Contributions à l'amélioration de l'extensibilité de simulations parallèles de plasmas turbulents

Fabien Rozar

▶ **To cite this version:**

Fabien Rozar. Contributions à l'amélioration de l'extensibilité de simulations parallèles de plasmas turbulents. Modélisation et simulation. Université de Bordeaux, 2015. Français. NNT : 2015BORD0211 . tel-01271032

HAL Id: tel-01271032

https://theses.hal.science/tel-01271032v1

Submitted on 8 Feb 2016

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

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

université de BORDEAUX

THÈSE

PRÉSENTÉ À

L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

Par **Fabien ROZAR**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Contributions à l'amélioration de l'extensibilité de simulations parallèles de plasmas turbulents

---

Soutenue le : 05 Novembre 2015

Devant la commission d'examen composée de :

Olivier COULAUD	Directeur de recherche, Inria	Président du jury
Jean-François MEHAUT	Professeur, Univ. de Grenoble 1	Rapporteur
Boniface NKONGA	Professeur, Univ. de Nice Sophia-Antipolis	Rapporteur
Alberto BOTTINO	Chargé de recherche, IPP Garching, Allemagne	Examinateur
Alain KETTERLIN	Maître de conférence, Univ. de Strasbourg	Examinateur
Guillaume LATU	Ingénieur-chercheur, CEA Cadarache	Co-encadrant
Jean ROMAN	Professeur, Inria et Bordeaux INP	Directeur de thèse

---

cea DE LA RECHERCHE À L'INDUSTRIE

Inria INVENTEURS DU MONDE NUMÉRIQUE

– 2015 –

Résumé

Les besoins en énergie dans le monde sont croissants alors que les ressources nécessaires pour la production d'énergie fossile s'épuisent d'année en année. Un des moyens alternatifs pour produire de l'énergie est la fusion nucléaire par confinement magnétique. La maîtrise de cette réaction est un défi et constitue un domaine actif de recherche.

Pour améliorer notre connaissance des phénomènes qui interviennent lors de la réaction de fusion, deux approches sont mises en œuvre : l'expérience et la simulation. Les expériences réalisées grâce aux *Tokamaks* permettent de prendre des mesures. Ceci nécessite l'utilisation des technologiques les plus avancées. Actuellement, ces mesures ne permettent pas d'accéder à toutes échelles de temps et d'espace des phénomènes physiques. La simulation numérique permet d'explorer ces échelles encore inaccessibles par l'expérience. Les ressources matérielles qui permettent d'effectuer des simulations réalistes sont conséquentes. L'usage du calcul haute performance (*High Performance Computing* – HPC) est nécessaire pour avoir accès à ces simulations. Ceci se traduit par l'exploitation de grandes machines de calcul aussi appelées *supercalculateurs*.

Les travaux réalisés dans cette thèse portent sur l'optimisation de l'application Gysela qui est un code de simulation de turbulence de plasma. L'optimisation d'un code de calcul scientifique vise classiquement l'un des trois points suivants : (*i*) la simulation de plus grand domaine de calcul, (*ii*) la réduction du temps de calcul et (*iii*) l'amélioration de la précision des calculs.

La première partie de ce manuscrit présente les contributions concernant la simulation de plus grand domaine. Comme beaucoup de codes de simulation, l'amélioration de la précision de la simulation est souvent synonyme de raffinement du maillage. Plus un maillage est fin, plus la consommation mémoire est grande. De plus, durant ces dernières années, les supercalculateurs ont eu tendance à disposer de moins en moins de mémoire par cœur de calcul. Pour ces raisons, nous avons développé une bibliothèque, la libMTM (*Modeling and Tracing Memory*), dédiée à l'étude précise de la consommation mémoire d'applications parallèles. Les outils de la libMTM ont permis de réduire la consommation mémoire de Gysela et d'étudier sa scalabilité. À l'heure actuelle, nous ne connaissons pas d'autre outil qui propose de fonctionnalités équivalentes permettant une étude précise de la scalabilité mémoire.

La deuxième partie de ce manuscrit présente les travaux concernant l'optimisation du temps d'exécution et l'amélioration de la précision de l'opérateur de gyromoyenne. Cet opérateur est fondamental dans le modèle gyrocinétique qui est utilisé par l'application Gysela. L'amélioration de la précision vient d'un changement de la méthode de calcul : un schéma basé sur une interpolation de type Hermite vient remplacer l'approximation de Padé. Il s'avère que cette nouvelle version de l'opérateur est plus précise mais aussi plus coûteuse en terme de temps de calcul que l'opérateur existant. Afin que les temps de simulation restent raisonnables, différentes optimisations ont été réalisées sur la nouvelle méthode de calcul pour la rendre très compétitive. Nous avons aussi développé une version parallélisée en MPI du nouvel opérateur de gyromoyenne. La bonne scalabilité de cet opérateur de gyromoyenne permettra, à terme, de réduire des coûts en communication qui sont pénalisants dans une application parallèle comme Gysela.

Mots clés : Calcul haute performance, Scalabilité mémoire, Mathématiques appliquées, Physique des plasmas.

Equipe-projet commune HiePACS, Inria Bordeaux – Sud-Ouest
LaBRI (UMR CNRS 5800), Université de Bordeaux, Bordeaux INP
Institut de Recherche sur la Fusion par confinement Magnétique, CEA Cadarache
Maison de la Simulation, USR 3441, CEA / CNRS / Inria / Univ. Paris-Sud / Univ. Versailles

Abstract

Energy needs around the world still increase despite the resources needed to produce fossil energy drain off year after year. An alternative way to produce energy is by nuclear fusion through magnetic confinement. Mastering this reaction is a challenge and represents an active field of the current research.

In order to improve our understanding of the phenomena which occur during a fusion reaction, experiment and simulation are both put to use. The performed experiments, thanks to *Tokamaks*, allow some experimental reading. The process of experimental measurements is of great complexity and requires the use of the most advanced available technologies. Currently, these measurements do not give access to all scales of time and space of physical phenomenon. Numerical simulation permits the exploration of these scales which are still unreachable through experiment. An extreme computing power is mandatory to perform realistic simulations. The use of High Performance Computing (HPC) is necessary to access simulation of realistic cases. This requirement means the use of large computers, also known as *supercomputers*.

The works realized through this thesis focuses on the optimization of the GYSELA code which simulates a plasma turbulence. Optimization of a scientific application concerns mainly one of the three following points : (*i*) the simulation of larger meshes, (*ii*) the reduction of computing time and (*iii*) the enhancement of the computation accuracy.

The first part of this manuscript presents the contributions relative to simulation of larger mesh. Alike many simulation codes, getting more realistic simulations is often analogous to refine the meshes. The finer the mesh the larger the memory consumption. Moreover, during these last few years, the supercomputers had trend to provide less and less memory per computer core. For these reasons, we have developed a library, the LIBMTM (*Modeling and Tracing Memory*), dedicated to study precisely the memory consumption of parallel softwares. The LIBMTM tools allowed us to reduce the memory consumption of GYSELA and to study its scalability. As far as we know, there is no other tool which provides equivalent features which allow the memory scalability study.

The second part of the manuscript presents the works relative to the optimization of the computation time and the improvement of accuracy of the gyroaverage operator. This operator represents a corner stone of the gyrokinetic model which is used by the GYSELA application. The improvement of accuracy emanates from a change in the computing method : a scheme based on a $2D$ Hermite interpolation substitutes the Padé approximation. Although the new version of the gyroaverage operator is more accurate, it is also more expensive in computation time than the former one. In order to keep the simulation in reasonable time, different optimizations have been performed on the new computing method to get it competitive. Finally, we have developed a MPI parallelized version of the new gyroaverage operator. The good scalability of this new gyroaverage computer will allow, eventually, a reduction of MPI communication costs which are penalizing in GYSELA.

Keywords : High performance computing, Memory scalability, Applied mathematics, Plasma physics.

Equipe-projet commune HiePACS, Inria Bordeaux – Sud-Ouest
LaBRI (UMR CNRS 5800), Université de Bordeaux, Bordeaux INP
Institut de Recherche sur la Fusion par confinement Magnétique, CEA Cadarache
Maison de la Simulation, USR 3441, CEA / CNRS / Inria / Univ. Paris-Sud / Univ. Versailles

Remerciements

Dans un souci de rigueur, je commencerai dans cette partie par remercier chaleureusement mon jury de thèse. Merci à Boniface Nkonga pour son regard vif et éclairé son mon manuscrit et pour ses remarques toujours pertinentes. Merci à Jean-François Méhaut pour l'intérêt qu'il a porté à mes travaux et pour les discussions pleines d'idées qui ont eu lieu avant et durant la soutenance. Je remercie Alberto Bottino de s'être intéressé à mon travail et d'avoir fait le déplacement pour contribuer à mon jury. De même, merci à Alain Ketterlin pour les courtes (peut-être trop courtes) discussions qu'on a pu avoir durant ma thèse et pour avoir contribué à mon jury. Je tiens à remercier particulièrement Olivier Coulaud, président du jury, qui a accepté très rapidement de faire partie de mon jury, sans qui les choses auraient été plus compliquées. J'ai eu l'honneur d'avoir comme directeur de thèse Jean Roman que j'ai souvent sollicité durant ces trois ans et que je remercie pour son regard aguerri dans le domaine du calcul intensif et pour sa bienveillance. Je remercie bien évidemment mon encadrant de thèse, Guillaume Latu, pour s'être si bien occupé de moi durant mon doctorat et mon stage de fin d'étude (parfois jusque tard le soir, parfois durant les weekends et même après la fin de mon doctorat), sans sa bonne humeur et son aide, mon travail en aurait certainement pâtit. Je le remercie aussi de m'avoir fait participer à deux CEMRACS qui ont joué un rôle important pour le déroulement de mon doctorat. Je n'oublie pas les invités, à savoir Michel Méhrenberger avec qui j'ai travaillé en étroite collaboration durant la seconde partie de ma thèse, merci de m'avoir enrôlé si rapidement aux travaux sur la gyromoyenne (clin d'œil spécial à Christophe Steiner), et Virginie Grandgirard que j'appelle parfois "maman Gysela" avec qui j'ai eu le plaisir de collaborer durant l'intégralité de mon doctorat (je n'oublie pas bien sûr les joyeuses soirées villelauraines auxquelles tu m'as convié ;-)).

La thèse a été une grande aventure pour ma part, et j'ai beaucoup appris en chemin. Ce chemin aurait été bien différent si je n'avais pas eu la chance de croiser tout au long de ma route des collègues et amis forts agréables. Je remercie Damien Estève (alias Daminou) de m'avoir supporté dans le bureau durant ma première année de thèse et de m'avoir tant fait rigoler – à c'qui paraît, à c'qui paraît – (il comprendra :-p). La superbe ambiance qui règne dans le laboratoire IRFM existe grâce aux nombreux non permanents et jeunes permanents : Thomas Cartier-Michaud (alias TC), Hugo Arnichand (alias Boby), Alexandre Fil (alias Filou), Hugo Bufferand, Laurent Choné, Clothilde Colin-Bellot (du M2P2 de Marseille, mais je te compte quand même), Jean-Baptiste Girardo, Romain Futtersack, Claudia Norcini, Walid Helou, Laurent Valade (l'exceptionnel ;)), Jae-heon Ahn, Olivier Février, Cristian Sommariva, Laura Mendoza (de Garching), Camille Baudoin, Nicolas Nace, merci à vous tous pour les nombreuses soirées partagées ^^.

L'ambiance chaleureuse de l'IRFM ne serait pas la même sans ses membres permanents. Je commencerai par remercier Chantal Passeron qui a toujours été là quand j'avais des soucis informatiques (et oui, ça arrive plus fréquemment qu'on ne le croit). Je ne peux pas oublier les experts physiciens que sont Philippe Ghendrih, Xavier Garbet et Yanick Sarazin pour qui j'ai le plus grand respect. Je remercie Patrick Tamin pour sa combativité sportive (je pense que je dois attendre ta retraite au badminton pour pouvoir te battre ^^'). Je remercie Guilhem Dif-Pradalier, certes pour ses conseils et sa vision éclairée des thématiques abordées par ma thèse, mais surtout pour sa contribution exceptionnelle le jour de mon pot de thèse ;-).

Enfin, je remercie Jennyfer Obry et Olivier Maegey qui ont été mes colocataires sur Aix en Provence durant un an et demi. Je n'oublierai pas non plus ta contribution indispensable Olivier le jour–J de ma soutenance pour la réussite de mon pot de thèse ;-). Lors de mes séjours à Cadarache durant la seconde partie de ma thèse, j'ai souvent été en squatte à la coloc formée de

Hugo (Boby), Damien (Daminou), Katia, Davide Galassi et Clément Nguyen Than Dao : merci à vous de m'avoir accueilli si souvent et toujours dans la bonne humeur ^^.

Quand je pense au CEA Cadarache, je ne peux pas m'empêcher d'avoir une pensée pour ce qui y sont passés et que j'ai eu la chance de croiser : Jonathan Faustin, Alvin Sashalanaik, Olivier Thomine, Victor Moncada, Jeremie Abiteboul, Antoine Merle, Antoine Strugarek, Didier Vézinet, Sabine Cockenpot, David Zarzoso, Grégoire Hornung, Timothée Nicolas, Jorge Morales, Yue Dong, François Orain, Farah Hariri, à vous tous, merci d'avoir contribué à la vie à l'extérieur du laboratoire ("ce soir, un p'tit sextius ou pas ?") :D.

Durant la seconde partie de ma thèse, j'ai eu la chance de travailler à la Maison de la Simulation (CEA Saclay) et surtout de rencontrer des gens sympathiques travaillant sur des thématiques diverses. Parmi mes collègues de la MdlS, je remercie tout d'abord Julien Bigot pour toutes ses contributions à l'amélioration des conditions de développement de Gysela (c'était vraiment une bouffée d'oxygène) et pour toutes ses qualités dans la vie à l'extérieur du labo. Thank you Ralitsa Ivanova to give me the opportunity to discover the squash, a sport I really love to play with you partner ;-) and thank you to have made MdlS a really funny place to work :-). Merci à Sébastien Cayrols pour tout, pour les débats techniques, pour les challenges, pour ton humour, pour avoir joué au taxi pour moi... franchement je regrette que tu n'aies pas été plus souvent là à la MdlS parce qu'on se marrait vachement bien quand même. J'en connais une autre qui a joué au taxi pour moi, et oui c'est de toi que je parle Florence Drui, merci pour ça, et surtout merci d'avoir si souvent déliré avec moi :D. Même si tu n'étais plus à la MdlS à la fin de mon doctorat Maxime Delorme, j'ai une pensée pour toi et je garde un très bon souvenir de nos échanges (souvent techniques) et de nos délires (souvent nawak :-p). Merci à Adeline Holton pour sa bonne humeur, son sourire et ses conseils capillaires. Merci à Matthieu Haefele qui m'a souvent donné de très bons conseils biens utiles pour la progression de mon travail. Au même titre je remercie Samuel Kock qui m'a vraiment fourni un support en mathématiques de très haut standing et tout simplement pour sa gentillesse (tu as réellement joué un rôle de co-encadrant à des moments clé de ma thèse). Thomas Dufaud, malgré le fait qu'on n'ait pas (encore) pu boire ce dernier verre ensemble (on pourra toujours trouver une autre occasion ;-)), je te remercie de m'avoir aidé sur un bout de ce manuscrit, même si ce n'était peut-être pas grand chose, ça m'a fait du bien psychologiquement ; et merci pour les nombreux débats (souvent non terminés) qu'on a eus. Quand je pense à Maximilien Levesque j'ai tout de suite le sourire, tu es un vrai scientifique au sens noble du terme, toujours à ouvrir ses horizons, c'était très instructif de discuter avec toi. Je remercie Thibault Gasc pour ses bons tuyaux et sa contribution à la bonne ambiance du labo. Récemment arrivé, Yacine Ould-Rouis est un sacré personnage, toujours de bonne humeur même quand il est en grande difficulté, un compagnon à toujours avoir à ses côtés. Parmi les nouvelles recrues mais déjà très actives, je remercie Maxime Stauffert, que j'ai que trop peu eu l'occasion de voir, pour sa curiosité et la franche camaraderie qui s'est installée entre nous. Parmi ceux qui ne sont plus de la MdlS, je remercie Rehan Malak et Frédérique Dauvergne pour les sessions escalade toujours accompagnées de la bière de la récompense ^^.

Je remercie tous les collègues et anciens collègues du labo : Pierre-Elliott Bécue, Lu Ding, Nicolas Doucet, Mohamed Gaalich, Giorgio Giogiani, Esra Karakas (que je n'ai pas assez vue), Rudi Leroy, France Boillod-Cerneux, Jeaniffer Vides, Sophie Félix et Yunsong Wang, pour m'avoir fait vivre des moments bien joyeux. La MdlS ne serait pas ce qu'elle est sans ses membres permanents : Edouard Audit, Julien Dérouillat, Michel Kern, Pierre Kestener, Daniel Borgis, Martial Mancip, Philippe Bigeon, Mathieu Salanne et Pascal Tremblin qui ont été là quand j'avais besoin d'eux. Enfin, je remercie tous les membres (actuels et anciens) du secrétariat, à savoir Valérie Belle, Aurélie Monteiro, Adeline Holton (à nouveau) et Layla Tarondeau pour m'avoir guidé à travers les méandres administratifs du CEA et pour leur disponibilité sans faille.

Je fais un clin d'œil spécial à Cyril Vuagnoux pour les nombreuses bières qu'on a partagées ensemble, à Marwa Sridi pour tout ce que tu as fait pour moi et à Stéphanie Hoareau pour l'orthographe.

Ayant mon directeur de thèse chez Inria Bordeaux–Sud-Ouest, j'ai eu l'occasion de revenir dans la ville de Bordeaux, là où j'ai fait mon école d'ingénieur. Merci à tous les membres de l'équipe HiePACS qui m'ont intégré très vite dans le groupe et avec qui je garde de très bons souvenirs. P'tit clin d'œil spécial à Jean-Marie Couteyen que j'ai croisé durant ma dernière année d'école d'ingénieur et avec qui j'ai appris (je te souhaite bon courage pour le petit bout de chemin qu'il te reste à faire avant la soutenance de thèse ;-)). Je n'oublie pas M. Nicolas Bouzat, que j'ai encadré durant son stage et qui est maintenant en thèse (la relève est assurée), merci d'avoir engagé tes forces sur le sujet que j'avais proposé, et bon courage jeune padawan. J'adresse un remerciement spécial à Brice Goglin pour les renseignements très précieux qu'il m'a fournis sur des comportements obscures du système :-).

Je terminerai cette section par des remerciements spéciaux à des personnes que j'ai croisées trop rapidement. Pour son cours et pour ses nombreux articles d'excellente qualité dans le domaine de la fusion par confinement magnétique (pédagogique pour la plupart), je remercie Eric Sonnendrücker grâce à qui ma compréhension de la physique a réellement évolué. Enfin, pour sa sympathie et la superbe introduction dans son manuscrit (c'est choupiment dit, non :-p), je remercie Céline Caldini-Queiros.

Ce chemin n'aurait pas été possible sans le support et le soutien de ma famille au quotidien et que je remercie de m'avoir accompagné tout au long de cette route.

Table des matières

Table des figures

Chapitre 1

Introduction et contexte de l'étude

Sommaire

Les travaux réalisés au cours de cette thèse sont en lien avec l'étude de la fusion nucléaire contrôlée par confinement magnétique. La simulation haute performance d'un plasma de tokamak nécessite un environnement interdisciplinaire qui regroupe la physique des plasmas, les mathématiques appliquées et l'informatique du calcul haute performance. Ce chapitre introduit le contexte scientifique de ces travaux.

La section 1.1 donne une vue d'ensemble des installations expérimentales et des fondements théoriques qui ont été développés par la communauté scientifique pour étudier le comportement d'un plasma chaud au sein d'un tokamak. La section 1.2 introduit les *supercalculateurs* et le modèle de programmation parallèle utilisé par GYSELA. Les architectures des supercalculateurs sont complexes et motivent le développement de modèles de programmation adaptés pour les exploiter au mieux. La section 1.3 décrit le code GYSELA qui est une application de simulation numérique massivement parallèle. Une description du mouvement de giration d'une particule chargée dans un champ magnétique est donnée. Ce phénomène est à l'origine du modèle *gyrocinétique* qui est utilisé par GYSELA. Les problématiques relatives au calcul haute performance, tels que la parallélisation ainsi que la consommation mémoire de GYSELA, sont introduites dans cette section. Aujourd'hui, beaucoup de domaines scientifiques utilisent la simulation numérique et ont recours au calcul haute performance, ce qui en fait un outil transverse. Ce chapitre se conclut sur le positionnement des travaux qui ont été menés dans le cadre de cette thèse.

1.1 Contexte physique

Les besoins en énergie dans le monde sont de plus en plus importants et les énergies fossiles s'épuisent d'année en année alors qu'elles représentent actuellement la principale source énergétique. Dans ce contexte, la fusion nucléaire contrôlée offre une source alternative d'énergie prometteuse. En effet, dans le cas où le processus de fusion serait maîtrisé, cela permettrait de produire de l'énergie de façon pérenne et sûre.

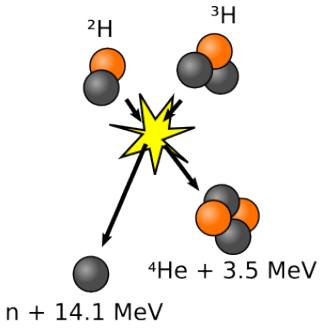


FIGURE 1.1 – Réaction de fusion entre Deutérium et Tritium. Extrait de : http://www.fusenet.eu/node/36

La réaction qui intervient dans les centrales nucléaires actuelles est la fission. Cette réaction génère deux atomes plus légers à partir d'un atome lourd. La réaction de fusion représente le processus inverse, c'est-à-dire la génération d'un atome à partir de deux atomes légers. Dans le cœur du soleil, la réaction de fusion se produit naturellement à une température de plusieurs millions de degré. L'histoire de la fusion contrôlée commence en 1920 grâce au physicien/chimiste Francis William Aston qui en mesurant le défaut de masse de l'Hélium, établit qu'il est possible de récupérer une importante quantité d'énergie en produisant un atome d'Hélium via la fusion de deux éléments plus légers [1]. La Figure 1.1 schématise la réaction de fusion entre un atome de Deutérium et un atome de Tritium qui est techniquement la plus intéressante et accessible, donc celle visée en premier lieu par les tokamaks.

Bien que cette réaction de fusion ait déjà été réalisée dans différentes installations expérimentales, l'entretien de cette réaction et son contrôle au sein d'une machine de type tokamak est toujours un sujet actif de recherche. Cette prouesse technique représente un enjeu environnemental et économique majeur. Contrairement à la fission, la réaction de fusion ne peut pas s'emballer car la quantité de réactifs dans le tokamak est très faible à tout instant et elle doit être ré-alimentée en permanence. Cette réaction est aussi plus propre car elle produit beaucoup moins de déchets radioactifs avec des demies-vies beaucoup plus courtes (de l'ordre de 10 ans).

1.1.1 Installations expérimentales

La réalisation de la réaction de fusion est un défi de taille car les conditions de pression et de température qui permettent de favoriser la réaction de fusion sont extrêmes. Principalement deux approches sont actuellement envisagées pour réaliser des réactions de fusion : la fusion par confinement inertiel et la fusion par confinement magnétique. Au CEA Cesta, le Laser Mégajoule

1. Une chronologie des avancés historiques de la fusion contrôlée est disponible via le lien https://en.wikipedia.org/wiki/Timeline_of_nuclear_fusion

réalise le processus de fusion par confinement inertiel en concentrant des faisceaux lasers de grande intensité sur une capsule de Deutérium/Tritium. On obtient une très forte densité de particules sur un temps relativement court. Dans le processus de fusion par confinement magnétique la réaction de fusion est obtenue au sein d'un *plasma* porté à très haute température avec une densité moins élevée que dans le processus inertiel, mais sur un temps plus long. Les contributions de cette thèse concernent la simulation de l'approche par confinement magnétique.

Le plasma est un état de la matière. Dans les conditions de température et de pression de notre quotidien, la matière se trouve sous trois états : solide, liquide et gazeux. Lorsqu'on chauffe très fortement un gaz, la matière se manifeste dans son quatrième état : le plasma. Ce terme a été introduit en 1928 par le physicien Irving Langmuir. Pour que la réaction de fusion se produise, la température requise au cœur du plasma doit dépasser le millions de degrés dans un environnement où la pression est très faible. Dans un plasma, les électrons ne sont plus en rotation autour d'un noyau d'atome. À l'état plasma, la matière est un mélange de particules chargées d'ions et d'électrons libres. Globalement, le plasma est électriquement neutre. Bien qu'il soit naturellement quasiment absent sur Terre, hormis au cours des aurores boréales aux pôles, le plasma constitue 99% de la masse de l'Univers.

L'étude de la matière à l'état plasma est un domaine à part entière de la physique appelé la *physique des plasmas*. Les recherches autour de ce domaine se décomposent en trois catégories : (*i*) la physique des plasmas industriels (tube à néon, écran plasma, satellite de communication, gravure par plasma en micro-électronique...), (*ii*) la physique des plasmas naturels (astrophysique) et (*iii*) la physique des plasmas thermonucléaires. C'est de ce dernier domaine dont il est question dans ce manuscrit.

Bien que globalement neutre, le plasma est localement chargé par les ions et les électrons qui le constituent. Il est donc possible de contrôler son comportement et de le confiner grâce aux forces électromagnétiques. Nous nous intéressons dans ce manuscrit aux installations de type *tokamak*, qui se constituent principalement d'une chambre toroïdale et de bobines autour de cette chambre. L'expérimentation de la fusion par confinement magnétique est aussi accessible grâce à un autre type d'installation analogue au tokamak appelé *stellarator*. Le stellarator confine le plasma uniquement grâce au champ magnétique généré par l'arrangement complexe de bobine le long du tore. Son étude ne fait pas l'objet de ce manuscrit.

Au CEA Cadarache, à l'Institut de Recherche sur la Fusion par confinement Magnétique (IRFM), les scientifiques réalisent des expériences grâce au tokamak Tore-Supra. Ces expériences permettent de mieux connaître les mécanismes de fusion et les effets du confinement magnétique sur le plasma. Elles consistent principalement à tester différentes configurations de la machine, à collecter un grand nombre de grandeurs physiques, puis à les dépouiller et les interpréter. Le projet international International Thermonuclear Experimental Reactor, Iter, lancé en 2006 a pour objectif de construire le plus grand tokamak au monde (voir Figure 1.2). Il est actuellement en cours de construction sur le site de Cadarache. L'objectif avec l'installation de recherche Iter est de démontrer, techniquement et scientifiquement, que la fusion peut devenir une source d'énergie industrielle à l'horizon 2050.

