, X. Garbet, P. Ghendrih, S. Jolliet, et al. Global full-f gyrokinetic simulations of plasma turbulence. *Plasma Physics and Controlled Fusion*, 49(12B) :B173, 2007.

[38] G. Hager, G. Schubert, T. Schoenemeyer, and G. Wellein. Prospects for truly asynchronous communication with pure mpi and hybrid mpi/openmp on current supercomputing platforms. In *Cray Users Group Conference*, pages 23–26, 2011.

[39] T. S. Hahm. Nonlinear gyrokinetic equations for tokamak microturbulence. *Physics of Fluids (1958-1988)*, 31(9) :2670–2673, 1988.

[40] H. W. Hendel, B. Coppi, F. Perkins, and P. Politzer. Collisional effects in plasmas-drift-wave experiments and interpretation. *Physical Review Letters*, 18(12) :439, 1967.

[41] B. Holman and L. Kunyansky. A second-order finite difference scheme for the wave equation on a reduced polar grid, 2015.

[42] C.-C. Huang, Q. Chen, Z. Wang, R. Power, J. Ortiz, J. Li, and Z. Xiao. Spartan : A distributed array framework with smart tiling. In *USENIX Annual Technical Conference*, pages 1–15, 2015.

[43] Y. Idomura, H. Urano, N. Aiba, and S. Tokuda. Study of ion turbulent transport and profile formations using global gyrokinetic full-f vlasov simulation. *Nuclear Fusion*, 49(6) :065029, 2009.

[44] M.-C. Lai. A note on finite difference discretizations for poisson equation on a disk. *Numerical Methods for Partial Differential Equations : An International Journal*, 37(3) :199–203, 2001.

[45] G. Latu. *Contribution to high-performance simulation and highly scalable numerical schemes*. Research Director thesis, 2018.

[46] G. Latu, J. Bigot, N. Bouzat, J. Gimenez, and V. Grandgirard. Benefits of SMT and of parallel transpose algorithm for the large-scale GYSELA application. *ACM, PASC'16 proceedings*, 2016. to appear.

[47] G. Latu, V. Grandgirard, N. Crouseilles, and G. Dif-Pradalier. Scalable quasineutral solver for gyrokinetic simulation. In *International Conference on Parallel Processing and Applied Mathematics*, pages 221–231. Springer, 2011.

[48] G. Latu, V. Grandgirard, N. Crouseilles, and G. Dif-Pradalier. Scalable quasineutral solver for gyrokinetic simulation. In *PPAM (2)*, LNCS 7204, pages 221–231. Springer, 2011.

[49] J. Li. General explicit difference formulas for numerical differentiation. *Journal of Computational and Applied Mathematics*, 183(1) :29–52, 2005.

[50] Z. Lin and W. W. Lee. Method for solving the gyrokinetic poisson equation in general geometry. *Physical Review E*, 52(5) :5646, 1995.

[51] R. G. Littlejohn. Phase anholonomy in the classical adiabatic motion of charged particles. *Physical Review A*, 38(12) :6034, 1988.

[52] A. Mock. Subgridding scheme for fdtd in cylindrical coordinates. In *Progress In Electromagnetics Research Symposium Proceeding*, 2011.

[53] K. Mohseni and T. Colonius. Numerical treatment of polar coordinate singularities. *Journal of Computational Physics*, 157(2) :787 – 795, 2000.

[54] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz. Analysis of the clustering properties of the hilbert space-filling curve. *IEEE Transactions on knowledge and data engineering*, 13(1) :124–141, 2001.

[55] J. Nieplocha and R. Harrison. Shared memory programming in metacomputing environments : The global array approach. *The Journal of Supercomputing*, 11(2) :119–136, 1997.

[56] G. F. Pfister. An introduction to the Infiniband architecture. *High Performance Mass Storage and Parallel I/O*, 42, 2001.

[57] R. Preissl, A. Koniges, S. Ethier, W. Wang, and N. Wichmann. Overlapping communication with computation using openmp tasks on the gts magnetic fusion code. *Scientific Programming*, 18(3-4) :139–151, 2010.

[58] A. Quarteroni, R. Sacco, and F. Saleri. *Numerical mathematics*, volume 37. Springer Science & Business Media, 2010.

[59] A. Querteroni, R. Sacco, and F. Selra. *Numerical Mathematics*, volume 37. Springer Science & Business Media, 2010.

[60] M. J. Quinn and P. J. Hatcher. On the utility of communication–computation overlap in data-parallel programs. *Journal of Parallel and Distributed Computing*, 33(2) :197–204, 1996.

[61] R. Rabenseifner. Hybrid parallel programming on hpc platforms. In *proceedings of the Fifth European Workshop on OpenMP, EWOMP*, volume 3, pages 185–194, 2003.