La Figure 1.3 schématise le confinement du plasma au sein d'un tokamak. De façon usuelle, on distingue deux directions pour se repérer dans un tokamak. On parle de la *direction toroïdale* lorsqu'on considère la direction le long du grand rayon du tore et de la *direction poloïdale* lorsqu'on se restreint à une section circulaire perpendiculaire à la direction toroïdale. Les anneaux qui se succèdent à équidistance dans la direction toroïdale sont des bobines dites *poloïdales* ; elles sont responsables de la génération d'un champ magnétique dont les lignes de champ sont parallèles à la direction toroïdale. La bobine au centre du tokamak, appelée *solénoïde central*, génère un champ magnétique dont les lignes de champ sont contenues dans les plans poloïdaux. La combinaison de

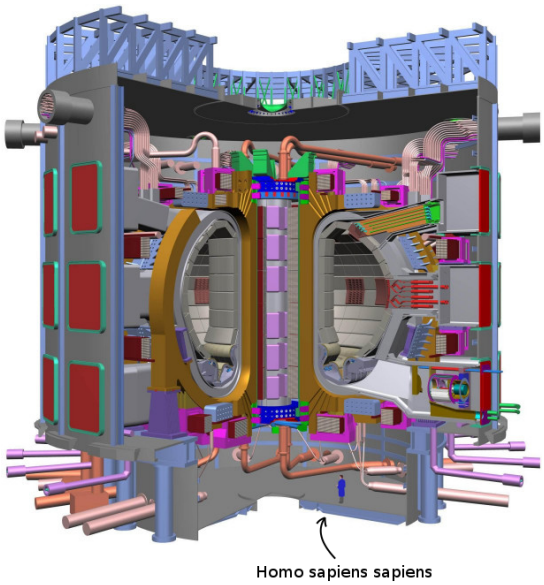


FIGURE 1.2 – Vue d'artiste du tokamak Iter en cours de construction ; chacune des 18 bobines pèse le poids d'un A380, soit 360 tonnes. Extrait de http://tempest.das.ucdavis.edu/pdg/ITER_Website/ITER.htm

ces deux champs magnétiques donne le champ magnétique résultant dont les lignes de champ sont de forme hélicoïdale. Ces lignes qui s'enroulent autour du tore confinent les particules du plasma et le maintiennent autant que possible éloigné des parois internes du tokamak (non représentées sur le schéma).

La maîtrise du confinement du plasma est une tâche d'une extrême complexité, car des interactions non linéaires entre le champ électromagnétique et la configuration du plasma sont à prendre en compte. De plus, ces phénomènes physiques existent à des échelles de temps et d'espace très diverses. En outre, le plasma est sujet à de nombreuses instabilités qui s'opposent à un bon confinement. Un des défis majeurs des installations de type tokamak est de trouver les configurations qui confinent le plus efficacement le plasma. Dans les cas où le plasma n'est plus efficacement contrôlé, il y a une diminution de l'efficacité du confinement et donc de la performance du tokamak. Dans des conditions particulières, la plasma risque d'entrer en contact avec la paroi du tokamak ce qui pourrait l'endommager. Les conditions de température et de flux de puissance étant extrêmes, des recherches sont toujours en cours afin de trouver des matériaux capables de mieux résister à ces événements.

1.1.2 La physique des plasmas

De façon complémentaire à l'expérimentation, la compréhension du comportement du plasma dans les tokamaks est aussi abordée par la modélisation théorique. Dans la section 1.2.1 de la thèse de C. Caldini-Queiros [CQ13] retrace l'historique des différents travaux scientifiques qui ont jalonné l'étude de la physique des plasmas.

Afin d'affiner notre compréhension de la dynamique temporelle du plasma au sein d'un tokamak, des modèles adaptés et des méthodes numériques spécifiques ont été développés. Dans cette section, une vue d'ensemble non exhaustive de ces modèles est donnée. Ils décrivent

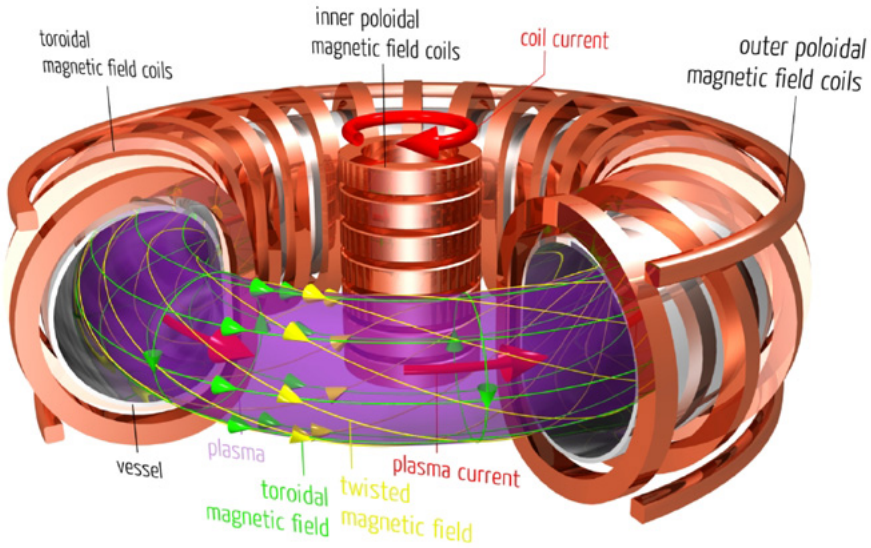


FIGURE 1.3 – Schéma d'un tokamak, du plasma confiné, des lignes de champ magnétique et des bobines qui génèrent ce champ. Extrait de https://en.wikipedia.org/wiki/Tokamak.

l'interaction entre les particules chargées du plasma et un champ électromagnétique. Ce champ est à la fois d'origine externe et généré par les particules elles-mêmes. Pour capturer parfaitement le comportement du plasma, un modèle complet consisterait à expliciter le mouvement de chaque particule du plasma grâce aux équations de Newton. À cause du très grand nombre de particules au sein d'un plasma, même de faible densité, cette approche est trop coûteuse à mettre en œuvre en temps de calcul.

Le développement de modèles accessibles à la simulation numérique est nécessaire. Une hiérarchie de modèles a ainsi été construite. Le choix d'un modèle représente un compromis entre précision et moyen de calcul. La description des modèles donnée ici s'inspire du cours de E. Sonnendrücker [Son10].

Modèle à N corps

Bien qu'il ne soit pas utilisé à cause de son coût de calcul trop élevé, le modèle à N corps constitue le modèle de référence à partir duquel se dérive les autres modèles. Au niveau microscopique, les particules évoluent suivant le principe fondamental de la dynamique. Dans le cas non relativiste, chaque particule obéit à la loi de Newton

$$\frac{dm\mathbf{v}}{dt} = \sum F_{ext}$$

où m est la masse de la particule, $\mathbf{v}$ sa vitesse et F_{ext} les forces qui s'appliquent à la particule. Dans notre cas, les forces F_{ext} se réduisent à la force de Lorentz induite par les champs électromagnétiques. Ils sont générés dans les particules chargées du plasma et par les bobines poloïdales. Les autres forces, comme le poids, sont généralement négligeables. On obtient donc

pour chaque particule la relation

$$\frac{dm\mathbf{v}_i}{dt} = \sum_j q(\mathbf{E}_j + v_i \times \mathbf{B}_j). \tag{1.1}$$

La vitesse d'une particule est liée à sa position par la relation

$$\frac{d\mathbf{x}_i}{dt} = \mathbf{v}_i. \tag{1.2}$$

Si on connait à un instant donné la position et la vitesse de chaque particule, ainsi que le champ magnétique, le système d'équation (1.1)–(1.2) détermine complètement l'évolution des particules.

Les plasmas thermonucléaires ayant approximativement une densité de 10^{20} particules par mètre cube, ce modèle microscopique est inutilisable. Il est alors nécessaire de trouver des modèles alternatifs, approchés mais assez précis pour capturer l'évolution du plasma et accessibles par nos moyens de calcul. Dans la hiérarchie des modèles, le modèle à N corps est le plus précis. Les approches alternatives qui en découlent se décomposent en deux grandes familles : (i) les modèles cinétiques (modèle intermédiaire) et (ii) les modèles fluides (modèle macroscopique). Une brève description des ingrédients qui les composent est donnée dans les deux paragraphes suivants.

Modèle cinétique

Dans le modèle cinétique, les particules de chaque espèce qui constitue le plasma (par exemple, ions et électrons) sont représentées par leur fonction de distribution $f_s(\mathbf{x}, \mathbf{v}, t)$ où l'indice s correspond à une espèce donnée. Cette fonction de distribution est à 6 dimensions : 3 pour la position en espace $\mathbf{x}$, 3 pour les coordonnées de vitesse $\mathbf{v}$, en plus de la dimension temporelle. Les coordonnées de position et de vitesse donnent une position dans l'*espace des phases*. À un temps t donné, la valeur $f_s(\mathbf{x}, \mathbf{v}, t)d\mathbf{x}d\mathbf{v}$ correspond au nombre moyen de particules dont la position et la vitesse se trouvent respectivement dans un élément de volume de taille d$\mathbf{x}$ dans l'espace ordinaire centré autour de la position $\mathbf{x}$ et dans un élément de taille d$\mathbf{v}$ dans l'espace des vitesses centré autour de la position $\mathbf{v}$.

Une description cinétique d'un plasma est nécessaire lorsque la fonction de distribution est éloignée de la distribution de Maxwell-Boltzmann (aussi appelée *Maxwellienne*) qui correspond à l'équilibre thermodynamique du plasma. Lorsque les effets collectifs sont dominants par rapport aux effets locaux telles que les collisions binaires entre particules, le modèle cinétique s'obtient en dérivant le modèle à N corps par des techniques de physique statistique. L'équation ainsi obtenue est appelée l'équation de *Vlasov* [Vla45] ; elle s'écrit dans le cas non relativiste :

$$\frac{\mathrm{d}f_s}{\mathrm{d}t} = \frac{\partial f_s}{\partial t} + \mathbf{v} \cdot \nabla_{\mathbf{x}} f_s + \frac{q_s}{m_s}(\mathbf{E} + \mathbf{v} \times \mathbf{B}) \cdot \nabla_{\mathbf{v}} f_s = 0.$$

On note ici $\nabla_{\mathbf{x}} f_s$ et $\nabla_{\mathbf{v}} f_s$ les gradients respectifs de f_s par rapport aux trois coordonnées en position et aux trois coordonnées en vitesse. Les constantes q_s et m_s correspondent à la charge et à la masse d'une particule de l'espèce s. L'équation de ce modèle exprime le fait que la fonction de distribution est conservée le long des trajectoires des particules qui sont déterminées par le champ électromagnétique.

Le champ électromagnétique résultant qui gouverne l'évolution du plasma dépend de la configuration du plasma lui-même. Pour décrire de façon réaliste l'évolution du plasma, il

convient de coupler l'équation de Vlasov aux équations de *Maxwell* [Max65, Dar05]. Ces équations permettent de calculer le champ électrique $\mathbf{E}$ et le champ magnétique $\mathbf{B}$ induits par la distribution des particules :

$$\begin{aligned} -\frac{1}{c^2}\frac{\partial \mathbf{E}}{\partial t} + \nabla \times \mathbf{B} &= \mu_0 \mathbf{J}, \\ \frac{\partial \mathbf{B}}{\partial t} + \nabla \times \mathbf{E} &= 0, \\ \nabla \cdot \mathbf{E} &= \frac{\rho}{\varepsilon_0}, \\ \nabla \cdot \mathbf{B} &= 0. \end{aligned}$$

Les termes sources des équations de Maxwell, la densité de charges $\rho(\mathbf{x}, t)$ et la densité de courant $\mathbf{J}(\mathbf{x}, t)$, s'expriment à partir des fonctions de distribution des différentes espèces de particules $f_s(\mathbf{x}, \mathbf{v}, t)$ à l'aide des relations :

$$\begin{aligned} \rho(\mathbf{x}, t) &= \sum_s q_s \int f_s(\mathbf{x}, \mathbf{v}, t) d\mathbf{v}, \\ \mathbf{J}(\mathbf{x}, t) &= \sum_s q_s \int f_s(\mathbf{x}, \mathbf{v}, t) \mathbf{v} d\mathbf{v}. \end{aligned}$$

On dit que le système d'équation *Vlasov-Maxwell* est auto-consistant. Ceci est dû au lien qui relie les deux équations. L'équation de Vlasov fait intervenir un champ électromagnétique. À un temps t donné, la résolution de l'équation de Vlasov fournit une nouvelle configuration de la fonction de distribution dans l'espace des phases. Cette nouvelle configuration induit une évolution du champ électromagnétique. La résolution des équations de Maxwell nous permet alors d'obtenir le champ électromagnétique associé. Ce nouveau champ correspondant au temps $t + \Delta t$ agit de nouveau sur la fonction de distribution à travers l'équation de Vlasov, et ainsi de suite. Ce couplage met en évidence le fait que les équations de Vlasov et de Maxwell sont intimement liées.

Dans le cas où d'autres interactions sont à prendre en compte dans l'évolution du plasma (par exemple les collisions entre particules, une source de chauffage ou encore un champ magnétique externe comme dans le code GYSELA), des termes additionnels sont introduits dans l'équation de Vlasov. Cette nouvelle équation est appelée équation de *Boltzmann*.

Modèle fluide

On peut dériver à partir du modèle cinétique introduit précédemment un modèle fluide qui permet de décrire l'évolution de grandeurs macroscopiques associées au plasma. Les principales grandeurs du modèle fluide sont :

— la densité définie par

$$n(\mathbf{x}, t) = \int f(\mathbf{x}, \mathbf{v}, t) d\mathbf{v},$$

— la vitesse moyenne $\mathbf{u}(\mathbf{x}, t)$ qui vérifie

$$n(\mathbf{x}, t)\mathbf{u}(\mathbf{x}, t) = \int f(\mathbf{x}, \mathbf{v}, t) \mathbf{v} d\mathbf{v},$$

— la température $T(\mathbf{x}, t)$ qui vérifie

$$n(\mathbf{x}, t)T(\mathbf{x}, t) = \frac{m}{3} \int |\mathbf{v} - \mathbf{u}(\mathbf{x}, t)|^2 f(\mathbf{x}, \mathbf{v}, t) d\mathbf{v}.$$

où m désigne la masse d'une particule.

Ces quantités sont définies à partir des trois premiers *moments* en vitesse de la fonction de distribution $f(\mathbf{x}, \mathbf{v}, t)$. Lorsque les particules sont proches de l'équilibre thermodynamique, le modèle fluide semble être une bonne approximation de l'évolution globale du plasma. Sous l'effet des collisions, la fonction de distribution tend vers une fonction de distribution Maxwellienne.

On peut noter que les grandeurs qui sont calculées ne dépendent pas de l'espace des vitesses $\mathbf{v}$. Ce modèle est donc moins précis que le modèle cinétique, mais bien moins coûteux en terme de temps de calcul car on passe d'un problème à 6 dimensions à un problème à 3 dimensions. Il ne sera plus fait mention du modèle fluide dans la suite de ce manuscrit. Notre étude se focalise sur les modèles de type cinétique.

1.2 Calcul haute performance

L'application qui fait l'objet de cette thèse est GYSELA. Elle utilise un modèle $5D$ dit *gyrocinétique* (voir section 1.3). Pour réaliser des simulations réalistes, les quantités de données à manipuler sont de très grande taille (dans de grandes configurations, la taille d'une seule fonction de distribution est d'environ 1 To) et les quantités de calculs à effectuer sont conséquentes. Ces simulations requièrent l'utilisation de puissantes machines de calcul : les *supercalculateurs*. Aujourd'hui, de nombreux domaines scientifiques qui utilisent la simulation ont recours au calcul haute performance.

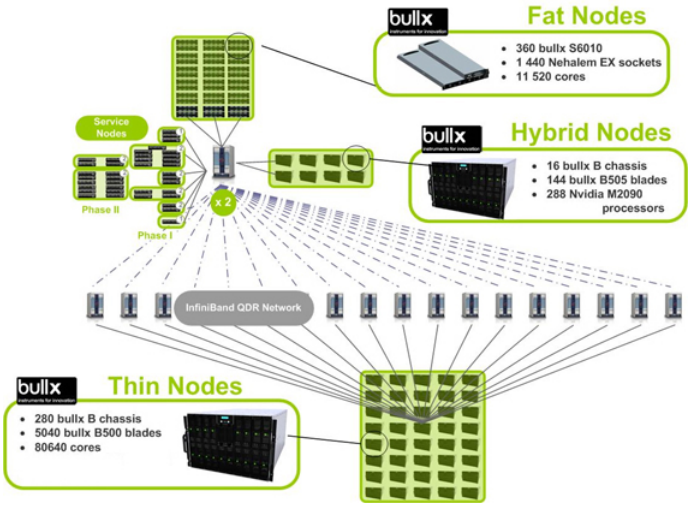


FIGURE 1.4 – Schéma de l'architecture de la machine Curie. Extrait de http://www-hpc.cea.fr/fr/complexe/tgcc-curie.htm

La Figure 1.4 illustre l'architecture hiérarchique du calculateur CURIE qui est une machine Tier-0 PRACE, c'est-à-dire dédiée à des projets Européens de grande envergure. Ce type de hiérarchie est commun dans le domaine du calcul haute performance. La machine est constituée plusieurs de nœuds de calcul qui sont reliés entre eux par un réseau rapide. Ces machines sont

souvent appelées *grappe de calcul* ou encore *cluster*. La machine HELIOS qui est dédiée aux simulations de la communauté de physique des plasmas présente une architecture très similaire à celle de CURIE. Au cours de cette thèse, des simulations et des tests ont été effectués sur ces deux machines.

De façon macroscopique, on peut distinguer deux niveaux de parallélisme sur ce type de machine. À haut niveau, la machine est à mémoire distribuée, une unité de calcul est représentée par un nœud de calcul et le réseau rapide est utilisé pour assurer les communications et par suite la cohérence globale des calculs. En descendant d'une échelle, un nœud de calcul peut être considéré comme une machine à part entière à mémoire partagée, où une unité de calcul est représentée par un cœur de calcul d'un processeur et des politiques de synchronisation entre unités de calcul doivent être mises en place pour assurer la cohérence des calculs[2]. Ces deux niveaux de parallélisme sont exploités respectivement grâce aux paradigmes de programmation MPI (*Message Passing Interface*) [GLDS96] et OPENMP (*Open Multi-Processing*) [DE98]. Les applications qui mélangent ces deux paradigmes, ou modèles, sont dites *hybrides*. C'est le cas de l'application GYSELA.

Durant ces dernières années, une activité croissante a été soutenue pour la création ou l'expérimentation de nouveaux modèles de programmation parallèle. Ces recherches sont motivées en partie par les changements et la diversité des architectures que proposent les supercalculateurs actuels (par exemple, l'utilisation de cartes accélératrices). De plus, il n'y a pas de certitude sur le type de matériel qui permettra d'atteindre l'Exascale qui représente le prochain cap symbolique en terme de puissance de calcul à franchir pour les constructeurs de machines et de puces. Dans ce contexte, il est important de pouvoir décorréler les aspects matériels du potentiel de parallélisation d'une application. La création d'un modèle de programmation permettant l'expressivité de tous les niveaux de parallélisme des futures machines est aujourd'hui un sujet de recherche très actif.

Actuellement, l'utilisation d'une technologie pour paralléliser une application impose un cadre de pensée qui lui est spécifique. Dans l'hypothèse où un changement de modèle doit s'effectuer, un effort de développement considérable doit être fourni pour adapter l'application. Les technologies qui offrent la possibilité d'effectuer une transition progressive d'un modèle de programmation à un autre devraient à l'avenir être privilégiées par les domaines applicatifs.

1.3 Description du code GYSELA

Dans un plasma faiblement collisionnel, les modèles fluides sont souvent inadaptés pour décrire l'évolution du plasma. Dans ce contexte, l'étude du comportement du plasma nécessite l'utilisation d'un modèle cinétique qui représente plus fidèlement la distribution en vitesse des particules. Le modèle cinétique admet comme inconnue une fonction $6D$ de distribution de densité des particules. La résolution numérique de l'équation de Vlasov associée et des différents opérateurs nécessaires à une description réaliste du plasma requiert une puissance de calcul toujours hors de portée aujourd'hui. Afin de réduire la quantité de calcul à réaliser, il est nécessaire de dériver un modèle intermédiaire entre le modèle cinétique à 6 dimensions et le modèle fluide à 3 dimensions.

L'application GYSELA [GSG+06, GBB+06, GSG+08, GAB+15], pour *GYrokinetic SEmi-LAgragian*, simule l'évolution du plasma grâce au modèle *gyrocinétique* qui réduit le modèle cinétique d'une dimension en vitesse.

2. Le *too much milk* problème ou encore le dîner des philosophes sont des exemples classiques qui mettent en évidence la nécessité de synchronisation.

La section 1.3.1 décrit l'observation qui est à l'origine du modèle gyrocinétique. Ce modèle réduit la dimensionnalité du problème et permet de réaliser des simulations dans un temps raisonnable. Pour une description détaillée du modèle gyrocinétique, le lecteur peut consulter le manuscrit de J. Abiteboul [Abi12] ainsi que ses références. La section 1.3.2 décrit le système de coordonnées utilisé par GYSELA ainsi que certains aspects de sa décomposition de domaine. La section 1.3.3 montre une étude de la consommation de mémoire de GYSELA. Enfin, la section 1.3.4 présente les outils de développement qui sont utilisés pour maintenir et faire évoluer une application de cette envergure.

1.3.1 Hypothèses du modèle gyrocinétique

Sous l'influence d'un fort champ magnétique non uniforme $\mathbf{B}(\mathbf{x})$, où $\mathbf{x}$ est une position de l'espace, les particules du plasma s'enroulent autour d'une trajectoire proche des lignes de champ [Lit88], comme on peut le voir sur la Figure 1.5. L'étude du plasma soumis à un fort champ magnétique est adapté aux installations de type tokamak car c'est typiquement le genre de champ magnétique qui est utilisé. On appelle *centre-guide*, $\mathbf{X}(t)$ sur la figure, la projection orthogonale de la particule sur cette trajectoire.

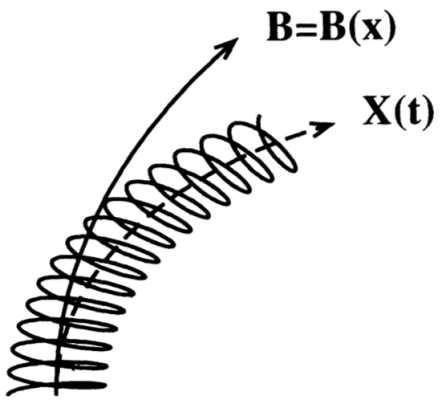


FIGURE 1.5 – Illustration du mouvement d'une particule chargée dans un fort champ magnétique non uniforme, $\mathbf{B}(\mathbf{x})$, où $\mathbf{x}$ désigne une position de l'espace. $\mathbf{X}(t)$ désigne la position du centre-guide associé au mouvement de la particule. Cette figure est extraite de l'article [Lit88].

La fréquence cyclotronique ω_c qui représente la fréquence de giration d'une particule autour de son centre-guide est grande dans l'hypothèse d'un fort champ magnétique. La distance qui sépare une particule de son centre guide est appelé rayon de Larmor, généralement noté ρ_c. On suppose aussi que le champ magnétique $\mathbf{B}$ varie lentement en espace et en temps par rapport à la fréquence cyclotronique ω_c. Sous ces hypothèses, l'étude de la dynamique des centres-guides constitue une bonne approximation de la dynamique de l'ensemble des particules du plasma. En appliquant une moyenne sur le mouvement rapide de la particule, on peut réécrire l'équation de Vlasov qui admet désormais comme inconnue, non plus une fonction de distribution des particules, mais une fonction de distribution des centres-guides. L'opérateur qui relie ces deux fonctions de distribution est appelé *gyromoyenne*. Il permet de réduire le modèle initial exprimé dans l'espace des phases à $6D$ à un modèle exprimé dans un espace à $5D$. Cet opérateur est un élément fondateur du modèle gyrocinétique. Cette réduction de la dimensionnalité du problème permet de réduire les coûts de calcul.

De même que dans le modèle cinétique, l'équation de Vlasov est couplée aux équation de Maxwell. Dans le modèle cinétique, les équations de Maxwell sont exprimées par rapport à la fonction de distribution des particules. Dans le modèle gyrocinétique, ces équations sont ré-exprimées par rapport à la fonction de distribution des centres-guides. Le couplage des équations Vlasov-Maxwell définies par rapport à la fonction de distribution des centres-guides constitue le modèle gyrocinétique. Pour plus de détails, le lecteur peut consulter les travaux [Bri89, Hah88, LL95] et leurs références. L'article [BH07] expose la théorie moderne du cadre gyrocinétique.

Des approximations supplémentaires sont faites dans le modèle résolu par GYSELA pour alléger la charge de calcul. L'une d'entre elles est l'approximation *électrostatique* qui réduit la résolution de l'ensemble des équations de Maxwell à une équation qui est la suivante :

$$\mathbf{E} = -\nabla\Phi,$$

où Φ correspond au potentiel électrique.

Plus de détails sur les approximations réalisées par GYSELA sont disponibles dans la section 2.2.4 de [Abi12].

1.3.2 GYSELA : une application massivement parallélisée

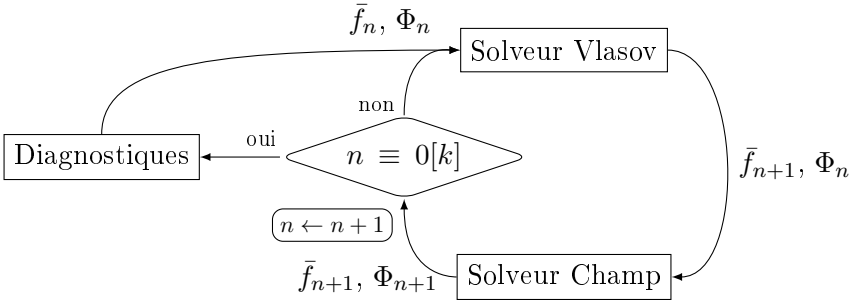


FIGURE 1.6 – Schéma d'un pas de temps de l'application GYSELA.

On peut décomposer grossièrement une exécution de GYSELA en trois étapes : (*i*) une phase d'initialisation, (*ii*) des itérations temporelles et (*iii*) une phase de sortie. La Figure 1.6 schématise une itération temporelle de GYSELA. L'évolution de la fonction de distribution des centres-guides, notée $\bar{f}_n$, se fait grâce à la résolution de l'équation de Vlasov et l'évolution du champ électrique grâce à la résolution d'une équation de type Poisson. Périodiquement, GYSELA exécute des diagnostiques afin d'extraire des grandeurs physiques de la fonction de distribution $\bar{f}_n$ et du champ de potentiel Φ_n. La sauvegarde des 5 dimensions de la fonction de distribution à chaque pas de temps étant trop coûteuse, les diagnostiques permettent d'extraire les grandeurs d'intérêt qui sont à 1, 2 ou 3 dimensions. Ces grandeurs sont sauvegardées sur disque dans des fichiers au format HDF5 [FCY99].

Nous utilisons le système de coordonnées (r, θ, φ) pour les 3 dimensions spatiales de $\bar{f}$; r et θ sont les coordonnées polaires dans une coupe poloïdale du tore et φ désigne l'angle toroïdal. La Figure 1.7 illustre ce système sur la géométrie d'un tokamak. Les deux dernières coordonnées modélisent l'espace des vitesses avec $v_\parallel$ la vitesse le long d'une ligne de champ magnétique, ainsi que μ le moment magnétique (équivalent à une vitesse moyenne dans un plan poloïdal). Dans le modèle gyrocinétique, μ est considéré comme un invariant adiabatique. Cette hypothèse est

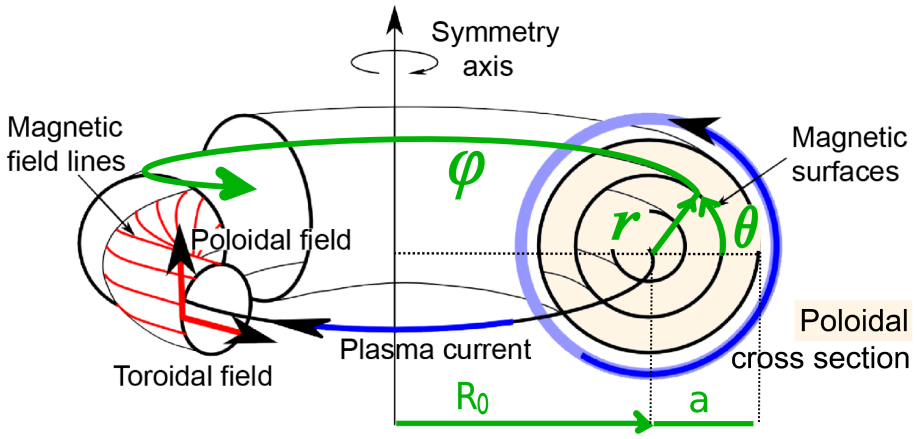


FIGURE 1.7 – Schéma du système de coordonnées utilisé dans GYSELA.

simplificatrice pour la résolution de l'équation de Vlasov, car elle permet de traiter de façon indépendante la fonction de distribution selon sa valeur en μ.

La Figure 1.8 illustre l'algorithme global de GYSELA. Sur ce schéma on peut identifier d'une part l'équation de Poisson qui permet de calculer le potentiel électrique et d'autre l'équation gyrocinétique de Vlasov qui décrit le comportement de la fonction de distribution. L'équation de Poisson permet de calculer le potentiel électrique à partir d'informations sur les particules (densité de charge), alors que l'équation gyrocinétique de Vlasov décrit le comportement de la fonction de distribution des gyro-centres. L'opérateur qui relie les grandeurs entre l'espace des particules et l'espace des gyro-centres est l'opérateur de gyromoyenne.

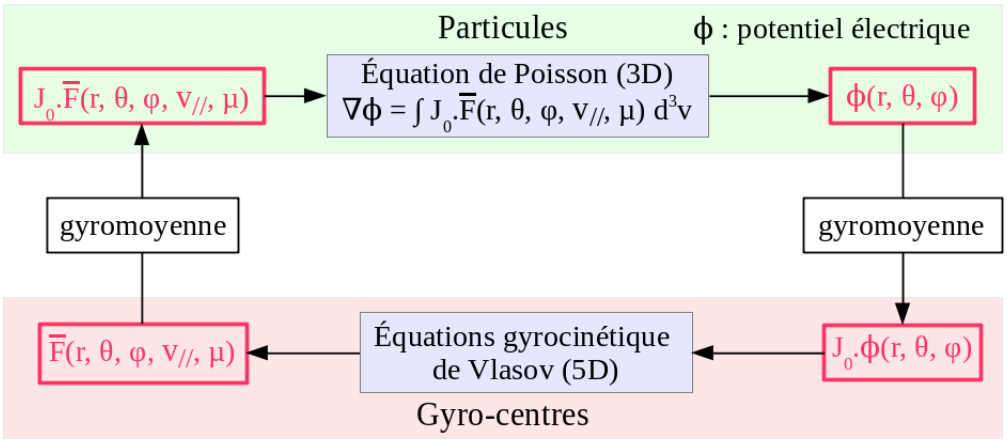


FIGURE 1.8 – Schéma algorithme global de GYSELA.

Le solveur de Vlasov

Le solveur de Vlasov représente la plus grande quantité de calcul lors d'une itération de GYSELA. Ce solveur prend en entrée la fonction de distribution au temps t et calcul la nouvelle configuration au temps $t + \Delta t$ de la fonction de distribution. Dans GYSELA, nous utilisons un

schéma de type Semi-Lagrangien explicite avec l'utilisation d'un *splitting* de Strang [Str68] pour résoudre l'équation de Vlasov. Le splitting permet de décomposer la résolution de l'équation de Vlasov en un enchaînement d'équations d'advection à 1 ou 2 dimensions dans le cas de GYSELA. Puisque la cinquième dimension en μ agit comme un invariant adiabatique, la résolution de l'équation Vlasov peut se voir sans splitting comme une advection en 4 dimensions. L'algorithme 1 schématise l'enchaînement des advections réalisées par GYSELA pour faire évoluer la fonction de distribution, ainsi que la distribution des données entre les processus MPI. Pour avoir des détails plus précis sur la méthode de résolution, le lecteur peut consulter la section "Time-splitting" de [GSG+06].

Entrées : $\bar{f}_n(r, \theta, \varphi, v_\parallel, \mu)$
Sorties : $\bar{f}_{n+1}(r, \theta, \varphi, v_\parallel, \mu)$

1 Advection $1D$ de $\frac{\Delta t}{2}$ en $v_\parallel$ $(\forall(\mu, r, \theta) = [local], \forall(\varphi, v_\parallel) = [*])$;
2 Advection $1D$ de $\frac{\Delta t}{2}$ en φ $(\forall(\mu, r, \theta) = [local], \forall(\varphi, v_\parallel) = [*])$;
3 Transposition de $\bar{f}$;
4 Advection $2D$ de Δt en (r, θ) $(\forall(\mu, \varphi, v_\parallel) = [local], \forall(r, \theta) = [*])$;
5 Transposition de $\bar{f}$;
6 Advection $1D$ de $\frac{\Delta t}{2}$ en φ $(\forall(\mu, r, \theta) = [local], \forall(\varphi, v_\parallel) = [*])$;
7 Advection $1D$ de $\frac{\Delta t}{2}$ en $v_\parallel$ $(\forall(\mu, r, \theta) = [local], \forall(\varphi, v_\parallel) = [*])$;

Algorithme 1 : Schéma du splitting directionnel utilisé par GYSELA pour résoudre l'équation de Vlasov.

Comme il l'a été dit dans la section 1.2, l'application GYSELA est une application hybride parallélisée en MPI/OPENMP. En pratique, la fonction de distribution évolue sur un maillage $4D$. Soient N_r, N_θ, N_φ et $N_{v_\parallel}$ respectivement le nombre de points dans les dimensions r, θ, φ et $v_\parallel$. Chaque valeur de μ utilisée par la simulation est associée à un ensemble de processus MPI (un communicateur MPI). Au sein de chacun de ces ensembles, une décomposition de domaine $2D$ du plan poloïdal nous permet d'attribuer à chaque processus MPI un sous-domaine pris dans les dimensions (r, θ). Un processus MPI est donc responsable du stockage du sous-domaine défini par $\bar{f}(r = [i_{start}, i_{end}], \theta = [j_{start}, j_{end}], \varphi = *, v_\parallel = *, \mu = \mu_{value})$[3] (voir les lignes 1, 2, 6 de 7 l'algorithme 1). La décomposition parallèle est caractérisée par les valeurs $i_{start}, i_{end}, j_{start}, j_{end}, \mu_{value}$ qui sont connues localement. Ces domaines $2D$ découlent d'une décomposition par bloc classique dans la direction r en p_r sous-domaines et dans la direction θ en p_θ sous-domaines. Le nombre de processus MPI utilisés durant une exécution est égal à $p_r \times p_\theta \times N_\mu$, avec N_μ le nombre de points dans la direction μ de la fonction de distribution. À l'intérieur de chaque processus, nous utilisons des threads OPENMP pour exploiter le parallélisme à grain-fin.

Pour donner une intuition plus précise de la parallélisation de GYSELA, l'algorithme 2 détaille l'étape d'advection dans la direction φ. Cette description montre l'utilisation des différents niveaux de parallélisme.

L'advection dans la direction $v_\parallel$ est très similaire à l'advection en φ. L'advection dans le plan poloïdal, en (r, θ), est différente car elle est en $2D$ et elle utilise une autre décomposition de domaine. Ce choix implique des communications collectives entre processus pour transposer la fonction de distribution. Sur de grandes configurations d'exécution, ces communications représentent potentiellement un goulot d'étranglement et sont donc pénalisantes pour la

3. La notation $*$ représente toutes les valeurs d'une dimension donnée.

Entrées : $\bar{f}^{\star}(r, \theta, \varphi, v_{\parallel}, \mu)$
Sorties : $\bar{f}^{\diamond}(r, \theta, \varphi, v_{\parallel}, \mu)$

pour μ **faire en parallèle (MPI)**
  pour r **faire en parallèle (MPI)**
    pour θ **faire en parallèle (MPI)**
      pour θ **faire en parallèle (OpenMP)**
        pour $v_{\parallel}$ **faire**
          $\Delta\varphi \leftarrow v_{\parallel}\Delta t$;
          Calcul de spline dans la direction φ de $\bar{f}^{\star}(r, \theta, \varphi = *, v_{\parallel}, \mu)$;
          pour φ **faire**
            $\bar{f}^{\diamond}(r, \theta, \varphi, v_{\parallel}, \mu) = interpolation\_spline(\bar{f}^{\star}(r, \theta, \varphi - \Delta\varphi, v_{\parallel}, \mu))$;
          fin
        fin
      fin
    fin
  fin
fin

Algorithme 2 : Advection dans la direction φ de la fonction de distribution $\bar{f}^{\star}$.

scalabilité de l'application. Elles sont utilisées actuellement principalement à cause de verrous techniques qui sont à l'étude actuellement.

Bien que le modèle gyrocinétique réduise d'une dimension la fonction de distribution à calculer, les ressources en termes de calcul et de mémoire utilisées par une simulation de Gysela sont très importantes. Les efforts d'optimisation réalisés sur Gysela en font une application qui exploite efficacement les calculateurs actuels [LGCDP11, BGL+13, LHB+15, RSL+15, TBG+13].

1.3.3 Consommation mémoire de Gysela

Lorsqu'on souhaite obtenir des résultats plus précis ou plus réaliste pour une simulation numérique on augmente souvent la résolution du maillage utilisé. Pour l'application Gysela, un maillage fin permet par exemple de capturer le comportement de structures plus petites dans la fonction de distribution. La nature de l'équation de Vlasov à résoudre étant à $5D$, la taille de la fonction de distribution à manipuler est conséquente.

Comme il a était dit dans la partie précédente, chaque processus MPI est associé avec une valeur de μ. La fonction de distribution ainsi que le potentiel électrique sont distribués avec une parallélisation de type MPI. Chaque processus prend en charge une partie de la fonction de distribution ainsi qu'une partie du potentiel électrique représentée respectivement par un tableau $4D$ et un tableau $3D$. Ces deux quantités représentent les variables d'intérêt pour Gysela. Intuitivement, on peut penser qu'elles occupent la plus grande partie de la mémoire allouée.

Le reste de la consommation mémoire vient de tableaux utilisés par exemple pour stocker des valeurs pré-calculées, de tampons mémoire MPI pour concaténer des données à l'envoi ou à la réception et de tampons privés aux threads OpenMP pour calculer des résultats temporaires. Dans la version initiale de Gysela, la plus grande partie des tableaux de l'application était allouée durant la phase d'initialisation.

Afin de mieux comprendre la consommation mémoire de Gysela, nous avons surchargé la fonction d'allocation Fortran (`allocate()`) afin de récupérer les méta données suivantes pour

Nombre de cœurs Nombre de processus MPI	$2K$ 128	$4K$ 256	$8K$ 512	$16K$ 1024	$32K$ 2048
structures 4D	209.2 67.1%	107.1 59.6%	56.5 49.5%	28.4 34.2%	14.4 21.3%
structures 3D	62.7 20.1%	36.0 20.0%	22.6 19.8%	19.7 23.7%	18.3 27.1%
structures 2D	33.1 10.6%	33.1 18.4%	33.1 28.9%	33.1 39.9%	33.1 49.0%
structures 1D	6.6 2.1%	3.4 1.9%	2.0 1.7%	1.7 2.0%	1.6 2.3%
Total par processus MPI en Go	311.5	179.6	114.2	83.0	67.5

TABLE 1.1 – Strong scaling : taille des allocations en Go (par processus MPI) et pourcentage par rapport au pic mémoire de chaque type de structure de données.

chaque tableau : le nom du tableau, sa dimension et sa taille. En utilisant ces données nous avons réalisé une étude en *strong scaling* de l'application.

Strong scaling mémoire

D'un point de vue de la mémoire, le strong scaling consiste à relever la consommation par processus d'un programme parallèle pour une taille fixe du problème d'entrée et en faisant varier le nombre de processus MPI. En utilisant un grand nombre de processus, la consommation mémoire des structures qui bénéficient d'une décomposition de domaine représente généralement une faible portion de la consommation mémoire globale. Si pour une simulation donnée avec n processus on utilise x Go de mémoire par processus, on peut espérer dans le cas idéal qu'en faisant la même simulation avec $2n$ processus, on obtienne une consommation mémoire de $\frac{x}{2}$ Go par processus. Dans ce cas, on dit que la *scalabilité* mémoire est parfaite. Mais en pratique, ce n'est généralement pas le cas à cause des surcoûts mémoire dus à la parallélisation.

Les résultats de cette étude sont donnés au Tableau 1.1. Ces chiffres ont été obtenus en utilisant 16 threads par processus MPI. Afin de maximiser la mémoire disponible pour un processus, nous configurons généralement une simulation de GYSELA de sorte à ce qu'un processus corresponde à un nœud de calcul. Dans notre cas, la consommation mémoire par processus MPI dépend essentiellement du nombre de processus et de la taille du maillage considéré.

Le Tableau 1.1 montre l'évolution de la consommation mémoire en fonction du nombre de cœurs et de processus MPI utilisés. Le pourcentage de la consommation mémoire par rapport au total de la mémoire consommée est donné pour chaque type de structure de données. Le maillage que nous avons utilisé est le suivant : $N_r = 1024$, $N_\theta = 4096$, $N_\varphi = 1024$, $N_{v_\parallel} = 128$, $N_\mu = 2$. Ce maillage est plus conséquent que ceux utilisés en production actuellement, mais il correspond aux besoins futurs, spécialement à ceux de la physique en configuration multi-espèces ou avec électrons cinétiques. Le dernier cas avec 2048 processus requiert 67.5 Go de mémoire par processus MPI.

Dans GYSELA, les structures $2D$ et $1D$ ne bénéficient pas de la décomposition de domaine. En regardant les relevés de consommation mémoire par type de structure, on peut voir que la consommation des structures $2D$ ne dépend pas du nombre de processus utilisés. La taille de

ces structures dépendent surtout du maillage et du nombre de threads par processus utilisés. On peut voir que sur le plus grand cas avec 2048 processus MPI, le coût mémoire des structures $2D$ représente 49 % de l'empreinte mémoire totale. Cela constitue une réelle difficulté pour le passage à plus grande échelle de l'application, c'est-à-dire pour de plus grands maillages.

Les nœuds de calcul des supercalculateurs que nous utilisons aujourd'hui disposent généralement d'une quantité de mémoire inférieur à 67.5 Go. On peut citer par exemple la machine HELIOS qui dispose de 64 Go par nœud ou encore les machines de types BLUE GENE/Q qui ne disposent que de 16 Go par nœud. Face à ce constat, "l'empreinte mémoire" de l'application GYSELA apparait clairement comme une difficulté pour son passage à plus grande échelle.

Dans ce manuscrit, l'usage de l'expression empreinte mémoire (*memory footprint*) ne correspond pas à sa définition stricte. Au sens strict, l'empreinte mémoire comptabilise l'ensemble des segments mémoire qui sont utilisés ou référencés par un programme durant son exécution ; elle ne se limite pas à la mémoire allouée sur le tas (cela correspond aux allocations avec la fonction `allocate()`). À titre d'exemple, parmi les zones mémoires qui ne sont pas considérées par notre étude, on peut citer la zone mémoire occupée par le programme à exécuter, la pile qui contient les arguments et les variables locales d'une fonction, le segment de données qui contient les variables globales et statiques et la consommation mémoire de bibliothèques tierces utilisées par l'application. Dans notre étude de la consommation mémoire de GYSELA, nous supposons que l'impact de ces zones mémoires est négligeable par rapport à la quantité de mémoire allouée sur le tas. Dans la suite de ce manuscrit, par abus de langage, nous utiliserons le terme empreinte mémoire pour désigner la mémoire allouée sur le tas par GYSELA.

1.3.4 Cycle et environnement de développement

L'application GYSELA est en développement depuis plus de 10 ans maintenant à l'IRFM. Plusieurs personnes contribuent simultanément à l'application en développant différentes fonctionnalités. On peut distinguer trois activités autour de l'application GYSELA :

physique : modification de certains termes des équations à résoudre, simulation de plusieurs espèces, ajout de composants physiques, etc ;

mathématiques appliquées : modification de la méthode numérique utilisée pour résoudre une équation, utilisation d'une méthode d'interpolation plus précise, etc ;

calcul haute performance : modification d'un schéma de communication, optimisation d'une section de l'application, etc.

Dans ce contexte interdisciplinaire, les outils qui peuvent faciliter la communication entre les différentes communautés sont très utiles. Les graphiques générés par les outils développés durant cette thèse font partie de ces outils.

Le développement de nouvelles fonctionnalités au sein d'une application de cette envergure est complexe et nécessite l'utilisation d'outils de développement.

Gestionnaire de version

Afin de permettre le développement concurrent de fonctionnalités, nous utilisons le gestionnaire de version `git`. Le mécanisme de branche qu'il propose permet premièrement d'isoler le développement des fonctionnalités et deuxièmement de simplifier leur intégration dans la branche de développement principale dite *master*. Le cycle de développement que nous avons mis en place dans GYSELA est illustré par la Figure 1.9 [4].

4. Cette organisation est inspirée de la description : http://nvie.com/posts/a-successful-git-branching-model/.

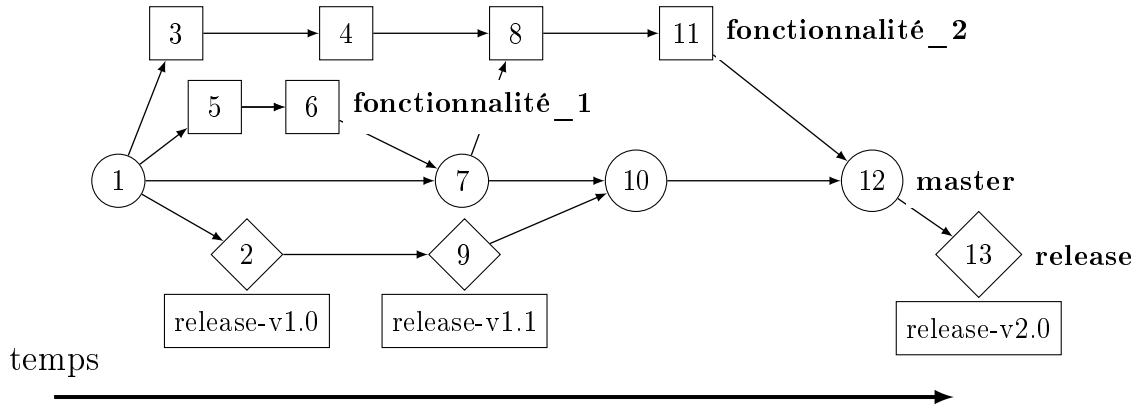


FIGURE 1.9 – Illustration du cycle de développement de GYSELA. Chaque cercle, carré et losange représente un commit `git`, ce qui conduit à une modification des fichiers qui constituent le projet. Les chiffres sont un exemple d'indexation chronologique des modifications. Un texte en gras correspond au nom d'une branche. Les rectangles en bas de la figure correspondent aux versions stables, diffusables de l'application, appelées aussi *release*.

Intégration continue

Pour limiter efficacement l'introduction de *bug* dans l'application, durant ma thèse, nous avons mis en place un système d'intégration continue. Il permet d'automatiser l'exécution de tests prédéfinis par l'utilisateur. Leur exécution est déclenchée soit de façon événementielle (par exemple lorsque la branche master est modifiée), soit de façon périodique (par exemple toutes les nuits). L'utilisation de cet outil est largement répandue dans le monde industriel, mais ce n'est pas toujours le cas des applications de calcul intensif.

Nous utilisons actuellement le logiciel JENKINS pour automatiser nos tests. Certains tests consistent simplement à vérifier que notre application peut être compilée sur les différentes machines utilisées. Ce type de vérification simple nous permet de garantir par exemple que la branche principale est toujours fonctionnelle. Le développement d'une nouvelle fonctionnalité peut donc commencer à tout moment à partir de la branche principale. Dans le cas où un test de compilation échoue, nous analysons au plut tôt la cause du problème. Cela peut être dû, par exemple, aux dernières modifications apportées au code, ou encore à la mise à jour d'une bibliothèque d'une machine qu'on utilise et qui n'est pas compatible avec l'application en l'état. Ce type de test a un intérêt double car il vérifie systématiquement les développements et leur comptabilité avec plusieurs environnements spécifiques aux différentes machines de production et de développement.

L'application GYSELA utilise différentes machines qui proposent chacune leur environnement d'exécution : différentes versions de bibliothèques, différents compilateurs, différents *job scheduler*[5], etc... Pour un développeur, vouloir tester la compilation de GYSELA sur chaque machine est une tâche très répétitive et fastidieuse. L'automatisation permet de garantir la validité du code par rapport aux machines utilisées et permet aussi de gagner du temps.

Des tests plus évolués vérifient la validité des résultats de simulation. Pour chacun de ces tests, les fichiers de résultats produits sur la version en développement de l'application pour une simulation (généralement courte) sont comparés à des fichiers de résultats de référence pré-existants. Ce type de test est particulièrement utile dans le cas où le développement d'une nouvelle fonctionnalité intègre la branche principale.

5. Gère les queues d'attente sur les calculateurs : https://en.wikipedia.org/wiki/Job_scheduler

Ces différents outils font partie des aspects techniques du développement de grandes applications. Ces outils améliorent grandement les conditions de travail et ils tendent à se rendre indispensables.

1.4 Positionnement et contribution de la thèse

Les travaux réalisés dans cette thèse ont contribué à optimiser des aspects relatifs au calcul haute performance de l'application GYSELA. On peut distinguer trois catégories de contributions lorsqu'il s'agit de l'optimisation de simulations numériques : (*i*) simulation d'un plus grand domaine de calcul, (*ii*) réduction du temps de calcul et (*iii*) amélioration de la précision des calculs.

Les contributions de cette thèse concernent ces trois points et le manuscrit se décompose en deux parties.

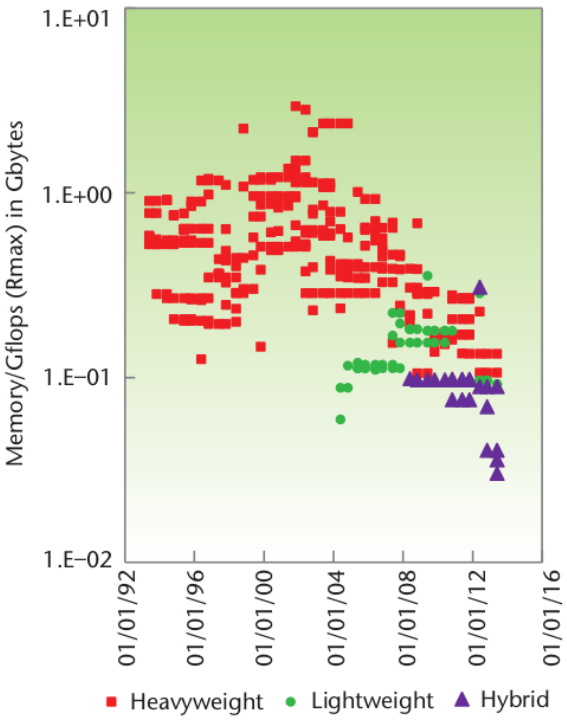


FIGURE 1.10 – Illustration de la tendance décroissante du ratio entre la quantité de mémoire disponible et le nombre de cœurs d'un supercalculateur. Ces chiffres ont été relevés sur les dix premières machines du classement TOP500. La configuration *heavyweight* ■ désigne les machines qui utilisent des processeurs hautes fréquences, la configuration *lightweight* ● désigne celles qui utilisent des processeurs faibles fréquences (par exemple BLUE GENE) et la configuration *hybrid* ▲ celles qui mixent l'utilisation de processeurs généralistes et de cartes accélératrices. Figure extraite de [KS13].

Étude de la consommation mémoire

Durant ces dernières années, les supercalculateurs ont eu tendance à offrir de moins en moins de mémoire par cœur de calcul comme on peut le voir sur la Figure 1.10. Cette diminution a été identifiée comme l'une des difficultés du challenge Exascale [HLP+11], même si de nouvelles technologies de mémoire non volatile (NVRAM) peuvent remettre en cause cette tendance[6]. D'autre part, l'étude de la scalabilité mémoire effectuée en section 1.3.3 (p. 14) témoigne de la consommation mémoire importante de l'application GYSELA. Nous savons enfin que les futures évolutions du modèle physique simulé par GYSELA nécessiteront l'usage d'une plus grande quantité de mémoire.

Pour ces différentes raisons, la première partie de cette thèse est dédiée à l'étude de la consommation mémoire de GYSELA. Pour se faire, nous avons développé un outil générique qui permet d'analyser l'empreinte mémoire d'un programme parallèle : la LIBMTM (*Modeling and Tracing Memory*). Cette bibliothèque propose une interface de programmation à l'utilisateur qui lui permet de remplacer les fonctions d'allocation/libération mémoire par des fonctions de la LIBMTM. Afin de faciliter son utilisation dans diverses applications, les dépendances externes ont été réduites au strict minimum et une interface en FORTRAN et en langage C est disponible.