[62] F. Rozar. *Contributions à l'amélioration de l'extensibilité de simulations parallèles de plasmas turbulents*. PhD thesis, Bordeaux, 2015.

[63] F. Rozar, C. Steiner, G. Latu, M. Mehrenberger, V. Grandgirard, J. Bigot, T. Cartier-Michaud, and J. Roman. Optimization of the gyroaverage operator based on hermite interpolation. *arXiv preprint arXiv :1602.02886*, 2016.

[64] L. I. Rudakov and R. Z. Sagdeev. On the instability of a nonuniform rarefied plasma in a strong magnetic field. In *Soviet Physics Doklady*, volume 6, page 415, 1961.

[65] O. Runborg. Notes on polynomial interpolation. *Kungliga Tekniska Högskolan, 2d1250, Tillämpade numeriska metoder II*, 2003.

[66] J. C. Sancho, K. J. Barker, D. J. Kerbyson, and K. Davis. Quantifying the potential benefit of overlapping communication and computation in large-scale scientific applications. In *SC 2006 Conference, Proceedings of the ACM/IEEE*, pages 17–17. IEEE, 2006.

[67] Y. Sarazin, V. Grandgirard, E. Fleurence, X. Garbet, P. Ghendrih, P. Bertrand, and G. Depret. Kinetic features of interchange turbulence. *Plasma physics and controlled fusion*, 47(10) :1817, 2005.

[68] E. Sonnendrücker, J. Roche, P. Bertrand, and A. Ghizzo. The semi-lagrangian method for the numerical resolution of the vlasov equation. *Journal of computational physics*, 149(2) :201–220, 1999.

[69] C. Steiner. *Numerical computation of the gyroaverage operator and coupling with the Vlasov gyrokinetic equations*. PhD thesis, Université de Strasbourg, IRMA, 2014.

[70] C. Steiner, M. Mehrenberger, N. Crouseilles, V. Grandgirard, G. Latu, and F. Rozar. Gyroaverage operator for a polar mesh. *The European Physical Journal D*, 69(1) :1–16, 2015.

[71] J. Stoer and R. Bulirsch. *Introduction to numerical analysis*, volume 12. Springer Science & Business Media, 2013.

[72] D. Unat, C. P. Chan, W. Zhang, J. Bell, and J. Shalf. Tiling as a durable abstraction for parallelism and data locality. Technical report, Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 2013.

[73] S. Wienke, P. Springer, C. Terboven, and D. Mey. Openacc–first experience with real-world applications. In *European Conference on Parallel Processing*, pages 859–870. Springer, 2012.

[74] M. Wittmann, G. Hager, T. Zeiser, and G. Wellein. Asynchronous MPI for the Masses. *CoRR*, abs/1302.4280, 2013.

[75] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. *ACM Sigplan Notices*, 26(6) :30–44, 1991.

Annexes

Sommaire

A Reduced mesh with uniform density

In order to have a distribution of points on the plane which is almost uniform, we can suppose that $\Delta r \simeq r_i \Delta\theta_{[i]}$ for every i and calculate the number of point on each radius.

So $\Delta\theta_{[i]} = r_i\Delta\theta_{[i]} = \frac{2\pi r_i}{N_{\theta[i]}}$, and we have to find $N_{\theta[i]}$ for $i \in [\![0, N_r - 1]\!]$. We then have

$$\begin{aligned}\Delta r = \Delta\theta_{[i]} &\Leftrightarrow \frac{r_{\max}}{N_r - \frac{1}{2}} = r_{\max}\frac{2\pi}{N_{\theta[i]}}\frac{i-\frac{1}{2}}{N_r - \frac{1}{2}} \\ &\Leftrightarrow N_{\theta[i]} = 2\pi(i - \frac{1}{2}), \quad i \in [\![0, N_r - 1]\!]. \end{aligned} \tag{11.1}$$

Of course, with this choice, $N_{\theta[i]}$ may not be an integer and a fortiori not a power of two which is the classical choice. So, one way to choose the sequence $N_{\theta[i]}$ is to define $\ell_i \in \mathbb{N}$ by

$$2^{\ell_i - 1} < (i - 1/2)2\pi \leq 2^{\ell_i}, \quad i = 1, \ldots, N_r,$$

and take $N_{\theta[i]} = 2^{\ell_i}$. In this way, $N_{\theta[i]}$ is always a power, while being the smallest possible satisfying $\Delta\theta^i < \Delta r$. Note that

$$2^{\ell_i - 1} < (i - 1/2)2\pi \leq 2^{\ell_i} \Leftrightarrow \ell_i - 1 < \frac{\ln((i - 1/2)2\pi)}{\ln(2)} \leq \ell_i,$$

and thus ℓ_i is explicitly given by

$$\ell_i = \lceil \frac{\ln((i - 1/2)2\pi)}{\ln(2)} \rceil.$$

It turns out that generally, several i lead to the same ℓ_i, so it may be useful, for having a more compact representation to know how many times the numbers ℓ_i are repeated.
So, for each $\ell \in \mathbb{N}$, we look for the indices i such that

$$2^{\ell-1} < (i - 1/2)2\pi \leq 2^{\ell} \Leftrightarrow \frac{2^{\ell-1}}{2\pi} + \frac{1}{2} < i \leq \frac{2^{\ell}}{2\pi} + \frac{1}{2},$$

this leads to

$$\lfloor \frac{2^{\ell-1}}{2\pi} + \frac{1}{2} \rfloor + 1 \leq i \leq \lfloor \frac{2^{\ell}}{2\pi} + \frac{1}{2} \rfloor,$$

as i is an integer. Writing $m_\ell = \lfloor \frac{2^\ell}{2\pi} + \frac{1}{2} \rfloor, \ell \in \mathbb{N}$, we have

$m_0 = m_1 = 0,\ m_2 = m_3 = 1,\ m_4 = 3,\ m_5 = 5,\ m_6 = 10,\ m_7 = 20,\ m_8 = 41,\ m_9 = 81,\ m_{10} = 163$

and further values are

$m_{11} = 326,\ m_{12} = 652,\ m_{13} = 1304,\ m_{14} = 2608,\ m_{15} = 5215,\ m_{16} = 10430,\ m_{17} = 20861.$

We then have

$$N_{\theta[i]} = 2^\ell, \quad i = m_{\ell-1} + 1, \ldots, m_\ell.$$

B Résolution tridiagonale avec l'algorithme de Thomas

La résolution de système tridiagonal est au cœur de cet opérateur de Poisson, aussi bien pour le calcul du potentiel moyen que pour celui du potentiel dans l'espace de Fourier. L'algorithme de Thomas utilisé présente un coût linéaire avec le rang du système et fonctionne pour des nombres réels et complexes. Basée sur le principe de l'élimination gaussienne, cette résolution n'est stable que pour les matrices à diagonale dominante (ce qui est le cas pour nos problèmes comme nous le verrons par la suite) ou symétriques définies positives. Étant donnée la matrice tridiagonale M_T (11.2) avec $a_1 = 0$, $c_N = 0$ et $\forall i \in [\![1, N]\!],\ |b_i| \geq |a_i| + |c_i|$, l'algorithme de Thomas permet de résoudre l'équation $M_T X = Y$ où le vecteur $X = (x_1, \ldots, x_N)^\intercal$ est l'inconnu et $Y = (y_1, \ldots, y_N)^\intercal$ est connu.

$$M_T = \begin{pmatrix} b_1 & c_1 & 0 & \ldots & \ldots & 0 \\ a_2 & b_2 & c_2 & \ddots & & \vdots \\ 0 & a_3 & b_3 & c_3 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & a_{N-1} & b_{N-1} & c_{N-1} \\ 0 & \ldots & \ldots & 0 & a_N & b_N \end{pmatrix} \tag{11.2}$$

L'algorithme 12 présente le solveur tridiagonal de Thomas. Le lecteur pourra se référer à [58] pour la preuve de l'algorithme.

Pour nos deux problèmes (calcul du potentiel moyen et calcul du potentiel), il est possible de montrer que les systèmes générés sont à diagonale dominante. Seule la preuve pour le système du calcul du potentiel moyen sera présentée, le système général pour le calcul du potentiel étant similaire avec des termes complexes plus complexes. Les termes des diagonales extérieures (facteurs de p_{i-1} et p_{i+1} dans l'équation (6.10)) s'expriment ainsi :

$$a_i = \frac{-1}{\Delta r^2} + [\frac{1}{r_i} + \frac{1}{n_0(r_i)} \frac{\partial n_0(r_i)}{\partial r}] \cdot \frac{1}{\Delta r}, \quad c_i = \frac{-1}{\Delta r^2} - [\frac{1}{r_i} + \frac{1}{n_0(r_i)} \frac{\partial n_0(r_i)}{\partial r}] \cdot \frac{1}{\Delta r}.$$

Considérons les différents membres de ces équations. Étant donnés les paramètres physiques de la simulation, on sait que

$$\frac{1}{r_i} \gg \frac{1}{n_0(r_i)} \frac{\partial n_0(r_i)}{\partial r}.$$

Algorithme 12 : Algorithme de Thomas pour la résolution de systèmes tridiagonaux à diagonale dominante.