Les traces `MTM` générées durant l'exécution sont post-traitées par un script fournit par la bibliothèque. Il permet de représenter graphiquement la consommation mémoire d'une simulation et, de fait, de localiser rapidement au niveau du code source où se produit le pic mémoire.

Dans le cadre des applications parallèles, la consommation mémoire d'un processus MPI dépend de la configuration de déploiement d'une simulation. Afin de pouvoir étudier la consommation mémoire d'une application dans différentes configurations, la LIBMTM permet de réaliser des prédictions hors-ligne précises de consommation mémoire. À l'heure actuelle, nous ne connaissons pas d'autres outils qui proposent une fonctionnalité équivalente.

Grâce à l'utilisation des outils fournis par la LIBMTM, nous avons pu réduire de 50% la consommation mémoire de GYSELA sur une grande simulation utilisant plus de 32K cœurs. Les réductions d'empreinte mémoire réalisées ont amélioré efficacement la scalabilité mémoire de l'application GYSELA et contribuent ainsi à l'accès à de plus grands maillages. L'utilisation de maillages plus fins permet d'avoir des simulations plus réalistes et permettra à terme d'enrichir le modèle physique simulé par GYSELA (par exemple l'ajout des électrons cinétiques). L'ensemble des outils de la LIBMTM représente un réel support pour le passage à plus grande échelle d'une application parallèle. Les travaux autour de la LIBMTM ont donné lieu aux publications [BGL+13, Roz14, RLR14, Roz15, RLRG15].

L'opérateur de gyromoyenne

Durant une itération temporelle de GYSELA, l'opérateur de gyromoyenne est sollicité à de nombreuses reprises sur le calcul du potentiel électrique $3D$ et sur la fonction de distribution $5D$ lors du calcul des diagnostiques. C'est un opérateur fondamental du modèle gyrocinétique. Sa précision et son temps de calcul ont donc un impact direct sur l'ensemble de la simulation.

L'opérateur de gyromoyenne utilisé initialement dans GYSELA est basé sur une approximation de Padé qui est valide pour de petits rayons de Larmor (voir section 1.3.1, p. 10). L'opérateur alternatif que nous proposons est basé sur une intégration directe dans l'espace réel qui fait intervenir des interpolations. Il est plus précis que l'opérateur initial, basé sur une approximation

6. Pour plus de renseignement, voir la section 4.2 de : http://science.energy.gov/~/media/ascr/pdf/program-documents/docs/Exascale-ASCR-Analysis.pdf ; le site : http://www.hpctoday.com/state-of-the-art/non-volatile-memory-nvm-technology-for-exascale-computing-a-position-paper/ ; ou encore le site : http://www.hpl.hp.com/research/systems-research/themachine/

de Padé, et valide sur un plus grand intervalle de rayons de Larmor. De plus, ce nouvel opérateur se prête mieux à la parallélisation.

La seconde partie de ce manuscrit décrit de façon détaillée l'opérateur de gyromoyenne, dans sa version existante basé sur approximation de Padé, et dans sa version basée sur une intégration directe. Cette dernière étant plus coûteuse que la version initiale en terme de temps de calcul, des optimisations séquentielles ont été réalisées. Une description formelle sous forme de produit matrice-vecteur a guidé ces optimisations qui ont conduit à différentes approches pour favoriser les effets de cache. La meilleure optimisation permet de réduire de 40% le temps d'exécution du calcul de gyromoyenne basé sur intégration directe, et se rapproche donc du temps de calcul de la gyromoyenne basée sur l'approximation de Padé.

Des comparaisons entre les différentes versions de gyromoyenne tant sur la précision que sur le coût d'exécution ont été réalisées. Après intégration dans GYSELA, nous avons validé les résultats du nouvel opérateur sur un cas test de référence de la physique des plasmas. Ce travail fait suite aux articles [CMS10, SMC+15] et a donné lieu à la publication [RSL+15].

Dans le dernier chapitre de ce manuscrit, nous étudions une extension de l'opérateur de gyromoyenne en MPI. Cette version parallèle de la gyromoyenne représente un enjeu important pour l'application GYSELA, car elle permettra de changer le schéma de communication de l'application et en particulier d'éviter certaines communications collectives qui représentent un goulot d'étranglement sur de grandes configurations. Des études de scalabilité en temps d'exécution de l'opérateur permettent de voir son comportement en fonction du nombre de processus. Les contraintes de mise en œuvre qui ont été rencontrées durant l'étude sont aussi discutées, l'objectif étant d'intégrer à moyen/long terme une version parallèle de la gyromoyenne à GYSELA.

Les travaux de cette thèse ont été réalisés dans le cadre d'une collaboration entre Inria (au sein de l'équipe HiePACS), le CEA/IRFM et le CEA/MdS.

Première partie

Etude de la consommation mémoire

Chapitre 2

Modélisation de l'évolution de la consommation mémoire grâce à la libMTM

Sommaire

Durant son exécution, un programme met à contribution différentes ressources d'une machine, à savoir la puissance de calcul du processeur, la mémoire physique (dite aussi mémoire vive), la mémoire de stockage (disque dur), le réseau de communication entre différents nœuds de calcul, etc. Chacune de ces ressources constitue potentiellement une difficulté pour le passage à l'échelle d'une application. Dans notre étude, nous nous sommes concentrés sur l'utilisation de la mémoire physique.

Notre étude est motivée par le contexte et les besoins de Gysela. Comme le montre la Figure 1.10 (p. 18), la tendance actuelle des supercalculateurs est d'offrir de moins en moins de mémoire par cœur de calcul. D'autre part, les quantités de données qui sont en jeu dans les applications scientifiques sont généralement très importantes. La section 1.3.3 qui présente la consommation mémoire de Gysela en témoigne. De plus, nous savons que les fonctionnalités futurs de Gysela nécessiteront l'usage d'une plus grande quantité mémoire. Par exemple, un des objectifs à moyen terme est de pouvoir simuler le comportement des électrons cinétiques. Pour être capable de capturer le comportement de ce type de particule, la résolution du maillage poloïdal doit être augmentée de plusieurs ordres de grandeur par rapport à sa taille actuelle

(échelle spatiale des structures physiques plus petite). Cela se traduit directement par une augmentation de la consommation mémoire.

Pour ces raisons, qu'on retrouve par ailleurs dans d'autres applications parallèles, la consommation mémoire est un point critique car c'est aujourd'hui un des principaux points bloquants pour l'utilisation d'un plus grand nombre d'unités de calcul pour l'application GYSELA. Afin de mieux maîtriser la consommation mémoire, la bibliothèque LIBMTM (*Modeling and Tracing Memory*) a été développée dans le cadre de cette thèse. Cette bibliothèque satisfait deux objectifs : (*i*) visualiser graphiquement la consommation mémoire d'une simulation et (*ii*) prédire la consommation mémoire d'un processus MPI pour un jeu de paramètres d'entrée donné.

Ce chapitre présente le fonctionnement et l'implémentation de la LIBMTM. Avant d'entrer dans les détails techniques, la section 2.1 donne une vue d'ensemble des outils d'analyse de performance et de profiling mémoire existants et positionne la LIBMTM. Dans la section 2.2, le fonctionnement et l'interface de programmation de la LIBMTM sont détaillés. La section 2.3 présente les analyses qui sont réalisées par le script de post-traitement de la LIBMTM. Une discussion sur les différents avantages que donnent l'utilisation de la LIBMTM conclut ce chapitre. Les travaux autour de la LIBMTM ont donné lieu aux publications [BGL+13, Roz14, RLR14, Roz15, RLRG15].

2.1 Travaux relatifs

2.1.1 Outils de profiling

Les outils d'analyse de performance peuvent être classés selon l'utilisation qu'ils proposent. Le premier type d'outil regroupe ceux qui peuvent fournir des métriques sur un programme qui est en train de s'exécuter, et le second type désigne ceux qui procèdent à l'analyse du programme une fois qu'il a terminé son exécution, c'est-à-dire de façon post-mortem. Ces deux approches ne sont pas motivées par les mêmes objectifs.

Les outils qui analysent un programme durant son exécution sont dits des outils de *monitoring*. Ce genre d'outil est très utile pour des applications qui ne doivent pas être arrêtées mais dont on souhaite étudier le comportement. Typiquement un serveur web est représentatif de ce genre d'application. Ces outils sont généralement orientés analyse de performance et utilisent généralement une méthode d'échantillonnage d'instructions, *sampling*, pour récupérer des informations.

Les applications de calcul intensif utilisent plutôt des outils d'analyse *post-mortem* qui s'appuient sur des traces générées à l'exécution du programme. Différents moyens techniques qui permettent de récupérer des informations sur l'exécution d'un programme sont explicités ci-dessous.

Outils de profiling de performance

Les outils d'analyse post-mortem s'appuient uniquement sur des traces générées à l'exécution. Le genre d'informations enregistrées peut être par exemple le temps d'exécution, le nombre de messages MPI envoyés, le temps inoccupé (idle), la consommation mémoire, et plus encore. Pour obtenir ces informations, l'application doit être instrumentée. L'instrumentation peut être faite à 4 niveaux : dans le code source, à la compilation, à l'édition de liens ou durant l'exécution (*just in time*).

Différents outils génériques sont disponibles pour analyser les aspects performance d'un programme. L'outil SCALASCA [GWW+10] est capable d'instrumenter le code source à la

compilation. Cette approche a l'avantage de couvrir toutes les parties du code et elle permet la personnalisation des informations à récupérer. Cette approche systématique donne une trace complète, mais la récupération d'informations dans toutes les fonctions du code peut induire un surcoût en exécution important [7].

L'ensemble d'outils EZTrace [AMGRT13] offre la possibilité d'étudier les communication MPI et l'activité des threads OpenMP permet aussi de suivre la consommation mémoire en interceptant les appels à l'ensemble des fonctions C qui gèrent les allocations mémoire (par exemple `malloc()`, `free()`). Cette approche ne demande pas beaucoup d'effort de la part de l'utilisateur car il lui suffit de lier son application à une bibliothèque fournie par l'outil EZTrace pour activer la génération de traces.
Les outils Pin [LCM+05], DynamoRIO [BGA03] ou Maqao [DBC+05] instrumentent l'exécutable d'une application durant son exécution. L'avantage est ici l'aspect générique de la méthode. Tous les programmes peuvent être instrumentés de cette façon, mais contrairement à notre approche, l'instrumentation dynamique introduit souvent un surcoût significatif en temps d'exécution.

Un autre logiciel couramment utilisé pour détecter les fuites mémoire est Valgrind [NS07], bien qu'il ne se limite pas à cet objectif. Ce logiciel utilise une instrumentation lourde au niveau binaire. Cette instrumentation lui permet de récupérer l'ensemble des accès mémoire effectués lors de l'exécution d'un programme. Parmi les outils développés à partir de Valgrind, l'outil Massif [Car13] permet de représenter graphiquement la consommation mémoire d'un programme.

Outils de profiling mémoire

Concernant plus particulièrement l'étude de la consommation mémoire sur le tas, des travaux similaires existent. Le programme mprof [ZH88], qui s'inspire de gprof, est potentiellement le premier projet à s'intéresser au suivi de la consommation mémoire sur le tas. Cette thématique est néanmoins toujours d'actualité comme on peut le voir avec [Mil14, gpe09, gme02, log14b, log14a]. Le projet purify [HJ91] est plus orienté vers la détection des fuites mémoire et des erreurs d'accès, mais il permet tout de même d'avoir accès au suivi de la consommation. Ces outils fournissent une bibliothèque qui permet d'intercepter les appels aux fonctions d'allocation. Ils se basent donc sur une instrumentation au niveau de l'édition de liens. Le but des outils de suivi mémoire étant de fournir des informations au développeur afin qu'il puisse minimiser la consommation mémoire de son application ou détecter des fuites mémoires, des études sur des méthodes de minimisation de l'utilisation du tas ont été réalisées dans [RR97] et ses références. Il existe même un brevet [GR12] qui protège des travaux sur un processus de réduction de l'empreinte mémoire.

Parmi toutes les approches ci-dessus, aucune d'entre elles ne propose d'étudier la scalabilité mémoire d'une application en fonction de ses paramètres d'entrée. L'approche de l'instrumentation au niveau binaire ou en utilisant une bibliothèque de tierce partie demande peu d'effort de la part de l'utilisateur, mais ce type d'approche s'appuie sur un programme déjà compilé. Après compilation, il est difficile de faire le rapprochement entre les valeurs que contient la pile d'exécution lors d'une allocation et les variables du code source responsables de cette allocation. C'est un aspect qui est essentiel dans le fonction de l'outil de prédiction de la libMTM détaillé dans la suite de ce chapitre.

Le but de l'outil de prédiction développé pour Gysela est de prédire la consommation mémoire du programme si la valeur de certains paramètres change (par exemple les dimensions

7. Il est toutefois possible de filtrer les fonctions/fichiers à prendre en compte lors de l'instrumentation.

du maillage). La prédiction de la consommation mémoire est inaccessible si on ne peut pas faire le lien entre la taille des structures allouées et le nom des variables qui dimensionnent la taille des allocations. Afin de récupérer les informations nécessaires à la prédiction, notre approche se base sur une instrumentation au niveau du code source. La mise en place précise de cette instrumentation est à la charge du développeur.

La prédiction de la consommation mémoire en fonction de paramètres d'entrée en mode hors-ligne permet d'investiguer la scalabilité mémoire d'une application, qui est GYSELA dans notre étude. À ce jour, nous ne connaissons pas d'outil équivalent pour modéliser le comportement mémoire d'un programme.

2.1.2 Exemple de modélisation analytique de consommation mémoire

L'étude de la scalabilité mémoire peut aussi se faire par une approche plus analytique. C'est la démarche qui est présentée dans [KPE+12]. Ce travail a été réalisé sur l'application NEST qui simule la propagation d'une impulsion électrique dans un réseau neuronal. L'objectif à long terme de ce projet est de pouvoir simuler des réseaux neuronaux de la taille du cerveau humain. La consommation mémoire représente aussi pour cette application un verrou technique qui doit être étudié.

Dans cette démarche, une formule analytique modélisant la consommation mémoire d'un processus MPI est introduite. Cette expression dépend principalement du nombre de processus MPI, du nombre de neurones présents dans le réseau et du nombre de connexions entrantes d'un neurone. Certaines constantes sont aussi présentes pour modéliser des consommations mémoire difficiles à estimer en mode hors ligne, comme par exemple les consommations mémoire qui dépendent fortement des événements qui se produisent durant l'exécution. Dans le cas de l'application NEST, certaines consommations mémoire sont directement liées au chemin parcouru par l'impulsion électrique pour un réseau neuronal donné. Ces allocations sont modélisées par des constantes qui sont déterminées grâce à un protocole expérimental.

L'avantage de cette approche repose sur le fait qu'avec une expression assez simple, il est possible de prédire correctement la consommation mémoire maximale, *pic mémoire*, de l'application NEST. Néanmoins, cette méthode suppose que le développeur a une bonne connaissance de l'application parallèle pour déterminer correctement l'expression qui modélise le pic mémoire de l'application. Dépendant de la complexité de l'application, cette tâche peut s'avérer ardue.

Dans le cas de l'application GYSELA, le pic mémoire dépend entièrement du jeu de paramètres d'entrée d'une simulation. Plus de détails sur les hypothèses qui permettent de prévoir précisément la consommation mémoire d'une application en mode hors ligne sont donnés dans la section 2.3.4.

Par rapport au travail qui a été effectué sur l'application NEST, notre approche peut être appliquée avec une connaissance moins pointue du comportement de l'application étudiée. De plus, la prédiction qui est réalisée par notre outil reflète très exactement le comportement mémoire de l'application. Ce dernier point peut être crucial dans certains cas (voir la section 3.4.2).

2.2 Description de la LIBMTM

Afin de cerner au mieux le comportement mémoire de GYSELA, les objectifs suivants doivent être remplis par l'outil d'analyse :

1. la visualisation de la consommation mémoire ;

2. la prédiction de la consommation mémoire en mode hors ligne.

Le premier objectif permettra de se concentrer rapidement sur la zone du code qui consomme le plus de mémoire afin de réduire l'empreinte mémoire. Le second objectif permettra d'étudier la scalabilité mémoire du code et de dimensionner une simulation (par exemple savoir quelle taille de maillage peut être utilisée sur un nombre de nœuds donnés).

Ces objectifs ont orienté la bibliothèque LIBMTM développée dans le cadre de cette thèse. L'article [TT11] introduit 3 critères qui caractérisent un outil d'analyse : la précision, le caractère intrusif de la méthode et la facilité d'utilisation/d'interprétation. Par rapport à ces critères, la LIBMTM est précise et facile d'interprétation ; par contre sa mise en place dans une application est intrusive, car l'instrumentation se fait au niveau du source. Cela implique que l'application doit être recompilée une fois la phase d'instrumentation effectuée. À ce jour, nous ne connaissons pas d'autre moyen pour récupérer les informations qui sont nécessaires pour prédire la consommation mémoire à l'extérieur d'une application.

L'objectif de la LIBMTM étant d'étudier le comportement mémoire d'une application parallélisée en MPI/OPENMP, ces aspects ont dus être pris en considération dans la conception de la bibliothèque. Dans le but de pouvoir réutiliser la LIBMTM dans d'autres applications, nous nous sommes imposés la contrainte que cette bibliothèque soit utilisable en FORTRAN et en langage C.

2.2.1 Description de la mise en œuvre globale de la LIBMTM

Le besoin de prédire la consommation mémoire à l'extérieur de l'application nous pousse à opter pour une solution qui s'appuie sur une instrumentation au niveau du code source (voir la section 2.3.3 pour plus de détails sur le mécanisme de prédiction). La Figure 2.1 donne une vision globale des modifications à faire sur une application existante pour utiliser la LIBMTM. Sur cette figure, les tâches en ▭ font partie du cycle de développement standard d'une application, celles en ▭ sont relatives à l'utilisation de la bibliothèque LIBMTM. Ce schéma met aussi en avant les sorties en ▭ qui sont produites par le script de post-traitement de la LIBMTM.

L'utilisation de la bibliothèque LIBMTM interfère avec le cycle de développement d'une application à 3 niveaux : dans les sources, à la compilation et en post-traitement. On voit aussi sur ce schéma que l'ensemble des informations accessibles en post-traitement est contenu dans les traces. Cette observation rappelle le fait que les traces doivent être assez riches pour pouvoir interpréter aisément le comportement de l'application qu'on souhaite étudier.

2.2.2 Interface de programmation de la LIBMTM (API)

L'interface de programmation de la LIBMTM fournit un ensemble de macros-instructions (macros en abrégé dans toute la suite) à l'utilisateur. Nous avons fait ce choix pour pouvoir bénéficier du mécanisme d'expansion de macro à la compilation ; il est utilisé par beaucoup de macros pour récupérer des chaînes de caractères du code source. Cette spécificité apparait clairement pour les macros d'allocation.

L'utilisation de macros nous offre aussi la possibilité de désactiver les appels aux fonctions de la LIBMTM. Dans le cas où la bibliothèque est inutilisable, il suffit de définir la macro `MTM_DISABLE` au niveau de la compilation du code client. Dans ce cas, l'ensemble des macros `MTM` n'ont plus d'effet, hormis les macros relatives à l'allocation et à la libération de mémoire qui sont remplacées par des appels aux fonctions standards `allocate()` et `deallocate()` en FORTRAN.

Cette partie décrit les principales macros qui permettent d'instrumenter une application ; différents extraits du fichier d'entête `mtm_alloc.h` sont décrits dans cette section. Bien que la

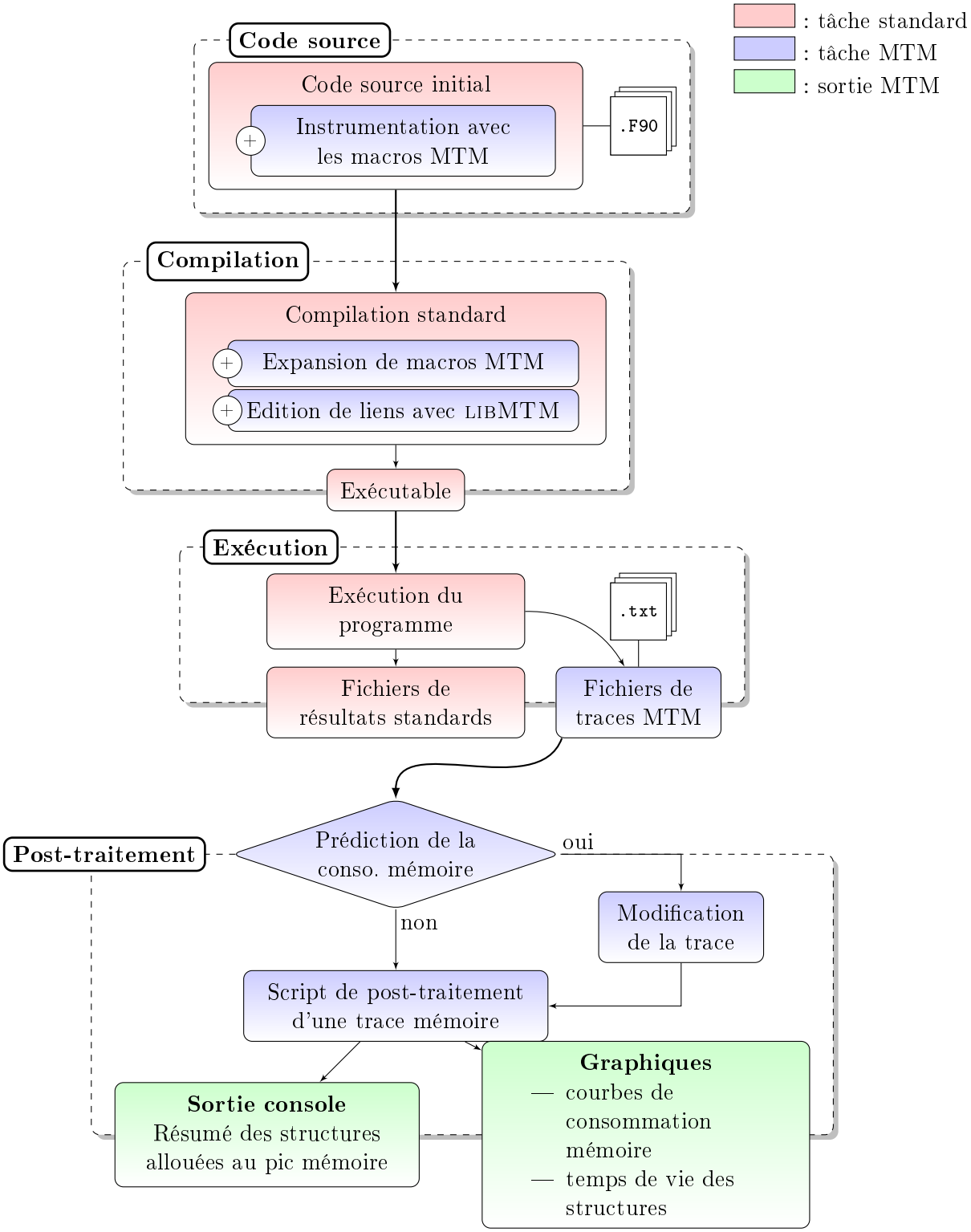


FIGURE 2.1 – Intégration de la LIBMTM dans le cycle de développement et de production d'une application.

bibliothèque soit utilisable en FORTRAN et en langage C, nous nous concentrons sur les macros utilisables en FORTRAN dans cette partie.

Le premier extrait ci-dessous concerne l'initialisation et la finalisation de la bibliothèque.

```
#define MTM_INIT(pid, pid_root)     \
  call MTM_do_init(pid, pid_root)
#define MTM_FINALIZE()        \
  call MTM_do_finalize()

#define MTM_OPTION(opt_arg, opt_value)    \
  call MTM_set_option(opt_arg, opt_value)
```

MTM_do_init : initialise des structures internes, identifie le processus `pid_root` comme le processus chargé de l'écriture d'éventuels messages d'erreur. Pour cette valeur de `pid` passée à cette fonction, un fichier de traces est créé. Cela permet d'avoir une trace par processus ;

MTM_do_finalize : vide les buffers d'écriture et ferme le fichier de traces associé au processus appelant la fonction ;

MTM_set_option : configure les options de la bibliothèque grâce aux chaînes de caractères `opt_arg` et `opt_value`. Cette fonction doit être appelée avant l'initialisation. Les différentes options sont :
- MTM_POWER : ["ON"|"OFF"] active ou désactive l'écriture de traces ;
- MTM_VERBOSITY : ["0"|"1"] choisit le degré de verbosité de la bibliothèque ;
- MTM_BINARY : ["true"|"false"] active/désactive l'écriture binaire ;
- MTM_FLUSH : ["true"|"false"] active/désactive la mise en buffer des écritures.

Les macros suivantes remplacent les fonctions FORTRAN `allocate()` et `deallocate()` :

```
#define MTM_DEFAULT  0
#define MTM_ZERO     1
#define MTM_NAN      2

#ifdef __GFORTRAN__
#define F_STR(arg) "arg"//char(0)
#else
#define F_STR(arg) #arg//char(0)
#endif
 [...]
#define MTM_ALLOC_1(ref_p, b_dim1,e_dim1, color)  \
  call MTM_do_alloc(ref_p, b_dim1,e_dim1, color, F_STR(ref_p), \
       F_STR(b_dim1),F_STR(e_dim1), \
       <INIT_PARAM>, __FILE__, __LINE__)

#define MTM_DEALLOC(p)        \
  call MTM_do_dealloc(p, __FILE__, __LINE__)
```

MTM_do_alloc : enregistre l'opération d'allocation et alloue le pointeur `ref_p` de l'indice `b_dim1` à l'indice `e_dim1`. Il est aussi possible de donner une `color` à l'allocation qui est utilisée en post-traitement. Il prend aussi en paramètre les chaînes des arguments `ref_p`, `b_dim1` et `e_dim1`. L'argument `<INIT_PARAM>` configure l'initialisation potentielle de la zone mémoire associée à `ref_p`. Il peut prendre 3 valeurs :
- MTM_DEFAULT : le tableau n'est pas initialisé ;
- MTM_ZERO : le tableau est initialisé à zéro ;

— MTM_NAN : le tableau est initialisé à *Not a Number* (NaN). Cette option est pratique pour détecter les tableaux qui sont utilisés sans être correctement initialisés (débogage) ;

MTM_do_dealloc : enregistre l'opération de libération et libère la mémoire associée au pointeur `ref_p`.