Entrée : N, M_T, Y
Sorties : X

$\gamma_1 \leftarrow 0$;
$\beta_1 \leftarrow 0$;
pour $i = 1 \,..\, N$ **faire**
  $\gamma_{i+1} \leftarrow \dfrac{-c_i}{a_i\gamma_i + b_i}, \quad \beta_{i+1} \leftarrow \dfrac{y_i - a_i\beta_i}{a_i\gamma_i + b_i}$
$x_N = \beta_{N+1}$;
pour $i = N{-}1 \,..\, 0$ **faire**
  $x_i = \gamma_{i+1}x_{i+1} + \beta_{i+1}$;

On a alors

$$[\frac{1}{r_i} + \frac{1}{n_0(r_i)}\frac{\partial n_0(r_i)}{\partial r}] \cdot \frac{1}{\Delta r} \approx \frac{1}{(i+\frac{1}{2})\Delta r} \cdot \frac{1}{\Delta r},$$

qui est inférieur à $\frac{1}{\Delta r^2}$ pour $i > 0$. D'autre part, on sait que si deux nombres peuvent s'écrirent $a_i = d + e$ et $c_i = d - e$, avec d et e des nombres réels, on a $|a_i| + |c_i| = 2|d|$ si $d \geq e$ et $|a_i| + |c_i| = 2|e|$ sinon. Ainsi, pour $i > 0$,

$$\frac{1}{\Delta r^2} \geq [\frac{1}{r_i} + \frac{1}{n_0(r_i)}\frac{\partial n_0(r_i)}{\partial r}] \cdot \frac{1}{\Delta r}.$$

et $|a_i| + |c_i| = \frac{2}{\Delta r^2} = |b_i|$. Dans le cas où $i = 0$, $a_0 = 0$ et $b_0 = -c_0$, d'où $|a_0| + |c_0| = |b_0|$. Nos systèmes sont donc à diagonale dominante et l'algorithme de Thomas peut être utilisé pour les résoudre.

Les opérateurs de Vlasov et de Poisson, centraux au modèle gyrocinétique, bénéficient d'une implémentation optimisée, bien qu'elle puisse encore faire l'objet de nombreuse amélioration. Le chapitre suivant est centré sur l'évaluation de l'ensemble des performances du code GYSELA++, du modèle de programmation par tâche et de communication asynchrone aux optimisations apportées aux différents opérateurs.

Université de Strasbourg

Nicolas BOUZAT

Inria informatics mathematics

Algorithmes à grain fin et schémas numériques pour des simulations exascales de plasmas turbulents

Résumé

Les architectures de calcul haute performance les plus récentes intègrent de plus en plus de nœuds de calcul qui intègrent eux-mêmes plus de cœurs. Les bus mémoires et les réseaux de communication sont soumis à un niveau d'utilisation critique. La programmation parallèle sur ces nouvelles machines nécessite de porter une attention particulière à ces problématiques pour l'écriture de nouveaux algorithmes. Nous analysons dans cette thèse un code de simulation de turbulences de plasma et proposons une refonte de la parallélisation de l'opérateur de gyromoyenne plus adapté en termes de distribution de données et bénéficiant d'un schéma de recouvrement calcul – communication efficace. Les optimisations permettent un gain vis-à-vis des coûts de communication et de l'empreinte mémoire. Nous étudions également les possibilités d'évolution de ce code à travers l'étude d'un prototype utilisant un modèle programmation par tâche et un schéma de communication asynchrone adapté. Cela permet d'atteindre un meilleur équilibrage de charge afin de maximiser le temps de calcul et de minimiser les communications entre processus. Un maillage réduit adaptatif en espace est proposé permettant de diminuer le nombre de points sans pour autant perdre en précision, mais ajoutant de fait une couche supplémentaire de complexité. Ce prototype explore également une nouvelle distribution de données ainsi qu'un maillage en géométrie complexe adapté aux nouvelles configurations des tokamaks. Les performances de différentes optimisations sont étudiées et comparées avec le code préexistant et un cas dimensionnant sur un grand nombre de cœurs est présenté.

Abstract

Recent high performance computing architectures come with more and more cores on a greater number of computational nodes. Memory buses and communication networks are facing critical levels of use. Programming parallel codes for those architectures requires to put the emphasize on those matters while writing tailored algorithms. In this thesis, a plasma turbulences simulation code is analyzed and its parallelization is overhauled. The gyroaverage operator benefits from a new algorithm that is better suited with regard to its data distribution and that uses a computation – communication overlapping scheme. Those optimizations lead to an improvement by reducing both execution times and memory footprint. We also study new designs for the code by developing a prototype based on task programming model and an asynchronous communications scheme. It allows us to reach a better load balancing and thus to achieve better execution times by minimizing communication overheads. A new reduced mesh is introduced, shrinking the overall mesh size while keeping the same numerical accuracy but at the expense of more complex operators. This prototype also uses a new data distribution and twists the mesh to adapt to the complex geometries of modern tokamak reactors. Performance of the different optimizations is studied and compared to that of the current code. A case scaling on a large number of cores is given.