Ces fonctions sont écrites en FORTRAN à l'aide d'une interface. Les interfaces FORTRAN permettent de gérer différents types d'arguments ou un nombre variable d'arguments pour un même nom de fonction. Suivant le type et le nombre d'arguments, l'implémentation qui correspond à ce prototype est appelée. Cela implique qu'en fonction du nombre de dimensions et du type d'un tableau, une implémentation doit être réalisée. Le langage FORTRAN permet à l'utilisateur d'utiliser des tableaux de 1 à 7 dimensions. De plus, les types `pointer` et `allocatable` ne sont pas équivalents du point de vue de l'interface, c'est à dire que les fonctions qui prennent en argument un `pointer` ne peuvent pas être utilisées pour un argument de type `allocatable`. En pratique, la gestion des tableaux de type `allocatable` nécessite la duplication des fonctions qui gèrent le type `pointer` en ajustant le type de l'argument `ref_p`. La LIBMTM gère les entiers de 4 et 8 octets, les nombres flottants de simple et double précision et enfin les complexes de simple et double précision. La distinction de l'ensemble de ces prototypes nous a conduit à l'écriture de 84 fonctions quasi-identiques pour la gestion de l'allocation.

Le langage FORTRAN 90 ne fournit pas de type générique à l'utilisateur. Il n'existe pas d'équivalent au type générique `void *` du langage C en FORTRAN 90. Cela explique ce nombre élevé de fonctions qui réalisent la même opération pour différents types d'arguments.

On peut noter que la fonction de libération prend en argument uniquement le pointeur `p`. La taille du tableau ainsi que la chaîne de caractères associée à l'adresse `p` sont enregistrées dans une structure interne au moment de l'allocation. Au moment de la libération, cette structure interne est interrogée avec l'adresse `p` pour récupérer la taille de sa zone mémoire associée et ainsi permettre la mise à jour d'un compteur interne qui indique la consommation mémoire instantanée.

Les macros suivantes servent à instrumenter les entrées/sorties de fonction, les boucles qui réalisent des allocations, ainsi que les relations qui existent entre les bornes d'un tableau et les paramètres d'entrée de l'application. Ces informations sont essentielles pour pouvoir réaliser la prédiction de la consommation mémoire à l'extérieur du programme.

```
#define MTM_BLOCK_ENTER(str)      \
  call MTM_do_block_enter(str)
#define MTM_BLOCK_EXIT(str)     \
  call MTM_do_block_exit(str)

#define MTM_LOOP_BEGIN(index, l_bound, u_bound) \
  call MTM_do_loop_begin(F_STR(index), F_STR(l_bound), l_bound, \
       F_STR(u_bound), u_bound)
#define MTM_LOOP_END()        \
  call MTM_do_loop_end()

#define MTM_PARAM_SCALAR_RECORD(param, t_param) \
  call MTM_do_param_vector_record(param, F_STR(param),   1   , \
          t_param, __FILE__, __LINE__)
#define MTM_PARAM_VECTOR_RECORD(array, nb_elt, t_array) \
  call MTM_do_param_vector_record(array, F_STR(array), nb_elt, \
          t_array, __FILE__, __LINE__)

#define MTM_EXPR_RECORD(param, expr, t_param) \
  call MTM_do_expr_record(t_param, F_STR(param), F_STR(expr), \
```

```
21      __FILE__, __LINE__)
```

MTM_do_block_enter : enregistre l'entrée d'une fonction ;

MTM_do_block_exit : enregistre la sortie d'une fonction ;

MTM_do_loop_begin : enregistre le début d'une boucle. Le nom de l'indice de boucle est enregistré ainsi que la valeur et le nom des bornes de la boucle ;

MTM_do_loop_end : enregistre la fin de la boucle courante ;

MTM_do_param_vector_record : enregistre les valeurs, le nom, le nombre de cases `nb_elt` et le type du tableau `array` ;

MTM_do_expr_record : enregistre le type, le nom et l'expression qui permet de calculer la valeur de `param`.

Les appels des fonctions `MTM_do_block_enter()` et `MTM_do_block_exit()` peuvent être imbriqués. Ces enregistrements d'entrée et de sortie de fonction permettent de reconstruire une pile d'appels a posteriori. Cela est utile pour pouvoir localiser les opérations mémoire dans le code, et particulièrement le pic mémoire.

L'enregistrement des informations concernant les boucles servent à reproduire leur comportement à l'extérieur du programme. Les boucles qui sont d'intérêt pour la LIBMTM sont uniquement celles qui contiennent des allocations ou libérations mémoire.

Les deux fonctions `MTM_do_param_vector_record` et `MTM_do_expr_record` servent à enregistrer des informations sur les variables qui interviennent dans le calcul de la taille des tableaux. Elles sont particulièrement précieuses pour la prédiction mémoire. Le mécanisme qui s'appuie sur ces informations est détaillé en section 2.3.3. Les types de données sont à spécifier grâce aux macros `MTM_I32`, `MTM_I64`, `MTM_F32` et `MTM_F64` qui désignent respectivement les entiers 32 et 64 bits et les flottants 32 et 64 bits.

La section suivante donne un exemple d'utilisation des macros de la LIBMTM sur un programme simple en FORTRAN.

```
1
2
3  program hello
4
5
6    implicit  None
7
8    integer  ::  Nr
9    integer, dimension(:), pointer  ::  tab
10
11   Nr = 10
12
13
14
15
16
17
18
19   allocate  ( tab( 1, Nr ) )
20   tab = 0
21   deallocate  ( tab )
22
23
24
25
26
27 end program hello
```

Listing 2.1 – Sans instrumentation.

```
1  #include "mtm_alloc.h"
2
3  program hello
4
5    use mtm_alloc
6    implicit  None
7
8    integer  ::  Nr
9    integer, dimension(:), pointer  ::  tab
10
11   Nr = 10
12
13   MTM_INIT( 0, 0 )
14
15   MTM_BLOCK_ENTER("hello")
16
17   MTM_PARAM_SCALAR_RECORD(Nr, MTM_I32)
18
19   MTM_ALLOC_1_ZERO(tab, 1, Nr, 1)
20
21   MTM_DEALLOC(tab)
22
23   MTM_BLOCK_EXIT("hello")
24
25   MTM_FINALIZE()
26
27 end program hello
```

Listing 2.2 – Avec instrumentation.

2.2.3 Exemple d'utilisation de la LIBMTM

Afin de mieux saisir l'utilisation de la LIBMTM, cette section présente un programme simple en FORTRAN qui utilise la LIBMTM.

Le programme sur le Listing 2.1 consiste simplement à allouer un tableau, à l'initialiser et à le libérer. Le programme sur le Listing 2.2 donne la version instrumentée par la LIBMTM du même programme. On peut noter aux lignes 1 et 5 l'inclusion du fichier d'entête `mtm_alloc.h` qui contient la définition des macros de la LIBMTM et l'utilisation du module `mtm_alloc` qui permet de contrôler le type des arguments qui sont passés aux fonctions `MTM` au niveau de la compilation. Le fichier d'entête et le module `MTM` sont fournis dans le répertoire `include` du dossier d'installation de la LIBMTM.

L'instrumentation que propose la LIBMTM est puissante car elle permet d'extraire le comportement mémoire d'une application, mais en contrepartie elle est intrusive.

2.2.4 Format des fichiers de traces

Les fichiers de traces générés par la LIBMTM contiennent principalement l'enchainement chronologique des allocations/libérations mémoire, des entrées/sorties des fonctions instrumentées et l'enregistrement des valeurs ou relations entre les paramètres. Afin de limiter les modifications au niveau de la lecture d'une trace en post-traitement, la lecture s'appuie sur le fichier `mtm_header.out` qui décrit le format des différentes lignes de la trace. Ce fichier est généré au moment de l'exécution par la fonction `MTM_do_init()`.

```
1 BEGIN FNCT_NAME
```

```
2 END FNCT_NAME
3 ALLOC PNTR_NAME PNTR_ADDR PNTR_SIZE NB_ELT_EXPR ELT_SIZE COLOR FILE_NAME
      LINE_NB
4 FREE PNTR_ADDR FILE_NAME LINE_NB
5 ARRAY ARRAY_TYPE ARRAY_NAME NB_ELT ARRAY_ELT_VALUE FILE_NAME LINE_NB
6 EXPR PARAM_TYPE PARAM_NAME EXPR_STR FILE_NAME LINE_NB
7 LOOP_BEGIN INDEX L_BOUND L_BOUND_VALUE U_BOUND U_BOUND_VALUE
8 LOOP_END
```

Listing 2.3 – Extrait du fichier `mtm_header.out`.

Le Listing 2.3 donne un extrait du fichier `mtm_header.out`. Chaque mot en début de ligne correspond à un mot-clé (en vert). Les lignes d'une trace `MTM` qui ne commencent pas par un de ces mots-clés ne sont pas interprétées par le script de post-traitement. Chaque mot-clé peut être suivi par un ensemble de sous mots-clés. Pour chaque sous mot-clé, le fichier `mtm_header.out` définit son type. Les définitions du type des sous mots-clés n'apparaissent pas dans le Listing 2.3 par commodité, car elles sont assez simples puisqu'elles correspondent à une liste de lignes de type *sous mot-clé=type*. La liste des types disponibles est donnée dans le Tableau 2.1.

type	sémantique	octet
c	caractère	1
s	chaîne de caractères	$n \in \mathbb{N}$
i8	entier	1
i16	entier	2
i32	entier	4
i64	entier	8
f32	flottant	4
f64	flottant	8
p	pointeur	8

TABLE 2.1 – Les différents types possibles des sous mots-clés.

Grâce à ce fichier, le script de post-traitement de trace décode le fichier de traces, qu'il soit écrit en ASCII ou en binaire.

Le programme présenté sur le Listing 2.2 génère la trace suivante :

```
1 # Processus : 0
2
3 BEGIN hello
4 ARRAY 0 Nr  1 10    simple.F90  23
5 ALLOC tab 0x1570250 40  (Nr-1+1)  4 1 simple.F90  25
6 FREE  0x1570250 simple.F90  27
7 END hello
```

Listing 2.4 – Exemple de trace `MTM`.

Le taille mémoire consommée par un tableau peut être calculée en multipliant le nombre de cases du tableau par la taille du type de base qu'il contient. Dans la trace ci-dessus, on peut voir qu'au niveau de la ligne de l'allocation, le nombre de cases du tableau `tab` est exprimé par la formule : `(Nr-1+1)`. On voit donc clairement la dépendance entre la quantité mémoire consommée par le tableau et le paramètre `Nr`. Pour que le calcul de cette expression puisse se faire, le paramètre `Nr` doit être défini préalablement, comme on peut le voir à la ligne 4. En

s'appuyant sur le Listing 2.3, on peut identifier sur cette ligne les paramètres `ARRAY_TYPE = 0`, `ARRAY_NAME = Nr`, `NB_ELT = 1` et `ARRAY_ELT_VALUE = 10`. Donc $Nr = 10$.

Lors de l'instrumentation d'une application parallèle, chaque processus MPI doit appeler la fonction d'initialisation `MTM_do_init` avec un argument `pid` différent par processus. Cela permet à la bibliothèque de générer une trace par processus.

Des détails concernant les contraintes de conception de la LIBMTM sont donnés dans l'annexe A. La section suivante décrit la phase de post-traitement de la bibliothèque.

2.3 Analyse post-mortem des traces

Les traces `MTM` reflètent l'évolution temporelle de la consommation mémoire d'une application. Les opérations mémoire instrumentées étant restreintes au code source de l'application cliente, les consommations mémoire dues à des bibliothèques externes ne sont pas prises en compte dans notre approche. Afin d'interpréter simplement les informations recueillies dans les traces, la LIBMTM fournit un script de post-traitement, `view_memory`, disponible dans le répertoire `bin` du dossier d'installation.

Cette section donne une vue d'ensemble du fonctionnement interne du script de post-traitement en mettant en avant les points clés qui permettent de réaliser la prédiction de consommation mémoire en mode hors ligne. Dans un deuxième temps, des exemples de sorties générées par ce script sont donnés.

2.3.1 Implémentation du script de post-traitement

Le script `view_memory` prend comme argument obligatoire le chemin du répertoire qui contient le fichier d'entête `mtm_header.out` ainsi que l'ensemble des traces `MTM`. Ce script gère actuellement un seul fichier de traces à la fois. Par défaut, la trace du processus 0 est traitée. Il est possible de spécifier un autre fichier de traces grâce à l'option `-f`.

Le fonctionnement du script de post-traitement se décompose en les étapes suivantes :

Etape 1 : lecture du fichier d'entête `mtm_header.out` ;

Etape 2 : lecture d'un fichier de traces ;

Etape 3 : simplification du contenu de la trace :
- suppression des lignes qui ne commencent pas par un mot-clé (voir le Listing 2.3) ;
- suppression des entrées/sorties de fonction qui ne contiennent pas d'opération mémoire ;
- réécriture des blocs de boucle ;
- évaluation des tailles des allocations en fonctions des paramètres d'entrée ;

Etape 4 : interprétation de la trace simplifiée ;
- calcul du pic mémoire et des structures qui sont allouées à ce moment là ;
- calcul des différents graphiques.

L'ensemble des étapes du post-traitement n'est pas détaillé ici. Seuls les éléments qui permettent de comprendre le mécanisme de prédiction sont détaillés. Ces éléments sont contenus dans la troisième étape, particulièrement au moment de l'évaluation des tailles des allocations.

En post-traitement, la trace est lue ligne par ligne. Chaque ligne qui commence par un mot clé déclenche un traitement correspondant. Les lignes de types `ARRAY` et `EXPR` servent à définir ou modifier la valeur d'une variable. En interne, le script `view_memory` maintient le dictionnaire `dict_arg` qui contient l'association `variable -> valeur`. Grâce à ce dictionnaire, les variables

définies par les lignes `EXPR` peuvent être calculées. Le dictionnaire `dict_arg` fournit la valeur des variables qui interviennent dans l'expression si elle est disponible. Dans le cas contraire, une exception est levée ; elle déclenche l'affichage d'un message d'erreur.

Lors du traitement d'une ligne de type `ALLOC`, l'évaluation de la formule qui donne le nombre d'éléments du tableau est traitée de la même façon que pour les lignes `EXPR`. Une fois que l'évaluation du nombre de cases est faite, on vérifie que la taille du tableau calculée de façon formelle est égale à la taille du tableau enregistrée au moment de l'exécution. Cette vérification est nécessaire pour valider la trace, et donc pour s'assurer que l'instrumentation de l'application cliente est suffisante. Dans le cas contraire, une exception est levée ; elle déclenche l'affichage d'un message d'erreur.

```
# Processus : 0

BEGIN hello
ARRAY 0 Nr  1 10    simple.F90  23
## dict_arg = { "Nr" : 10 }
ALLOC tab 0x1570250 40  (Nr-1+1)  4 1 simple.F90  25
## PNTR_SIZE = 40 octets
## NB_ELT_EXPR = (Nr-1+1) = (10-1+1) = 10
## ELT_SIZE = 4
## size( tab ) = NB_ELT_EXPR * ELT_SIZE = 40 octets
## Est-ce que ( PNTR_SIZE == size( tab ) ) ? oui
FREE  0x1570250 simple.F90  27
END hello
```

Listing 2.5 – Trace `MTM` commentée.

Le Listing 2.5 ajoute des commentaires à l'exemple de trace 2.4. Ces commentaires montrent l'état du dictionnaire `dict_arg` et les étape du calcul formel de la taille d'un tableau. Les sous mots-clés `PNTR_SIZE`, `NB_ELT_EXPR` et `ELT_SIZE` qui apparaissent sont définis dans le Listing 2.3. On constate sur cet exemple simple que le calcul de la taille du tableau `tab` correspond effectivement à la taille `PNTR_SIZE` enregistrée au moment de l'exécution.

2.3.2 Les sorties générées au post-traitement

Afin d'analyser finement la consommation mémoire, le script `view_memory` génère deux types de sortie : (`i`) texte sur le terminal et (`ii`) des graphiques dans des fichiers. Ces informations aident le développeur à comprendre le coût mémoire des algorithmes qu'il manipule et lui donnent des indices sur comment et où il peut appliquer des modifications pour diminuer l'empreinte mémoire.

Exemple de sortie sur le terminal

```
 Variables allocated at the memory peak :
               40 (  40.0   B) tab
   ************************************************************
  Total:       40 (  40.0   B)

 The consumption of the different kinds of structure at the memory peak :
   ---------------------------------------------------------
      COLOR  :  MEMORY SIZE                    : PERCENTAGE
   .........................................................
         1  :  40.0 Bytes (         40) : 100.0 %
```

```
  ------------------------------------------------------------
 Total : 40.0 Bytes (          40)

MTM block stack at the memory peak :
  hello

Memory peak :           40 (    40.0 Bytes)
```

Listing 2.6 – Sortie sur le terminal.

Le Listing 2.6 donne la sortie terminale obtenue sur la trace 2.4. Cette sortie est pratique pour situer où se produit le pic mémoire, quelle est sa taille et quelles sont les structures allouées à ce moment dans l'application. Une synthèse par couleur des structures allouées au pic mémoire est aussi faite. Cette synthèse nous permet par exemple sur GYSELA de distinguer la consommation mémoire des tableaux $1D$, $2D$, $3D$ et $4D$.

Dans notre exemple simpliste où un seul tableau est alloué, le pic mémoire est de 40 octets et se situe dans la fonction `hello`. Ce nom de fonction est enregistré dans la trace par les macros d'instrumentation d'entrées/sorties de fonction.

Exemple de sortie graphique

Pour pouvoir avoir une vue d'ensemble rapidement de la consommation mémoire, le script `view_memory` génère deux types de graphiques.

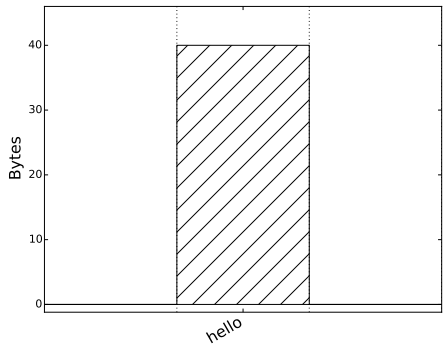


FIGURE 2.2 – L'évolution de la consommation mémoire durant l'exécution du programme `hello`

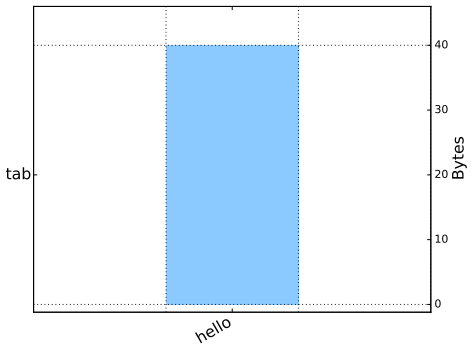


FIGURE 2.3 – Allocation et libération du tableau utilisé durant l'exécution du programme `hello`

La Figure 2.2 représente graphiquement la consommation mémoire des allocations instrumentées. L'axe des abscisses est chronologique. Il représente les entrées/sorties au cours du temps des sous-routines instrumentées. L'axe des ordonnées donne la consommation en octets. La Figure 2.3 montre la durée de vie des structures ainsi que l'endroit où elles sont allouées ou libérées. L'axe des abscisses reste identique au graphique précédent et l'axe des ordonnées affiche les noms des tableaux. L'allocation d'un tableau est représentée graphiquement par un rectangle (ici en bleu) dans la ligne réservée à ce tableau. Plus un rectangle est large, plus longue est sa durée de vie. L'espace mémoire consommée par un tableau est proportionnel à sa hauteur.

Les graphiques qu'on obtient sur les traces générées par GYSELA sont similaires aux Figures 2.4 et 2.5. Sur la Figure 2.4, on peut localiser dans quelle sous-routine le pic mémoire est atteint. Sur la Figure 2.5, on peut ensuite identifier les tableaux qui sont effectivement alloués

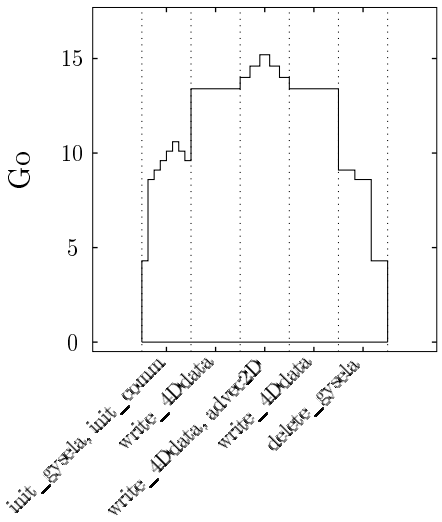


FIGURE 2.4 – L'évolution de la consommation mémoire durant l'exécution d'une courte simulation de GYSELA.

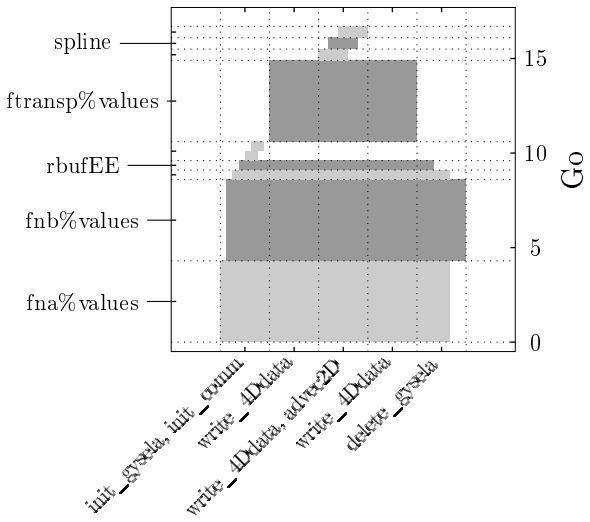


FIGURE 2.5 – Allocations et désallocations des tableaux utilisés dans diverses fonctions de GYSELA.

quand le pic mémoire est atteint. Grâce à ces informations, nous savons exactement où modifier le code pour réduire son pic mémoire.

Les traces `MTM` peuvent être de l'ordre de quelques Mo. Afin de pouvoir parcourir efficacement les graphiques, il est possible de les générer sur une seule figure et d'afficher ce graphique en mode interactif. Ce mode interactif permet d'agrandir et de translater la figure, ce qui se révèle pratique lors de la visualisation d'une grande trace. Cette fonctionnalité a été développée en PYTHON grâce à l'extension MATPLOTLIB [Hun07].

2.3.3 Prédiction de la consommation mémoire

Anticiper la consommation mémoire d'une application avant de la soumettre sur un supercalculateur est un des objectifs de la LIBMTM. Cette prédiction se fait sur un jeu de paramètres d'entrée donné.

Déterminer le jeu de paramètres d'entrée qui impacte la consommation mémoire d'une application peut être une tâche délicate à réaliser. L'utilisation de la LIBMTM permet de construire ce jeu de paramètres de façon itérative. Les paramètres d'intérêt pour l'étude de la consommation mémoire sont ceux qui interviennent dans la taille des tableaux alloués. Dans l'application cliente, la valeur de ces paramètres est définie soit :

— par lecture d'un fichier d'entrée (typiquement les paramètres de la simulation à exécuter) ;
— par une valeur définie par une constante dans l'application ;
— par un calcul dépendant des variables de l'application.

Ces valeurs et relations doivent être instrumentées à l'aide des macros `MTM_PARAM_SCALAR_RECORD`, `MTM_PARAM_VECTOR_RECORD` et `MTM_EXPR_RECORD` (voir section 2.2.2) par l'utilisateur afin que le mécanisme de prédiction soit fidèle au comportement de l'application. Cette tâche qui est laissée à la charge du développeur est essentielle pour assurer une prédiction mémoire correcte. Les messages d'erreur affichés par le script de post-traitement `view_memory` sont d'une aide précieuse pour la réussite de cette étape.

Grâce aux expressions des tailles de tableaux et aux paramètres numériques contenus dans

le fichier trace, nous pouvons modéliser le comportement mémoire hors ligne. L'idée ici est de reproduire les allocations avec n'importe quel ensemble de paramètres d'entrée. Les variables qui sont enregistrées grâce aux macros `MTM_PARAM_SCALAR_RECORD` ou `MTM_PARAM_VECTOR_RECORD` constituent les paramètres d'entrée du processus de prédiction mémoire. En changeant la valeur d'un de ces paramètres, les tailles de tableaux qui en dépendent s'ajustent.

```
1  # Processus : 0
2
3  BEGIN hello
4  ARRAY 0 Nr  1 20    simple.F90  23
5  ## dict_arg = { "Nr" : 20 }
6  ALLOC tab 0x1570250 40  (Nr-1+1)  4 1 simple.F90  25
7  ## PNTR_SIZE = 40 octets
8  ## NB_ELT_EXPR = (Nr-1+1) = (20-1+1) = 20
9  ## ELT_SIZE = 4
10 ## size( tab ) = NB_ELT_EXPR * ELT_SIZE = 80 octets
11 ## Est-ce que ( PNTR_SIZE == size( tab ) ) ? non
12 FREE  0x1570250 simple.F90  27
13 END hello
```

Listing 2.7 – Modification d'un paramètre d'entrée d'une trace `MTM` – Nr vaut 20 au lieu de 10.

La trace présentée par le Listing 2.7 correspond à la trace 2.4 ayant été modifiée avec un éditeur de texte. Dans cette trace modifiée, le paramètre $Nr = 20$ au lieu de 10. Avec cette nouvelle valeur, la taille du tableau `tab` est évaluée à 80 octets. On voit aussi que tester la correspondance entre la taille effective `PNTR_SIZE` et la taille calculée durant l'interprétation de la trace n'a plus de sens. L'option `-p` permet de désactiver l'affichage des messages d'erreur dus à ce type de test qui n'ont plus de sens.

C'est grâce au mécanisme décrit ci-dessus que nous pouvons faire de la prédiction de consommation mémoire en mode hors ligne. Sur cette trace modifiée, la sortie sur terminal et les graphiques peuvent être générés. Ce mécanisme nous donne donc accès virtuellement à l'étude de la consommation mémoire d'une application pour différentes tailles de maillages par exemple, ou encore pour différentes tailles de déploiements MPI/OpenMP.

On peut naturellement se demander quel est le degré de précision de la prédiction du comportement mémoire par rapport aux simulations réelles. Pour valider l'outil de prédiction, nous vérifions les résultats sur différents cas tests. La procédure de vérification est la suivante :

1. fixer un jeu de paramètres d'entrée et lancer la simulation correspondante ;
2. récupérer la trace mémoire, *tm1*, associée à ces paramètres ;
3. doubler la valeur d'un des paramètres d'entrée (une dimension du maillage par exemple) ;
4. lancer la simulation correspondante à ce nouveau jeu de paramètres ;
5. comparer la prédiction du pic mémoire sur la trace *tm1* avec le paramètre d'entrée modifié et la valeur du pic mémoire réel obtenue sur la seconde simulation.

Ces tests nous ont permis de valider l'outil de prédiction dans de nombreuses configurations. La prédiction du pic mémoire correspond exactement à celui qui est effectivement atteint durant la simulation. Un test reproduisant ce protocole est lancé automatiquement de façon régulière pour s'assurer de la pérennité de l'instrumentation dans Gysela. Les résultats obtenus grâce à cet outil sont présentés dans la section 3.4.1.

2.3.4 Extensions et limitations

L'objet d'étude de la LIBMTM est la consommation mémoire d'une application ; néanmoins, l'approche que nous avons mis en œuvre ne permet pas de mesurer toutes les consommations

mémoire liées à l'exécution d'une application. Typiquement les zones mémoires qui sont allouées sur la pile (par exemple les variables locales aux fonctions, les piles locales aux threads) ou les zones allouées par des bibliothèques de tierce partie utilisées par l'application ne sont pas comptabilisées. Seuls les tableaux de certains types primitifs alloués sur le tas sont pris en compte. Néanmoins, ce sont ces allocations qui représentent pour une grande classe d'applications une grande portion de l'utilisation de la mémoire.

Les consommations qui ne sont pas capturées par les macros de la libMTM restent mesurables de façon indirecte en récupérant les informations liées au processus auprès du système d'exploitation [8]. Pour chaque processus, le système d'exploitation comptabilise l'ensemble des consommations mémoire qui lui correspondent ; cela inclut les allocations mémoires effectuées par des bibliothèques externes à une application. Le relevé de cette consommation au niveau du système nous permettrait de quantifier la quantité globale de mémoire réellement utilisée par un programme. Cela nous permettrait aussi de savoir quelle est la portion de la consommation mémoire instrumentée par la libMTM.

L'utilisation de la bibliothèque libMTM ne se restreint pas à l'application Gysela. Elle peut être utilisée par d'autres applications en langage C ou Fortran. Par contre, pour bénéficier de la prédiction de consommation mémoire en mode hors ligne, certaines conditions doivent être satisfaites par l'application cliente. Les bornes des tableaux instrumentés dont on souhaite prédire le comportement à l'extérieur de l'application ne doivent pas dépendre de grandeurs calculatoires produites par l'application. Par exemple, si une allocation dépend d'une grandeur physique produite par l'application, il n'est pas possible de reproduire fidèlement cette allocation à l'extérieur de l'application car la taille de l'allocation dépend d'une valeur qui est accessible uniquement en exécutant effectivement l'application.

On peut considérer que les valeurs non calculables à l'extérieur d'une application sont celles pour lesquelles une grande partie de l'application cliente doit être reproduite afin de les calculer. Les valeurs qui ne sont pas calculables de façon déterministe ne peuvent pas être capturées par le mécanisme de prédiction. Seules les variables qui sont calculables de façon déterministe et assez simplement permettent de réaliser une prédiction mémoire précise. C'est le cas de l'application Gysela. Il est aussi possible de ne pas instrumenter les tableaux dont le comportement est difficile à prédire à l'extérieur de l'application à partir du moment où on peut borner leurs tailles.

Nous pouvons ainsi capturer efficacement le comportement mémoire des programmes qui utilisent un maillage statique avec une décomposition de domaine. Par statique, nous entendons plus précisément que le nombre de points du maillage ne varie pas en fonction des valeurs calculées durant l'exécution. Par opposition, la prédiction de consommation mémoire des applications qui utilisent des maillages non réguliers avec une méthode de raffinement dynamique n'est pas, par exemple, facilement envisageable dans notre approche.

Dans les cas où des dépendances entre les valeurs liées à une exécution et la consommation mémoire de l'application sont fortes, la prédiction hors ligne de la consommation mémoire est délicate. Une prédiction du pic mémoire dans le pire des cas sur ces applications pourrait permettre de savoir si l'application pourra tenir en mémoire ou pas.

Malgré le fait qu'une application soit difficile à instrumenter pour bénéficier de l'outil de prédiction mémoire, les traces `MTM` sont toujours utiles pour analyser finement l'évolution temporelle de la consommation mémoire d'une application. Nous avons détecté différentes situations où une instrumentation qui permet de réaliser une prédiction précise de la consommation mémoire est laborieuse à mettre en place. Des exemples de ces situations sont

8. Sur les systèmes Linux : `http://www.mjmwired.net/kernel/Documentation/filesystems/proc.txt`.

donnés dans l'annexe B.

Conclusion et perspectives

La bibliothèque LIBMTM permet de suivre la consommation mémoire d'une application et de prédire sa consommation à condition que l'instrumentation soit correcte et que l'application respecte les différentes conditions évoquées dans la section 2.3.4. La possibilité de prédire la consommation mémoire donne un accès facile à des études pertinentes pour des applications parallélisées, telle que la scalabilité mémoire. Les études que nous avons réalisées sur GYSELA font l'objet du chapitre suivant. Ces études peuvent s'effectuer sur d'autres applications puisque la LIBMTM n'est pas dédiée à GYSELA.

Parmi les points forts, on peut rappeler que la bibliothèque a été conçue de façon à être portable. Un autre point fort est la génération de graphiques qui permettent d'avoir une vision d'ensemble et précise de la consommation mémoire. Ces graphiques sont aussi pratiques comme support de communication, de réflexion et de prise de décision par rapport au développement d'une application. Ils stimulent aussi le dialogue entre les acteurs d'un projet. La LIBMTM est utilisée dans les simulations en production par GYSELA, ce qui montre une bonne maturité de la bibliothèque. Les outils fournis par la LIBMTM permettent de mieux comprendre et d'analyser une application dans différentes configurations.

Différents verrous techniques sont néanmoins présents dans l'implémentation actuelle de la LIBMTM. On peut évoquer le fait que la bibliothèque ne soit actuellement pas *thread-safe*, ou que la visualisation doive être améliorée pour faciliter l'interprétation de grandes traces, ou encore que les traces pourraient bénéficier d'une compression pour réduire la taille des fichiers produits. Ces aspects ne faisaient pas partie des priorités.

Un grand nombre des points techniques du fonctionnement interne de la LIBMTM ont été abordés dans ce chapitre. Le chapitre suivant aborde les différentes études réalisables grâce à cette technologie.

Chapitre 3

Réduction et étude de la consommation mémoire grâce aux outils de la libMTM

Sommaire

Il existe au moins deux démarches pour réduire l'empreinte mémoire par processus d'une application parallèle. Premièrement, on peut augmenter le nombre de processus utilisés par une simulation. La taille des structures qui bénéficient d'une décomposition de domaine devrait diminuer lorsque le nombre de processus MPI augmente. Deuxièmement, on peut gérer plus finement les allocations et libérations des tableaux afin de réduire leur impact sur le pic mémoire. Ainsi, limiter l'impact des structures qui ne bénéficient pas d'une décomposition de domaine est susceptible d'améliorer la scalabilité mémoire.

Pour réussir à réduire l'empreinte mémoire et à repousser les limitations dues à la consommation mémoire, nous avons choisi de nous concentrer sur la seconde approche. Dans la version initiale du code Gysela, une grande partie des variables était allouée durant la phase d'initialisation. Ce choix est légitime pour les variables de type *persistante* utilisées durant une grande partie de chacune des itérations temporelles, variables qu'il faut distinguer des variables de type *temporaire* qui servent généralement à stocker des résultats intermédiaires. Les variables persistantes bénéficient dans la plupart des cas d'une décomposition de domaine, ce qui n'est pas

le cas des variables temporaires dans GYSELA. En se basant sur le Tableau 1.1, on peut identifier les structures $3D$ et $4D$ comme étant des variables persistantes, alors que les structures $1D$ et $2D$ sont de nature temporaire. La consommation mémoire globale de l'application sur l'ensemble des ressources utilisées est de 135 To sur le plus grand cas avec 2048 processus MPI.

Effectuer l'ensemble des allocations dans la phase d'initialisation permet de déterminer la consommation mémoire d'une simulation rapidement dès son lancement, et ce sans avoir à l'exécuter entièrement. Dans ce cas, il est possible de savoir si la simulation qu'on souhaite réaliser peut s'exécuter ou pas en peu de temps. Deuxièmement, cette configuration évite les potentiels surcoûts en temps d'exécution dus à une gestion dynamique des allocations. Le principal désavantage de cette approche repose alors sur le fait que les variables qui sont utilisées de façon temporaire dans quelques fonctions occupent leur espace mémoire durant l'intégralité d'une simulation.

La consommation mémoire devenant un point critique lorsqu'un grand nombre de cœurs est utilisé, nous gérons un grand sous-ensemble des variables temporaires de GYSELA de façon dynamique. Néanmoins, avec une utilisation plus importante des allocations dynamiques, nous perdons les deux principaux avantages des allocations statiques, en particulier la possibilité de connaître la consommation mémoire d'une simulation dès le lancement. Grâce à la LIBMTM, nous aurons à nouveau la possibilité de prédiction de la consommation mémoire d'une simulation tout en utilisant des allocations dynamiques.

Ce chapitre présente les améliorations que nous avons apportées à l'application GYSELA pour réduire sa consommation mémoire. Il se décompose selon le plan suivant. La section 3.1 définit la méthode de réduction de l'empreinte mémoire que nous avons utilisée. La section 3.2 décrit différentes itérations de la méthode IRPM. La section 3.3 propose une étude des surcoûts dus à la gestion dynamique de la mémoire. Enfin la section 3.4 présente différentes études qui sont accessibles grâce à l'outil de prédiction de la LIBMTM.

3.1 Méthode incrémentale de réduction du pic mémoire (IRPM)

La méthode que nous avons appliquée pour réduire le pic mémoire de l'application GYSELA se décrit de la façon suivante :

Méthode Incrémentale de Réduction du Pic Mémoire

Etape 0 Déterminer le jeu de paramètres de la simulation qu'on souhaite réaliser ;
Etape 1 Localiser le pic mémoire au niveau du code source ;
Etape 2 Identifier les structures de données qui sont allouées lorsque le pic mémoire est atteint ;
Etape 3 Evaluer quelle structure n'est pas indispensable au pic. Si aucune des structures ne peut être libérée, les algorithmes utilisés sont à remettre en cause en vue de réduire leur consommation mémoire ;
Etape 4 Après modification du code source (libérations mémoire et/ou modification d'algorithme), retour à l'**étape 0**.

Chaque étape de la méthode Incrémentale de Réduction du Pic Mémoire (IRPM) nécessite une attention particulière. L'**étape 0** de cette méthode a un impact direct sur l'ensemble des étapes suivantes. Les paramètres les plus influents sur la consommation mémoire de GYSELA sont la taille du maillage et le nombre de processus MPI. La localisation du pic mémoire dans le code dépend de ces paramètres. Par conséquent, l'endroit où doivent se concentrer les efforts

de minimisation de l'empreinte mémoire dépend de la simulation cible à exécuter. L'outil de prédiction (voir section 2.3.3) nous permet facilement d'explorer à la fois des petites et des grandes configurations à partir de traces `MTM` obtenues sur un cas de référence. Cette fonctionnalité fournit les informations nécessaires aux étapes suivantes.

L'**étape 1** consiste à localiser le pic mémoire ; cela correspond à l'identification des lignes du code où le pic mémoire est atteint. La fonction où le pic mémoire est atteint peut naturellement être appelée à différent endroit du code. Il est donc souvent utile de connaître la pile d'appel où le pic mémoire est atteint. Pour pouvoir lister les structures allouées dans l'**étape 2**, l'ensemble des allocations/libérations mémoire de l'application doivent être suivies. Ces deux étapes sont aisées à réaliser grâce à l'outil de post-traitement de la libMTM (voir section 2.3.2).

À l'**étape 3**, le développeur doit prendre en considération les dépendances de données pour comprendre quelles sont les structures qui peuvent être potentiellement libérées lorsque le pic mémoire est atteint. Il y a plusieurs moyens de réduire l'empreinte mémoire. Il est possible de réutiliser un espace mémoire existant, ou de faire plus de calculs au lieu de stocker des résultats, ou de remarquer une duplication de données qui peut être évitée, ou encore de remarquer qu'il est possible de libérer certains tampons mémoire durant un court laps de temps. Si aucune de ces méthodes n'est applicable, il est possible que la consommation mémoire soit intrinsèquement liée au schéma de calcul utilisé. Dans ce cas, le seul moyen de réduire la consommation mémoire est de changer l'algorithme et/ou la méthode numérique associée. Ce type de modification est envisageable à moyen terme ou long terme. Dans l'application Gysela, nous nous sommes concentrés sur la réduction de l'impact des structures qui ne bénéficient pas de la décomposition de domaine. Ce processus fait l'objet de la section 3.2.1.

Après modification du code source à l'**étape 4**, on peut observer un éventuel déplacement du pic mémoire. À la prochaine itération de la méthode IRPM, les efforts de développements se concentreront possiblement sur une autre partie de l'application. Le processus de réduction de l'empreinte mémoire se termine selon l'appréciation du développeur.

3.2 Application de la méthode IRPM

Dans cette section nous présentons différentes itérations de la méthode IRPM sur l'application Gysela. Nous donnons plus de détails sur la dernière itération qui est la plus délicate à réaliser.

3.2.1 Itérations de la méthode IRPM

Pour réduire l'empreinte mémoire de Gysela, nous avons donc suivi la méthode IRPM. Les changements que nous avons réalisés dans ce processus de réduction consistent à déplacer des allocations/libérations mémoire d'un ensemble de structures de données et à changer certains algorithmes. Pour suivre les déplacements du pic mémoire ainsi que la quantité de mémoire utilisée, nous avons défini une simulation de référence qui utilise 64 processus MPI et 16 threads par processus, soit un total de 1024 cœurs de calcul. Dans cette partie, nous suivons les évolutions de la consommation mémoire en observant la valeur du pic mémoire pour un processus MPI. Le maillage du cas de référence est le suivant :

$$N_r = 512,\ N_\theta = 1024,\ N_\varphi = 128,\ N_{v_\parallel} = 128,\ N_\mu = 2\,. \tag{3.1}$$

Dans la version initiale de Gysela, une grande partie des structures de données est allouée durant la phase d'initialisation et libérée à la sortie du code. Grâce à l'étude de la section 1.3.3 (p. 15), on se rend compte que les structures $1D$ et $2D$ représentent une partie de la mémoire

allouée pour des simulations sur grande configuration. Dans cette version initiale, le pic mémoire de GYSELA est de 14.14 Go.

Nous avons considéré dans un premier temps les structures de données qui sont pénalisantes sur les grandes configurations, à savoir les structures $1D$ et $2D$. Elles stockent généralement des résultats intermédiaires. Nous avons constaté qu'elles consommaient une grande partie de la mémoire sur les configurations avec un grand nombre de processus MPI. Pour y remédier, nous avons déplacé les allocations de ces structures au plus proche de leur utilisation. L'esprit de cette démarche est d'allouer de la mémoire uniquement lorsque cette dernière est utilisée. La gestion dynamique de la mémoire pouvant entraîner des surcoûts en temps d'exécution lors du déplacement des allocations/libérations mémoire, nous avons suivi l'évolution du temps d'exécution global du cas de référence. Nous avons conclu que les allocations et libérations placées respectivement juste avant et après les zones OPENMP n'entraînaient pas de surcoûts significatifs. Une étude approfondie de ces potentiels surcoûts est faite en section 3.3. Dans cette nouvelle version où un ensemble de tableaux est alloué de façon dynamique, le pic mémoire est de 13.53 Go.

Durant une itération de la méthode IRPM, nous avons remarqué que certains tableaux $2D$ qui stockent des coefficients de spline sont alloués durant la totalité d'une simulation alors qu'ils ne sont utilisés que lors de l'exécution du solveur de Vlasov (le noyau de calcul principal de GYSELA). De la même façon que pour l'optimisation précédente, nous avons déplacé les allocations/libérations de coefficients de spline au plus près de leur utilisation. Dans cette configuration, le calcul des coefficients de spline doit s'effectuer après chaque allocation. Cette modification a réduit le pic mémoire à 11.89 Go.

Dans une troisième itération, nous nous sommes concentrés sur les tableaux $2D$ qui stockent des matrices utilisées durant le calcul de l'opérateur de gyromoyenne. Ces matrices contiennent des coefficients pré-calculés pour accélérer le calcul de l'opérateur de gyromoyenne (les pré-calculs se font à chaque initialisation de GYSELA). De la même façon que pour les coefficients de spline, nous avons déplacé leur allocation/libération au plus proche de leur utilisation. De façon similaire aux coefficients de spline, le contenu de ces tableaux doit être calculé après chaque allocation. Malgré un léger surcoût en temps d'exécution, ces modifications nous permettent de réduire le pic mémoire à 11.63 Go.

Ces optimisations successives nous ont permis de réduire le pic mémoire de 17% par rapport à la version initiale. Ces réductions successives améliorent l'application GYSELA du point de vue de sa consommation mémoire et repoussent donc les limitations de l'application liées à ces aspects. La section suivante montre les détails d'une itération de la méthode IRPM dans laquelle nous avons dû porter une attention particulière aux dépendances de données.

3.2.2 Détails d'une itération de la méthode IRPM

La courbe de consommation mémoire de GYSELA donnée par la Figure 3.1 a été obtenue sur la version du code qui cumule l'ensemble des optimisations présentées dans la section précédente.

Sur cette version du code, le pic mémoire se produit dans la fonction `advec2D` qui est utilisée durant la résolution de l'équation de Vlasov. Cette équation est résolue par une méthode de *splitting* directionnel dans GYSELA (voir la section "Time-splitting" de [GSG+06]). Grossièrement, cette méthode consiste à calculer des advections (équation de transport) direction par direction. Dans la version du splitting que nous utilisons, les directions (r, θ) sont traitées ensemble car elles sont corrélées. Cette advection correspond à l'appel de la fonction `advec2D`.

Dans GYSELA, chaque processus MPI est responsable de faire évoluer la portion de la fonction de distribution qui est à sa charge, à savoir $\bar{f}(r = [i_{start}, i_{end}], \theta = [j_{start}, j_{end}], \varphi =$

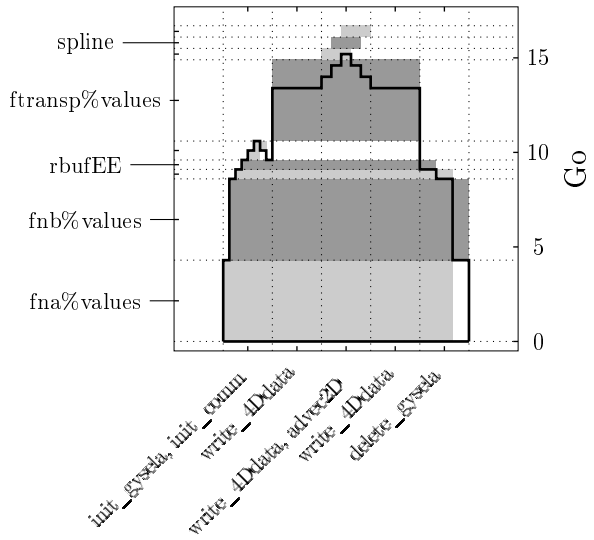


FIGURE 3.1 – Visualisation simplifiée d'une trace mémoire.

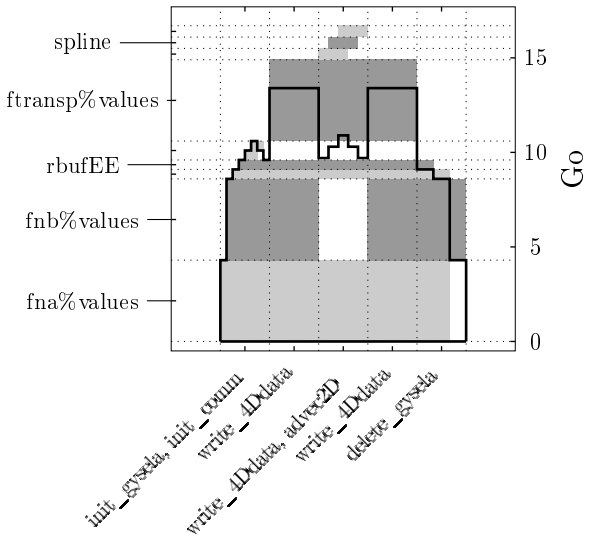


FIGURE 3.2 – Visualisation après optimisation des allocations.

$*, v_{\|} = *$[9]$, \mu = \mu_{value}$) (voir section 1.3.2). Dans l'état actuel de GYSELA, lors du calcul de l'advection dans le plan poloïdal, la décomposition de domaine de la fonction de distribution change. Une redistribution des données entre tous les processus s'effectue pour obtenir une transposition de la fonction de distribution qui est maintenant de la forme $\bar{f}(r = *, \theta = *, \varphi = [k_{start}, k_{end}], v_{\|} = [l_{start}, l_{end}], \mu = \mu_{value})$. Après transposition, chaque processus dispose localement d'un ensemble de plans poloïdaux. Au niveau du code, la variable `fnb%values` correspond à la fonction de distribution qui bénéficie de la décomposition de domaine initiale alors que la variable `ftransp%values` bénéficie de la seconde décomposition.

Après analyse, nous avons remarqué qu'au moment du pic mémoire, les variables `ftransp%values` et `fnb%values` contiennent toutes les deux la fonction de distribution, chacune correspondant à une décomposition de domaine. La Figure 3.2 illustre l'optimisation que nous avons réalisée. Une fois que la communication collective qui transfert les données de `fnb%values` à `ftransp%values` est effectuée, nous libérons la mémoire associée à la variable `fnb%values`.

Cette modification nous a permis de réduire le pic mémoire de 10% sur certaines simulations. Ce type d'optimisation réduit efficacement le pic mémoire mais détériore un peu la lisibilité du code.

3.3 Evaluation des surcoûts dus aux allocations dynamiques

Comme nous avons pu le voir dans la section précédente afin de réduire l'empreinte mémoire de l'application GYSELA, nous avons rapproché l'allocation des données de leur utilisation. Pour que cette approche soit la plus judicieuse pour les grandes configurations de simulation, nous nous sommes d'abord concentrés sur les allocations les plus pénalisantes concernant la scalabilité mémoire. Les Figures 3.3 et 3.4 illustrent les modifications que nous avons opérées dans la gestion des allocations mémoire de GYSELA. La Figure 3.3 montre que l'ensemble des tableaux utilisés par l'application est alloué dans la phase d'initialisation et libéré à la fin, alors que sur

9. La notation $*$ représente toutes les valeurs d'une dimension donnée.

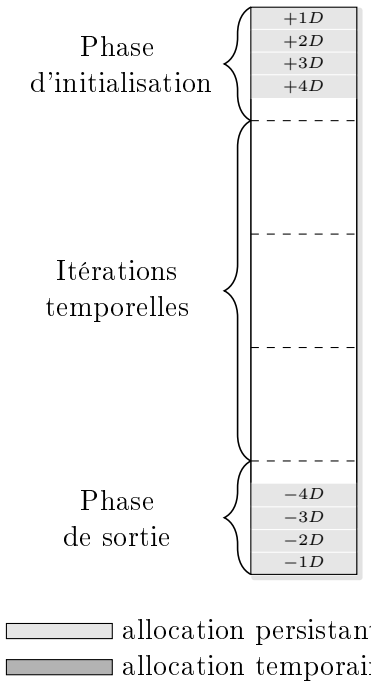


FIGURE 3.3 – Allocation persistante des structures.

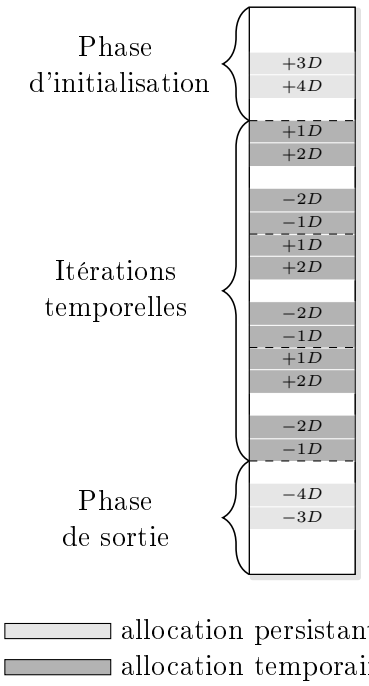


FIGURE 3.4 – Allocation temporaire de certaines structures.

la Figure 3.4 une portion importante des allocations se fait de façon dynamique durant chaque itération.

Bien que cette approche minimise l'empreinte mémoire, elle implique des appels fréquents aux primitives d'allocation (`allocate()` et `deallocate()` en FORTRAN), ce qui peut se révéler pénalisant pour les performances de l'application.

L'allocation de la mémoire sur les machines actuelles est gérée à deux niveaux : (*i*) par le système (espace noyau) et (*ii*) par des bibliothèques (espace utilisateur). Le système fournit des fonctions dites *appels système* qui permettent aux applications dans l'espace utilisateur d'exploiter les ressources matérielles via cette abstraction. Les appels du système LINUX qui permettent de demander de la mémoire sur le tas sont `mmap()` et `brk/sbrk()` [Gor04]. Les appels système ne sont généralement pas utilisés directement par une application car ces fonctions, dites de bas niveau, proposent une interface où des aspects système sont à prendre en considération. D'autre part, des appels système fréquents peuvent être pénalisants pour les performances de l'application. Le chapitre 2 de la thèse de S. Valat [Val14] décrit les différents mécanismes de gestion mémoire mis en œuvre par la système d'exploitation qui sert d'interface entre le matériel et le logiciel.

Les bibliothèques d'allocation dans l'espace utilisateur proposent deux avantages majeurs : (*i*) une interface plus simple pour manipuler les allocations que les appels système et (*ii*) leur fonctionnement interne tente de limiter les interactions entre l'espace utilisateur et le système. La bibliothèque standard *glibc* fournit les fonctions `malloc()` et `free()` avec une interface simple pour gérer les allocations dynamiques. Dans la suite de cette section, nous nous sommes intéressés à l'influence de différentes bibliothèques d'allocation sur les temps d'exécution de GYSELA.

Différentes études de performance des bibliothèques d'allocation dans l'espace utilisateur existent [LB00, FMMA11, BSKM11, EMFB14, WWWG13], mais elles s'intéressent

principalement à des applications qui font énormément d'allocations de petites tailles ce qui n'est typiquement pas le cas dans GYSELA. Nous allons comparer l'impact de différentes façons d'allouer et d'initialiser la mémoire en utilisant la bibliothèque standard dans un premier temps. Dans un second temps, des tests avec les allocateurs JEMALLOC [Eva06] et TCMALLOC [GM09] ont été réalisés.

3.3.1 Exploration de différentes stratégies d'allocation + initialisation

L'application GYSELA bénéficie d'une parallélisation MPI + OPENMP. Des tableaux temporaires sont généralement nécessaires à l'intérieur des zones OPENMP pour stocker des résultats intermédiaires. Ces tableaux sont pour la plupart alloués juste avant l'entrée de la zone OPENMP et libérés à la sortie [BGL+13]. Dans une application plus réduite de GYSELA, un prototype où des algorithmes alternatifs du solveur de Vlasov sont testés, nous avons mis en place des allocations dynamiques autour de la section de calcul du solveur de Vlasov et nous l'avons comparé à la version avec allocations persistantes (Figure 3.3 versus Figure 3.4).

Nous avons constaté dans un premier temps que le coût associé à l'opération d'allocation + initialisation représente près de 17% du temps d'exécution (1 nœud, 16 cœurs) dans la version dynamique, alors que pour la version persistante cette opération ne représente que 2% du temps d'exécution.

Afin d'identifier la cause de ce surcoût important, 8 approches d'allocation mémoire et initialisation éventuelle ont été mises en œuvre :

allocate allocation avec la fonction FORTRAN `allocate()` sans initialisation des données allouées ;

allocate + init (1) allocation avec la fonction FORTRAN `allocate()` avec initialisation des données à l'aide de boucles. Une fois le tableau alloué, on parcourt l'ensemble de ces éléments pour les mettre à zéro ;

allocate + init (2) allocation avec la fonction FORTRAN `allocate()` avec initialisation à 0. Les tableaux sont initialisés de façon globale. Par exemple l'écriture "`tab = 0.0`" initialise à `0.0` l'ensemble des éléments du tableau ;

allocate + init (3) allocation avec la fonction FORTRAN `allocate()` avec initialisation à l'aide de boucles `for` parallélisées en OPENMP. L'ensemble des cœurs disponibles est utilisé pour l'initialisation ;

malloc allocation avec à l'appel de la fonction C `malloc()` sans initialisation ;

malloc + init (1) allocation avec l'appel de la fonction C `malloc()` avec initialisation à l'aide de boucles `for` en C. Une fois le tableau alloué, on parcourt séquentiellement l'ensemble de ces éléments pour les mettre à zéro ;

malloc + init (2) allocation avec l'appel de la fonction C `malloc()` avec initialisation à l'aide de la fonction `memset()` ;

calloc allocation avec l'appel de la fonction C `calloc()` qui assure à l'utilisateur que la mémoire obtenue est mise à zéro.

Parmi ces 8 approches, on peut regrouper les 4 premières comme étant des approches purement FORTRAN, par opposition aux 4 dernières qui consistent à appeler des fonctions C à partir d'un code FORTRAN. Ceci est possible de façon portable à partir de FORTRAN 2003 grâce au module `ISO C BINDING`. On peut aussi noter que la fonction `calloc()` assure que la zone mémoire retournée est mise à 0 ; dans ce cas l'initialisation à zéro de la part de l'utilisateur n'est pas nécessaire. Il n'y a pas d'équivalent FORTRAN pour cette fonction.

	allocate	**allocate + init(1)**	**allocate + init(2)**	**allocate + init(3)**
allocation (+ init.)	0.01 0.02%	13.5 19.5%	11.1 16.0%	1.1 1.8%
calcul du solveur Vlasov	46.8 83.5%	46.5 67.1%	48.6 70.2%	46.8 81.8%
autres calculs	9.2 16.5%	9.3 13.4%	9.3 16.3%	9.6 13.8%
Total	56	69	69	57

	malloc	**malloc + init (1)**	**malloc + init (2)**	**calloc**
allocation (+ init.)	0.01 0.02%	12.4 17.7%	14.8 20.6%	0.01 0.02%
calcul du solveur Vlasov	46.5 83.3%	47.4 67.9%	47.0 65.3%	48.8 82.6%
autres calculs	9.3 16.7%	10.0 14.4%	10.1 14.0%	10.3 17.4%
Total	56	70	72	59

TABLE 3.1 – Comparaison de l'impact des différentes stratégies d'allocation + initialisation sur les temps d'exécution en secondes.

Le Tableau 3.1 nous permet de comparer les temps d'exécution de ces 8 approches. Les simulations ont été réalisées avec 1 processus MPI et 16 threads OPENMP. Ces temps ont été obtenus sur un nœud de la machine POINCARE (IDRIS, Orsay) équipée de 2 processeurs Sandy Bridge E5-2670 (2.60GHz, 8 cœurs par processeur, soit 16 cœurs par nœud). Les différentes simulations du Tableau 3.1 ont été faites sur le maillage suivant :

$$N_r = 128,\ N_\theta = 256,\ N_\varphi = 64,\ N_{v_\parallel} = 64,\ N_\mu = 1. \tag{3.2}$$

Le Tableau 3.1 présente les temps d'allocations et d'éventuelles initialisations précédant la zone OPENMP qui calcule le solveur de Vlasov, leur pourcentage par rapport au temps d'exécution total, le temps de calcul du solveur Vlasov, son pourcentage par rapport à l'exécution total, le temps d'exécution complémentaire des deux étapes précédentes, son pourcentage par rapport à l'exécution totale et le temps d'exécution total des simulations. Dans les cas sans initialisation, les résultats du solveur de Vlasov restent valides (l'initialisation est superflue ici), mais ce n'est pas le cas de tous les algorithmes. Le calcul du solveur de Vlasov est le calcul le plus coûteux de notre application ; il nécessite de plus l'utilisation de grandes zones mémoire. Pour récupérer les temps d'allocation, des compteurs ont été placés juste autour de l'appel à la fonction d'allocation et de l'initialisation lorsqu'elle est faite.

Après analyse des résultats donnés par ce tableau, on peut séparer les simulations en deux catégories, celles où l'initialisation à zéro est faite sans faire appel à `calloc()` et les autres. Dans le premier cas de figure, le temps de l'opération d'allocation et d'initialisation occupe une portion non négligeable du temps d'exécution total (jusque 16%) ; néanmoins ces temps sont moindres dans le cas **allocate + init (3)**. On peut constater aussi qu'une initialisation "en pur" FORTRAN

est aussi efficace qu'une initialisation avec la fonction `memset()` du C (le cas **allocate + init (2)**), le temps pris dépendant directement de la bande passante mémoire.

Dans les cas **allocate**, **allocate + init (3)**, **malloc** et **calloc**, on voit que les temps de simulation totaux sont nettement inférieurs aux autres cas. Le cas **calloc** est très intéressant pour nous car il présente un temps de calcul total comparable au cas **allocate** et **malloc** tout en assurant à l'utilisateur que la mémoire obtenue est initialisée à zéro. Le cas **allocate + init (3)** est aussi remarquable car il présente aussi un temps de calcul total comparable au cas **allocate** et **malloc** avec une initialisation à zéro (ou éventuellement à une autre valeur ce qui présente un réel avantage) des structures allouées dans l'espace utilisateur. Cette amélioration découle uniquement de la parallélisation en OpenMP de l'initialisation.

Les temps d'exécution des cas étudiés peuvent en partie s'expliquer du fait que les allocations se font de façon paresseuse[10] sur le système Linux. Le principe de l'allocation paresseuse signifie qu'au moment des appels aux fonctions d'allocation, de la mémoire virtuelle est allouée mais pas les pages associées en mémoire physique. La manipulation de la mémoire virtuelle est bien moins coûteuse que la manipulation des pages physiques. L'allocation de la mémoire physique est retardée au moment du premier accès. Lors du premier accès à une page virtuelle, cette dernière n'est pas encore associée à une page physique. Une exception de type *faute de page* est levée et le système traite l'événement. Dans notre étude, les temps relevés sur le Tableau 3.1 pour les cas sans initialisation ne correspondent pas au coût d'allocation des pages physiques, mais uniquement au coût d'allocation des pages virtuelles qui est négligeable par rapport au temps d'exécution total. Dans les cas sans initialisation, le coût des allocations des pages physiques est inclus dans le temps d'exécution du solveur de Vlasov qui accèdera pour la première fois aux pages virtuelles et physiques lors des calculs. À la vue du temps d'allocation dans le cas **calloc**, il s'avère que la fonction `calloc()` agit aussi de façon paresseuse. Cela implique que l'initialisation à zéro des pages physiques est retardée au moment du premier accès.

L'allocation paresseuse a aussi la bonne propriété de ne pas perturber la politique de *first-touch* [TaMS+08]. Cette politique consiste à allouer la mémoire physique sur le banc mémoire le plus proche du cœur exécutant le thread qui y accède pour la première fois. Dans les cas où l'initialisation se fait de manière explicite par l'utilisateur avec un seul thread, des effets NUMA (Non Uniform Memory Access) peuvent dégrader les performances. Dans le cadre de notre application prototype, le motif d'accès à la mémoire est tel qu'il ne permet pas de mettre en évidence ces effets.

L'utilisation de `calloc()` qui assure l'initialisation à zéro des tableaux nous semble être un bon compromis entre simplicité d'utilisation et efficacité. L'utilisation de `calloc()` est moins contraignante comparée à la mise en œuvre d'une zone OpenMP dédiée à l'initialisation de tableaux à chaque allocation, mais ne permet que les initialisations à zéro.

Bien que l'utilisation de `calloc()` assure l'obtention d'une zone mémoire initialisée à zéro, cette méthode est plus efficace que les cas où l'initialisation se fait de façon explicite par l'utilisateur. Pour expliquer l'efficacité de la fonction `calloc()`, nous nous sommes intéressés au fonctionnement interne de l'allocateur standard.

3.3.2 Aperçu du fonctionnement interne de l'allocateur standard : ptmalloc

Dans l'objectif d'améliorer les performances de l'application Gysela qui est programmée en Fortran, l'étude de l'allocateur fourni par défaut par la bibliothèque standard GNU C Library est pertinente car l'utilisation de la fonction `allocate()` est généralement redirigée

10. Pour plus de détails sur l'allocation paresseuse : `http://landley.net/writing/memory-faq.txt`

vers un appel à la fonction C `malloc()`. Actuellement l'allocateur par défaut de la GNU C LIBRARY est PTMALLOC [Glo06] qui se base sur l'allocateur DLMALLOC [LG96].

L'explication du fonctionnement interne de l'allocateur standard est largement détaillée dans [Kae01]. On peut distinguer deux types de comportements de la fonction `malloc()`. Pour les petites allocations, la mémoire est allouée avec l'appel système `sbrk()` et pour les plus grosses allocations au-delà d'un certain seuil, la mémoire est obtenue via l'appel système `mmap()`. Le seuil qui différencie les deux comportements est défini à l'aide d'un paramètre nommé `M_MMAP_THRESHOLD`. Ce paramètre est ajustable par l'utilisateur grâce à la fonction `mallopt()`. La valeur par défaut de ce paramètre est de 128 kilo octets dans PTMALLOC[11].

Les allocations de grande taille se font grâce à l'appel système `mmap()` avec l'argument `MAP_ANONYMOUS`. Cet argument indique qu'il n'y a pas de fichier à projeter sur la zone mémoire demandée. Deux comportements particuliers à l'utilisation de la fonction `mmap()` doivent être mentionnés : *(i)* avec l'argument `MAP_ANONYMOUS`, la mémoire retournée par le noyau LINUX est initialisée à zéro pour des raisons de sécurité[12] ; *(ii)* les pages physiques associées aux zones mémoires allouées via `mmap()` sont rendues au système lors d'une libération. Pour les allocations de plus petites tailles qui se font via l'appel système `sbrk()`, lors d'une libération les zones mémoire associées sont gardées dans l'espace utilisateur pour une éventuelle réutilisation et ce afin de limiter le recyclage des pages par le système et les coûts associés.

Dans toutes les simulations qui ont été réalisées, l'ensemble des tableaux dont le temps d'allocation est résumé dans le Tableau 3.1 sont de taille bien supérieure au seuil `M_MMAP_THRESHOLD`. Un appel à la fonction `malloc()` pour ces structures correspond donc un appel à `mmap()`. La mémoire retournée par le système est de ce fait déjà initialisée à zéro sur la plupart des systèmes. Néanmoins, la sémantique de la fonction `malloc()` ne garantit pas l'initialisation à zéro, contrairement à la fonction `calloc()`.

Dans notre cas, l'allocation des structures gourmandes en ressource mémoire se fait donc via l'appel à `mmap()` derrière l'appel à `malloc()`. Comme il a été dit plus haut, la partie la plus coûteuse des allocations est l'obtention des pages physiques. Notre approche pour réduire l'empreinte mémoire d'une application consiste principalement à introduire des allocations mémoire dynamiques, ce qui implique des interactions entre l'application et le système pour les grosses structures. Cette pratique augmente substantiellement le temps d'exécution. Pour pallier à cette difficulté, nous avons testé d'autres bibliothèques d'allocation. La section suivante les présente brièvement et montre les résultats que nous avons obtenus.

3.3.3 Limitation des interactions entre application et système

Afin de limiter les surcoûts, nous avons testé d'autres allocateurs que celui proposé par défaut. Dans notre étude nous nous sommes concentrés sur les allocateurs TCMALLOC et JEMALLOC. Ils se différencient par rapport à l'allocateur par défaut dans leur gestion des zones mémoire. Leur fonctionnement ne sera pas détaillé ici, mais une analyse poussée de nombreux allocateurs a été réalisée dans la thèse de S. Valat [Val14]. Pour résumer brièvement l'esprit de l'allocateur JEMALLOC, des tas locaux contenant des pages mémoire réutilisables sont associés aux threads de l'application. Cet allocateur maintient un équilibre des tailles des tas locaux entre tous les threads. Il assure aussi une consommation mémoire faible en retournant régulièrement des zones mémoire libérées au système. Dans l'approche de TCMALLOC, un tas global est partagé entre tous les threads. Les pages mémoire de ce tas sont retournées au système uniquement si nécessaire,

11. Il est possible de vérifier cette valeur grâce à la fonction `malloc_get_state()`.
12. Pour plus de renseignement sur la sécurité`http://stackoverflow.com/questions/2688466/why-mallocmemset-is-slower-than-calloc`

	allocate		calloc	
	TCMALLOC	JEMALLOC	TCMALLOC	JEMALLOC
allocation (+ init.)	0.001 0.0%	0.0002 0.0%	8.7 13.2%	0.0003 0.0%
calcul du solveur Vlasov	44.2 82.3%	45.9 82.3%	46.9 71.1%	46.4 80.0%
autres calculs	9.5 17.7%	9.9 17.7%	10.3 15.6%	11.6 20.0%
Total	53	56	66	58

TABLE 3.2 – Comparaison de l'impact des allocateurs TCMALLOC et JEMALLOC sur le temps d'exécution en secondes.

sinon elles sont conservées. Bien que les approches de ces deux allocateurs soient différentes, elles ont toutes les deux pour objectif de limiter les interactions fréquentes avec le système.

L'utilisation des bibliothèques TCMALLOC et JEMALLOC est aisée sur les systèmes UNIX. Il suffit de renseigner la variable d'environnement `LD_PRELOAD` avec le chemin de la bibliothèque qu'on souhaite utiliser avant de lancer une application. Les fonctions disponibles dans la bibliothèque renseignée par `LD_PRELOAD` seront appelées en priorité à l'exécution. Dans notre cas, cela nous permet de court-circuiter les fonctions d'allocation de la GNU C LIBRARY.

Nous avons identifié le cas **calloc** comme étant le plus attrayant pour allouer et initialiser de la mémoire. Nous avons testé les allocateurs TCMALLOC et JEMALLOC sur les cas **allocate** et **calloc** du Tableau 3.1 afin de pouvoir comparer les différents temps d'exécution globaux avec et sans initialisation.

Les temps d'exécution exposés dans le Tableau 3.2 ont été obtenus dans les mêmes conditions que pour les simulations du Tableau 3.1. On peut voir une différence notable sur le temps d'allocation entre les cas **allocate** et **calloc** avec l'utilisation de TCMALLOC. Dans le premier cas, le temps d'allocation est négligeable alors qu'il ne l'est pas avec l'utilisation de `calloc()`. En comparant avec le Tableau 3.1, on peut conclure que l'implémentation de `calloc()` de la bibliothèque TCMALLOC n'est pas paresseuse. Cela peut s'expliquer par le fait que les pages mémoire obtenues par TCMALLOC sont gardées dans l'espace utilisateur. Pour assurer que la sémantique de la fonction `calloc()` soit respectée, l'initialisation à zéro se fait certainement de façon séquentielle au niveau de l'appel dans l'espace utilisateur. Pour limiter au maximum les interactions avec le système d'exploitation, TCMALLOC rend des pages mémoire au système uniquement lorsque c'est nécessaire. De fait, cette bibliothèque ne peut pas bénéficier de la remise à zéro liée à l'appel système `mmap()`.

Ces bibliothèques qui limitent les interactions avec le système offrent de faibles gains de performance sur notre application cible dans certaines configurations d'utilisation.

3.3.4 Mise en œuvre d'une initialisation performante dans GYSELA

Les résultats que montre le Tableau 3.1 encouragent l'utilisation de la primitive d'allocation `calloc()` au sein de l'application GYSELA. Durant le processus de réduction de l'empreinte mémoire, les allocations ont été déplacées et nous avons évité, autant que possible, d'initialiser les structures lorsque cela n'était pas nécessaire. Nous avons identifié des situations où l'initialisation est requise. Par exemple dans le cas où des sommes partielles sont cumulées dans un tableau

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 3.3 – Strong scaling : taille des allocations en Go (par processus MPI) et pourcentage par rapport au pic mémoire de chaque catégorie de structure de données.

initialisé à zéro. D'autre part, nous initialisons aussi les tableaux pour des raisons de débogage. Pour cela, nous utilisons la valeur *sNaN* (signal Not a Number) pour initialiser les tableaux. Lors de l'exécution si cette valeur est utilisée, une exception est levée et interrompt l'exécution du programme. Cela nous permet de repérer rapidement des bugs difficiles à détecter autrement (utilisation de zones mémoire non initialisées). Ces initialisations ne peuvent pas être effectuées avec un simple appel à `calloc()`.

Afin de limiter l'impact des initialisations à zéro, nous avons utilisé la fonction `calloc()` pour tous les tableaux nécessitant une initialisation. Pour se faire, nous avons développé cette fonctionnalité dans une version dédiée de la LIBMTM. Cette version utilise le module `ISO C BINDING` pour appeler la fonction `calloc()`. Dans la version de production de GYSELA, nous n'observons pas de gain significatif de performance avec cette version de la LIBMTM. Comme il est dit plus haut, durant la mise en place des allocations dynamiques, nous avons évité autant que possible l'initialisation à zéro des structures, ce qui limite l'impact des surcoûts dus aux initialisations. Face à ce constat, nous n'avons pas intégré l'utilisation du module `ISO C BINDING` dans la version de production de GYSELA.

3.4 Etudes réalisables grâce à l'outil de prédiction

La fonctionnalité de prédiction de la LIBMTM nous donne la possibilité de réaliser des études difficilement accessibles sans elle. Dans cette partie, nous présentons deux exemples.

3.4.1 Etude de la scalabilité mémoire

La première application de l'outil de prédiction que nous proposons est l'étude de la scalabilité mémoire de GYSELA. Pour ce faire, nous reprendrons les paramètres de simulation utilisés pour produire le Tableau 1.1 (p. 15). Le maillage utilisé dans cette étude était le suivant :

$$N_r = 1024, N_\theta = 4096, N_\varphi = 1024, N_{v_\parallel} = 128, N_\mu = 2.$$

À partir de la version optimisée du point de vue mémoire de GYSELA, nous avons reproduit les configurations utilisées lors de l'étude de scalabilité de la section 1.3.3.

Le Tableau 3.3 présente les différents pics mémoire que nous avons obtenus. Contrairement au Tableau 1.1 où les chiffres ont été obtenus grâce à des exécutions effectives du programme, les chiffres du Tableau 3.3 ont été obtenus grâce à l'outil de prédiction. Comme on peut le voir en comparant ces tableaux, sur le plus gros cas (32k cœurs), la consommation des structures 2D a été réduite à 20.8%. Aussi, la consommation mémoire sur ce cas a été réduite de 50.8% par rapport à la version originale (voir Tableau 1.1). Les structures $4D$ contiennent les données physiques les plus utilisées durant une simulation et elles consomment la plus grande partie de la mémoire comme on peut s'y attendre. Malgré le fait que les chiffres du Tableau 3.3 aient été obtenus uniquement grâce à l'outil de prédiction, nous avons vérifié en pratique que la configuration du cas à 32k cœurs pouvait effectivement s'exécuter sur la machine Helios. Aussi nous avons pu vérifier que la prédiction du pic mémoire correspondait exactement au pic mémoire atteint par la simulation réelle.

La comparaison des Tableaux 1.1 et 3.3 nous permet de faire un bilan sur la consommation mémoire et surtout sur les bienfaits des améliorations apportées à l'application sur des grandes configurations. Dans notre cas, la gestion fine des allocations/libérations des structures pénalisantes nous a permis de repousser de façon significative les limites de l'application d'un point de vue consommation mémoire.

3.4.2 Choix de paramètres pour les simulations de production

Avant de soumettre une simulation sur un supercalculateur, il est souhaitable de savoir si les besoins mémoire de cette simulation sont réalistes par rapport à la quantité de mémoire disponible. Cette information permet aux utilisateurs d'éviter de soumettre des simulations qui "planteront" à cause d'une surconsommation mémoire. Ceci est particulièrement utile lors de la soumission de grandes simulations car le temps d'attente avant son exécution peut être de l'ordre de la semaine et le nombre d'heures de calcul perdues est proportionnel au nombre de cœurs de calcul utilisés.

La seconde application de l'outil de prédiction que nous présentons ici est une aide dans le choix des paramètres de déploiement MPI/OpenMP d'une simulation. Pour illustrer ce point, nous avons réalisé une simulation de référence sur la machine Juqueen qui dispose d'un grand nombre de nœuds de calcul. Juqueen est une machine de type Blue Gene/Q avec 16 Go de mémoire et 16 cœurs de calcul par nœud. De plus, pour utiliser au mieux la puissance de calcul de cette machine, il est recommandé d'utiliser 64 threads par nœud (soit 4 threads par cœur).

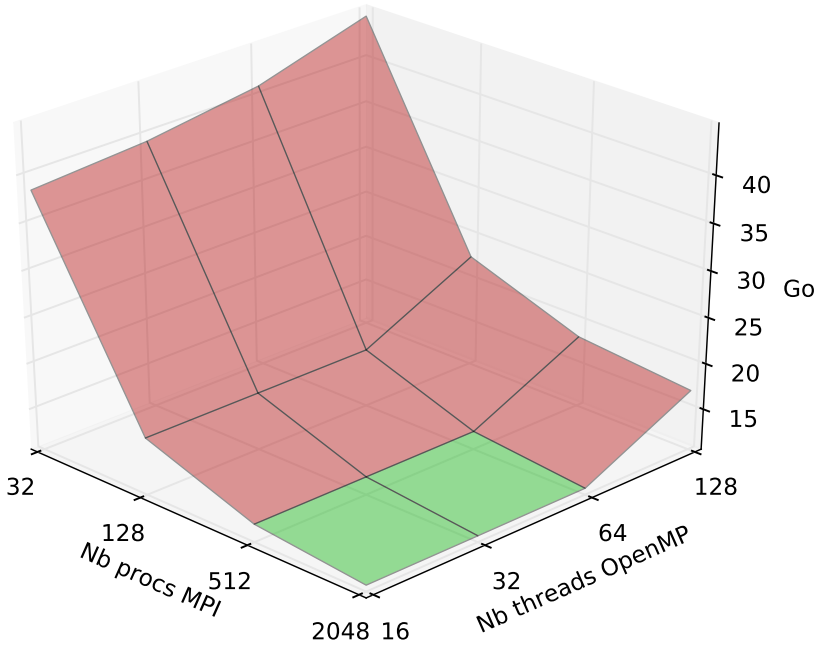
La simulation de référence que nous avons exécutée utilise 256 processus MPI et 64 threads OpenMP, soit 4k cœurs. De façon usuelle pour les simulations de Gysela, un processus MPI utilise un nœud de calcul. Nous avons conservé ce déploiement sur la simulation de référence ; le nombre de processus MPI correspond donc au nombre de nœuds de calcul utilisés. Le maillage utilisé est le suivant :

$$N_r = 256,\ N_\theta = 256,\ N_\varphi = 128,\ N_{v_\parallel} = 64,\ N_\mu = 4\,. \tag{3.3}$$

Grâce aux traces `MTM` générées par le cas de référence, nous pouvons prévoir le comportement mémoire de Gysela pour différents déploiements. Nous souhaitons lister l'ensemble des configurations qui permettent l'exécution de Gysela sur le maillage suivant :

$$\mathbf{N_r = 1024},\ \mathbf{N_\theta = 1024},\ N_\varphi = 128,\ N_{v_\parallel} = 64,\ N_\mu = 4\,. \tag{3.4}$$

En mettant les valeurs du maillage cible dans une trace et en scannant différentes configurations de déploiement, l'outil de prédiction de la libMTM (voir section 2.3.3) nous



		# threads OpenMP			
		16	32	64	128
# processus MPI	32	38.47	39.36	41.12	44.64
	128	16.78	16.79	16.80	22.60
	512	*12.51*	*12.52*	*12.53*	18.26
	2048	*11.24*	*11.24*	*11.26*	17.07

FIGURE 3.5 – Pic mémoire en Go en fonction du nombre de processus MPI et de threads OpenMP.

permet de connaître le pic mémoire en fonction du nombre de processus et de threads utilisés. Les résultats obtenus sont affichés par la Figure 3.5. Avec 16 Go de mémoire disponible par nœud, on peut voir que les configurations avec au moins 512 processus et au plus 64 threads tiennent dans l'espace mémoire.

Certaines structures de données de GYSELA bénéficient de la décomposition de domaine, ce qui signifie que leur coût mémoire diminue lorsque le nombre de processus MPI utilisés augmente. En suivant la consommation mémoire le long de l'axe du nombre de processus sur la Figure 3.5, on peut constater ce comportement. Plus on utilise de ressources matérielles, plus le pic mémoire par processus diminue. Néanmoins plus grand est le nombre de processus MPI, plus cet effet s'atténue. Cette observation signifie que les structures qui ne bénéficient pas de la décomposition de domaine représentent une portion de plus en plus grande du pic mémoire. Typiquement, dans cette étude où on double le nombre de processus MPI utilisés selon les directions r et θ, les structures de données qui ne sont pas distribuées dans ces deux directions sont pénalisantes.

L'influence du nombre de threads sur le pic mémoire est moins marquée. En regardant la Figure 3.5 le long de l'axe du nombre de threads avec 32 processus, on constate une augmentation graduelle du pic mémoire. Dans les autres configurations, pour un nombre de processus plus

grand, le nombre de threads a une influence négligeable sur le pic mémoire sauf dans le cas avec 128 threads. Indépendamment du nombre de processus MPI, le pic mémoire augmente entre une configuration avec 64 threads et celle avec 128 threads. Après analyse des courbes chronologiques de consommation mémoire des deux configurations, on se rend compte que le pic mémoire n'est pas localisé au même endroit. Dans la configuration avec 128 threads, le pic mémoire se situe dans une zone du code où la taille des structures dépend fortement du nombre de threads, alors que ce n'est pas le cas du pic mémoire dans la configuration à 64 threads.

Cette étude illustre le fait qu'en fonction de la taille du maillage, du nombre de processus MPI et de threads OpenMP, le pic mémoire peut se localiser dans différentes zones du code. Ce comportement s'explique en regardant la dépendance entre la taille de certains tableaux caractéristiques et la valeur de certains paramètres d'entrée. Par exemple, la taille des buffers MPI est sensible aux paramètres de parallélisation. Dans Gysela, les tailles des buffers temporaires sont sensibles au nombre de points dans les directions r et θ.

Ce type de comportement peut être étudié grâce à l'outil de prédiction. Il reproduit fidèlement les allocations/libérations à l'extérieur de l'application instrumentée (voir section 2.3.3). Les déplacements du pic mémoire que permet d'étudier la libMTM n'est pas accessible par une approche empirique et moins systématique que celle qui est utilisée pour l'application Nest (voir section 2.1.2).

3.4.3 Surveillance de la consommation mémoire (en perspective)

En perspective, dans le cadre des études possibles grâce à l'outil de prédiction, nous envisageons de suivre la consommation mémoire de la version de production de Gysela (voir section 1.3.4). Dans notre processus de développement, nous utilisons l'intégration continue. À chaque modification de la version de production de l'application, une série de tests est exécutée. Le test que nous souhaitons mettre en place à terme permettrait de suivre l'évolution de la consommation de Gysela sur une grande configuration grâce à l'outil de prédiction. À partir d'un petit cas de référence, une prédiction sur les traces `MTM` générées permettrait d'estimer le pic mémoire sur des grandes configurations. Ce test ne demanderait pas l'utilisation d'un grand nombre de ressources tout en fournissant des informations pertinentes sur le comportement de l'application sur des configurations de grande taille. L'idéal serait de maintenir alors un historique de l'évolution de ce pic mémoire.

Conclusion

Ce chapitre illustre des usages de la libMTM. Les outils qui sont mis à disposition par la bibliothèque permettent d'étudier les aspects mémoire d'une application parallèle, de guider les améliorations et de réaliser des études difficilement accessibles autrement. Ils constituent un réel support dans le cycle de développement d'une application et représentent un réel outil pratique pour assister le passage à l'échelle d'une application scientifique parallèle.

Dans les développements futurs de la libMTM, son utilisation dans différentes applications serait bénéfique. Cela permettrait d'avoir des retours d'expérience qui contribueraient à la robustesse de la bibliothèque. Pour faciliter son utilisation, on peut imaginer le développement d'une instrumentation automatique des sources. Cette tâche est plutôt ambitieuse, mais elle peut faciliter grandement le premier contact avec la bibliothèque.

Une extension possible est l'amélioration de l'outil de visualisation. Lorsqu'un grand volume de données doit être affiché, l'outil de visualisation actuel manque de réactivité. La libMTM génère actuelle les graphiques en Python grâce à l'extension matplotlib. En mode interactif,

la réactivité du script de post-traitement est peu véloce. De plus, beaucoup d'informations de la trace sont actuellement inutilisées (affichage d'une seule trace à la fois, pas d'accès au nom du fichier source responsable d'une allocation, pas d'accès au numéro de ligne de l'allocation, etc...). Le développement d'un outil de visualisation plus sophistiqué serait un atout pour l'utilisation de la bibliothèque.

Il serait aussi intéressant de coupler la LIBMTM avec une autre bibliothèque [13] qui intercepte l'ensemble des appels aux primitives de gestion mémoire. Cela permettrait de comptabiliser la consommation mémoire globale associée au tas d'une application. En particulier, cela permettrait de capturer les allocations qui sont faites par des bibliothèques tierces utilisées par l'application. L'enregistrement de l'ensemble de ces informations risque d'augmenter fortement la taille de la trace générée. Il sera éventuellement nécessaire à ce moment là d'utiliser des méthodes de compression.

13. Bibliothèque d'interception de primitives de gestion mémoire : https://github.com/qgears/log-malloc-simple

Deuxième partie

Étude et optimisation de l'opérateur gyromoyenne

Chapitre 4

Calcul de l'opérateur de gyromoyenne

Sommaire

Cette seconde partie du manuscrit est dédiée au calcul de l'opérateur de gyromoyenne sur un maillage polaire. Le travail réalisé dans le cadre de cette thèse à contribuer au développement, à l'optimisation, à l'intégration et à l'évaluation d'une méthode de calcul pour l'opérateur de gyromoyenne alternative par rapport à la méthode existante par approximation de Padé qui est utilisée par défaut dans GYSELA. La méthode de calcul alternative est basée sur une intégration directe dans l'espace réel. Elle est proche de la méthode déjà mise en œuvre dans le code GENE [GLB+11, Gör09]. Les interpolations qui constituent le cœur des calculs se font dans un plan $2D$. Les travaux détaillés dans ce chapitre et dans le chapitre suivant font l'objet des articles [SMC+15, RSL+15].

Ce chapitre décrit en détail le calcul de la gyromoyenne par approximation de Padé (voir section 4.2) et par interpolation d'Hermite (voir section 4.3). Avant d'aborder les aspects techniques, la section suivante définie l'opérateur de gyromoyenne.

4.1 Définition de l'opérateur de gyromoyenne

Les particules se déplacent de façon hélicoïdale autour des trajectoires des centres-guides [Lit88]. Soit $\vec{x}_G$ la position du centre-guide, $\vec{x}$ la position de la particule et $\vec{\rho}$ le rayon de giration, usuellement appelé rayon de *Larmor* (voir section 1.3.1, p. 10). Ces quantités sont reliées par la relation suivante :

$$\vec{x} = \vec{x}_G + \vec{\rho}.$$

L'opérateur de gyromoyenne consiste à moyenner une grandeur sur le cercle de rayon $\vec{\rho}$ et de centre $\vec{x}_G$. Le vecteur $\vec{\rho}$ est contenu dans le plan orthogonal au vecteur unitaire $\vec{b} = \vec{B}/||\vec{B}||$ et il dépend de la *gyrophase* $\alpha \in [0, 2\pi]$. Il s'exprime ainsi :

$$\vec{\rho} = \rho(\cos(\alpha)\vec{e}_{\perp 1} + \sin(\alpha)\vec{e}_{\perp 2})$$

avec $(\vec{e}_{\perp 1}, \vec{e}_{\perp 2})$ une base locale du plan orthogonal à la direction du champ magnétique $\vec{b}$. En réalité, dans le cas d'un champ magnétique au sein d'un tokamak, les lignes de champ ne sont pas parfaitement orthogonales au plan poloïdal. Néanmoins, il est d'usage de considérer que le rayon de giration est contenu dans un plan poloïdal. L'article [FBDR+98] justifie cette approximation par le fait que l'erreur commise est de l'ordre de $(B_\perp/B_\parallel)^2$ avec $B_\parallel$ et $B_\perp$ respectivement la composante parallèle et perpendiculaire du champ magnétique dans un repère d'un plan poloïdal. Dans le contexte des tokamaks, la valeur de l'erreur commise est négligeable. Nous considérons donc dans la suite de ce manuscrit que le cercle de giration des particules appartient au plan poloïdal.

Soit $f : (r, \theta) \in \mathbb{R}^+ \times \mathbb{R} \mapsto \mathbb{R}$ une fonction définie dans le système de coordonnées polaires du plan poloïdal et soit l'opérateur de gyromoyenne $\mathcal{J}_\rho$ qui désigne l'intégrale le long du cercle de Larmor d'un point $\vec{x}_G$, pour une valeur de φ donnée. On définit la gyromoyenne usuellement comme l'intégrale suivante :

$$\mathcal{J}_\rho(f)(\vec{x}_G) = \frac{1}{2\pi}\int_0^{2\pi} f(\vec{x}_G + \vec{\rho})d\alpha \tag{4.1}$$

où $\vec{x}_G = (r, \theta)$ et $\vec{\rho} = (\rho, \alpha)$ en coordonnées polaires. La démonstration de la section 5.1 donne les étapes de calcul qui permettent de passer de l'intégrale le long d'une courbe paramétrée (le cercle de Larmor) à l'intégrale donnée par l'équation (4.1).

L'opérateur de gyromoyenne a pour effet de lisser les fluctuations de la fonction d'entrée f qui sont à des échelles spatiales inférieures au rayon de Larmor.

Ce chapitre présente 2 méthodes de calcul numérique pour l'opérateur de gyromoyenne. La section 4.2 présente la méthode existante utilisée actuellement par défaut dans l'application Gysela qui est basée sur une approximation de Padé. La section 4.3 présente l'opérateur de gyromoyenne basé sur l'interpolation d'Hermite. Une description formelle de type produit matrice–vecteur de cet opérateur est donnée en section 4.4. Les différentes optimisations qui ont été réalisées sur cette méthode sont exposées en section 4.5. Les comparaisons et discussions entre la gyromoyenne basée sur une approximation de Padé et la gyromoyenne basée sur l'interpolation d'Hermite font l'objet du chapitre 5.

4.2 Gyromoyenne basée sur une approximation de Padé

L'article [SGF+05] traite du calcul de la gyromoyenne basée sur une approximation de Padé et de ses impacts. Cette méthode s'appuie sur une réécriture de l'opérateur de gyromoyenne dans

l'espace de Fourier. Cette approche est motivée par le fait qu'une translation dans l'espace réel (ici de $\vec{\rho}$) se traduit par un facteur multiplicateur dans l'espace de Fourier. Cette section détaille les étapes de calcul qui guident la réécriture de l'opérateur de gyromoyenne.

4.2.1 Gyromoyenne exprimée dans l'espace de Fourier

À partir de l'équation (4.1), considérons la transformée de Fourier spatiale [Kna05] de la fonction f :

$$\begin{aligned} f(\vec{x}_G + \vec{\rho}) &= \int_{\mathbb{R}^3} \widehat{f}(\vec{k}) e^{i\vec{k}\cdot(\vec{x}_G+\vec{\rho})} d\vec{k} \\ &= \int_{\mathbb{R}^3} \widehat{f}(\vec{k}) e^{i\vec{k}\cdot\vec{x}_G} e^{i\vec{k}\cdot\vec{\rho}} d\vec{k} \end{aligned} \tag{4.2}$$

avec $\vec{k} = (k_\perp cos(\phi), k_\perp sin(\phi), k_\parallel)$ le vecteur d'onde. En développant le second produit scalaire de l'équation (4.2), on a :

$$\begin{aligned} \vec{k}\cdot\vec{\rho} &= \begin{bmatrix} k_\perp cos(\phi) \\ k_\perp sin(\phi) \\ k_\parallel \end{bmatrix} \cdot \rho \begin{bmatrix} cos(\alpha) \\ sin(\alpha) \\ 0 \end{bmatrix} \\ &= k_\perp \rho (cos(\phi)cos(\alpha) + sin(\phi)sin(\alpha)) \\ &= k_\perp \rho cos(\alpha - \phi). \end{aligned} \tag{4.3}$$

En utilisant (4.3) dans (4.1) :

$$\begin{aligned} \mathcal{J}_\rho(f)(\vec{x}_G) &= \frac{1}{2\pi} \int_0^{2\pi} \int_{\mathbb{R}^3} \widehat{f}(\vec{k}) e^{i\vec{k}\cdot\vec{x}_G} e^{ik_\perp \rho cos(\alpha-\phi)} d\vec{k} d\alpha \\ &= \int_{\mathbb{R}^3} \widehat{f}(\vec{k}) e^{i\vec{k}\cdot\vec{x}_G} \Big[\frac{1}{2\pi} \int_0^{2\pi} e^{ik_\perp \rho cos(\alpha-\phi)} d\alpha \Big] d\vec{k}. \end{aligned}$$

Dans l'intégrale la plus interne de l'expression précédente, on peut considérer la variable ϕ comme un paramètre de l'expression. De plus, la fonction $x \to e^{ik_\perp \rho cos(x)}$ est 2π–périodique. Le calcul de l'intégrale étant réalisé sur une période de cette fonction, la valeur du paramètre ϕ n'impacte pas la valeur de l'intégrale. On peut donc se ramener au calcul suivant :

$$\mathcal{J}_\rho(f)(\vec{x}_G) = \int_{\mathbb{R}^3} \widehat{f}(\vec{k}) e^{i\vec{k}\cdot\vec{x}_G} \Big[\frac{1}{2\pi} \int_0^{2\pi} e^{ik_\perp \rho cos(\alpha)} d\alpha \Big] d\vec{k}.$$

Dans l'expression précédente, on peut identifier la fonction de Bessel [Wei, Kre12] de première espèce à l'ordre 0. Pour un ordre $n \in \mathbb{N}$ quelconque, $\forall z \in \mathbb{C}$, la fonction peut s'écrire de la façon suivante :

$$J_n(z) = \frac{1}{2\pi i^n} \int_0^{2\pi} e^{izcos(\alpha)} e^{in\alpha} d\alpha$$

d'où :

$$J_0(z) = \frac{1}{2\pi} \int_0^{2\pi} e^{izcos(\alpha)} d\alpha.$$

On obtient donc :

$$\mathcal{J}_\rho(f)(\vec{x}_G) = \int_{\mathbb{R}^3} \widehat{f}(\vec{k}) e^{i\vec{k}\cdot\vec{x}_G} J_0(k_\perp \rho) d\vec{k}. \tag{4.4}$$

Considérons maintenant la transformée de Fourier de l'opérateur de gyromoyenne :

$$\mathcal{J}_\rho(f)(\vec{x}_G) = \int_{\mathbb{R}^3} \widehat{\mathcal{J}_\rho(f)}(\vec{k}) e^{i\vec{k}\cdot\vec{x}_G} d\vec{k}. \tag{4.5}$$

Par identification des coefficients de Fourier entre (4.4) et (4.5), on obtient la relation suivante :

$$\widehat{\mathcal{J}_\rho(f)}(\vec{k}) = J_0(k_\perp \rho)\widehat{f}(\vec{k}). \tag{4.6}$$

Cette dernière relation montre que l'opérateur de gyromoyenne correspond dans l'espace de Fourier à multiplier les amplitudes des différents modes de la fonction d'entrée f par la fonction de Bessel J_0. La Figure 4.1 donne l'allure de la fonction J_0 sur l'intervalle $[0\,,17]$. On peut voir que les images de cette fonction sont contenues dans l'intervalle $[-0.5\,,1]$ et ont un comportement d'oscillations amorties autour de 0. La relation 4.6 montre donc que l'évaluation de la fonction de Bessel en $k_\perp\rho$ nous permettrait de calculer les coefficients de Fourier de la gyromoyenne de f à partir de sa transformée de Fourier. Néanmoins, cette relation qui fait intervenir des coefficients de l'espace de Fourier ne permet pas de revenir simplement à l'espace des positions surtout dans le cas où certaines directions ne sont pas périodiques.

Afin de contourner cette difficulté, il est commun d'utiliser une approximation de la fonction de Bessel. Dans [CMS10] les estimations de la fonction de Bessel par développements limités et par approximation de Padé à différents ordres sont comparées. La section suivante présente l'approximation de Padé à l'ordre 1 que nous utilisons pour estimer la fonction de Bessel.

4.2.2 L'approximation de Padé de la fonction de Bessel

L'approximation de Padé est une méthode d'approximation d'une fonction analytique par une fonction rationnelle. En ce sens, elle est un peu analogue à un développement limité de Taylor qui donne une approximation d'une fonction selon les mêmes critères à l'aide de polynômes. L'approximation de Padé à l'ordre 1 de la fonction de Bessel d'ordre 0 est :

$$J_0(k_\perp\rho) \sim \frac{1}{1+(k_\perp\rho)^2/4}.$$

On peut voir sur la Figure 4.1 les courbes de la fonction de Bessel et de son approximation de Padé à l'ordre 1. Cette approximation est valide pour de petites valeurs de $k_\perp\rho$ et elle a le même comportement asymptotique quand $k_\perp\rho \to \infty$ que la fonction de Bessel initiale.

En utilisant cette approximation dans l'équation (4.6), on obtient :

$$(1+(k_\perp\rho)^2/4)\widehat{\mathcal{J}_\rho(f)}(\vec{k}) \sim \widehat{f}(\vec{k}).$$

Cette relation nous permet maintenant de revenir à l'espace des positions. En utilisant l'équivalence $ik_\perp \leftrightarrow \nabla_\perp$, on obtient l'équation différentielle suivante :

$$(1-\frac{\rho^2}{4}\nabla_\perp^2)\mathcal{J}_\rho(f)(\vec{x}_G) \sim f(\vec{x}_G). \tag{4.7}$$

Dans le système de coordonnées polaires, le Laplacien a l'expression suivante :

$$\nabla_\perp^2 = \Delta_\perp = \frac{\partial^2}{\partial r^2} + \frac{1}{r}\frac{\partial}{\partial r} + \frac{1}{r^2}\frac{\partial}{\partial\theta^2}.$$

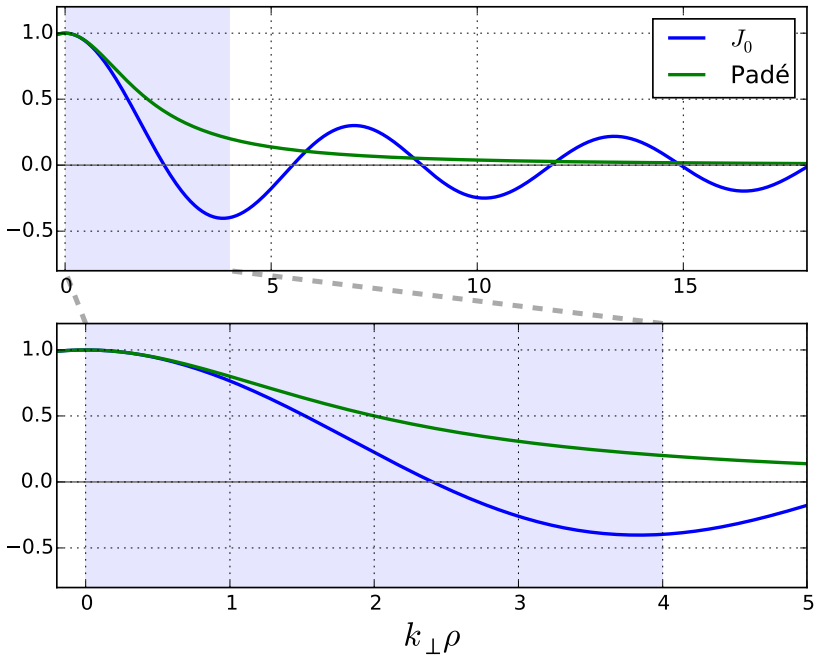


FIGURE 4.1 – Comparaison entre la fonction de Bessel J_0 et son approximation de Padé.

4.2.3 Calcul de gyromoyenne basé sur l'approximation de Padé

Considérons un maillage polaire uniforme $(r, \theta) \in [r_{\min}, r_{\max}] \times [0, 2\pi[$ avec $N_r \times N_\theta$ cellules ; notre but est de calculer la gyromoyenne en chaque point du maillage grâce à l'approximation de Padé :

$$(f_{j,k}) \in \mathbb{R}^{(N_r+1)\times N_\theta} \mapsto (\mathcal{J}_\rho(f)_{j,k}) \in \mathbb{R}^{(N_r+1)\times N_\theta}.$$

$N_r + 1$ points sont nécessaires pour définir N_r cellules dans la direction radiale r et N_θ points dans la direction poloïdale θ.

La direction θ étant périodique, en projetant f et $\mathcal{J}_\rho(f)$ dans une base de Fourier dans cette direction, on obtient :

$$f(r,\theta) \approx \sum_{n=0}^{N_\theta-1} A_n(r) e^{in\theta}, \quad \mathcal{J}_\rho(f)(r,\theta) \approx \sum_{n=0}^{N_\theta-1} B_n(r) e^{in\theta}.$$

À r fixé, les coefficients $A_n(r)$ et $B_n(r)$ correspondent respectivement aux coefficients de Fourier de f et de sa gyromoyenne dans la direction θ.

À partir de l'équation (4.7) et des relations précédentes, on obtient la relation suivante :

$$A_n(r) = -\frac{\rho^2}{4} B_n''(r) - \frac{\rho^2}{4r} B_n'(r) + (1 + \frac{\rho^2 n^2}{4r^2}) B_n(r). \tag{4.8}$$

Cette dernière relation, qui est une équation implicite pour l'opérateur de gyromoyenne, permet de construire un système linéaire où les coefficients $B_n(r)$ se calculent par différences finies.

4.2.4 Contraintes – limitations de l'approximation de Padé

Cette approche pour calculer l'opérateur de gyromoyenne est directe dans le cas d'un maillage régulier ; c'est actuellement le cas de l'application GYSELA qui utilise un maillage polaire dans le plan poloïdal. Néanmoins dans le cas des maillages non structurés, l'exploitation de l'équation (4.8) peut ne pas être possible.

Dans cette méthode de calcul, on fait l'hypothèse implicite que le rayon de giration ρ est constant dans tout le plan poloïdal. Le cas où le rayon de giration admet une dépendance spatiale n'est pas prise en compte.

Un autre facteur limitant de cette approche est l'utilisation d'une transformée de Fourier. Son calcul implique de connaître la fonction dont on veut calculer la transformée sur tout les points dans une direction périodique. Dans le contexte du calcul haute performance, les applications sont souvent parallélisées grâce à une décomposition de domaine. Dans ce contexte, l'utilisation d'une transformation de Fourier impose une réelle contrainte, car elle nécessite que l'ensemble des données du plan poloïdal soit local à un processus pour effectuer le calcul de gyromoyenne.

Le manque de flexibilité de cette méthode de calcul représente un handicap pour l'évolution de l'application GYSELA. Pour pallier à ces limitations, des méthodes de calcul de gyromoyenne basées sur des interpolations dans l'espace réel ont été développées.

4.3 Gyromoyenne basée sur l'interpolation d'Hermite

Depuis la publication de l'article [Lee87], l'évaluation de la gyromoyenne dans l'espace réel a été étudiée. La méthode de calcul se base sur une intégration directe et sur un opérateur d'interpolation. Pour une fonction f définie sur un plan en entrée, en un point $\vec{x}_G$ donné du plan, l'intégration directe consiste à interpoler f en $N \in \mathbb{N}^*$ positions sur le cercle de Larmor associé à $\vec{x}_G$ et à calculer la moyenne des valeurs obtenues. On peut l'interpréter simplement comme la formulation discrète de l'équation (4.1) :

$$\mathcal{J}_\rho(f)(\vec{x}_G) = \frac{1}{2\pi} \sum_{k=0}^{N-1} \Delta\alpha f(\vec{x}_G + \vec{\rho_k}) \tag{4.9}$$

avec $\vec{\rho_k} = (\rho, \alpha_k)$, $\alpha_k = \theta_j + k\Delta\alpha$ et $\Delta\alpha = 2\pi/N$. Le calcul numérique de cette intégrale est ici réalisé grâce à la méthode des rectangles. La Figure 4.2 donne une représentation graphique de ce calcul.

L'étude dans [Lee87] établit que 4 points équidistants sur le cercle peuvent être suffisants en terme de précision. Cette approximation s'est démocratisée dans d'autres travaux [LL95, FBDR+98]. Ces points d'interpolation sont appelés points d'échantillonnage ou points de quadrature dans la suite.

Plus récemment dans [HTK+02], le nombre de points d'interpolation sur le cercle varie linéairement avec le rayon de giration ρ entre 4 et 32 afin d'augmenter la précision du calcul. L'objectif de cette relation est de conserver autant que possible une même longueur d'arc entre deux points d'interpolation quelque soit la taille du cercle d'intégration. L'interpolation est réalisée grâce à des B-splines. La même approche est prise dans [ITK03, IIK+08], mais l'interpolation est cette fois effectuée grâce à des splines quadratiques. Malgré le fait que 4 points suffisent à capter la physique due aux grandes longueurs d'onde ($k_\perp \rho < 1$), l'utilisation d'un plus grand nombre de points d'interpolation permet de retrouver le comportement d'un filtre de Bessel [JBA+07]. Dans les travaux [JBA+07, HKM07] le nombre de points d'échantillonnage du cercle est linéaire avec la vitesse perpendiculaire $v_\perp$ à la position où la gyromoyenne s'applique.

De plus, les positions des points de quadrature réparties de façon équidistante sont pivotées d'un angle aléatoire recalculé à chaque pas de temps.

Le choix de la méthode d'interpolation pour le calcul de gyromoyenne a un impact à la fois en terme de précision et en terme de temps d'exécution. En effet, les points de quadrature ne coïncidant pas avec les nœuds du maillage sous-jacent, la précision de la gyromoyenne dépend de la précision de la méthode d'interpolation. Différentes méthodes d'interpolation sur un maillage cartésien sont mises à l'épreuve dans [CMS10]. L'adaptation des méthodes d'interpolation par spline cubique et par polynôme d'Hermite sur un maillage polaire a été réalisée dans [SMC+15].

Avant de pouvoir intégrer la méthode de calcul à base d'interpolation dans le code de production GYSELA, un travail d'optimisation a été réalisé afin de réduire son temps de calcul. Cela est nécessaire afin que les simulations qui bénéficient de ce nouvel opérateur se réalisent dans un temps raisonnable. Le travail d'optimisation qui est détaillé dans ce chapitre fait l'objet de l'article [RSL+15].

4.3.1 Principe du calcul par méthode d'interpolation

Le calcul de l'opérateur de gyromoyenne par intégration directe implique l'utilisation intensive d'un opérateur d'interpolation sur le cercle de Larmor. Considérons une fonction f discrétisée sur un plan dont on veut calculer la gyromoyenne. Pour effectuer le calcul, nous considérons N points de quadrature répartis de façon uniforme sur le cercle de Larmor. Nous estimons la valeur de la fonction f en ces points grâce à un opérateur d'interpolation $\mathcal{P}$. Dans notre étude, l'opérateur d'interpolation construit une fonction définie par morceaux à base de polynômes. Selon le choix de l'opérateur d'interpolation, les contraintes de construction des polynômes sont différentes, typiquement aux extrémités de leur domaine de définition.

À partir de la définition de l'opérateur de gyromoyenne discret (4.9), son évaluation à la position $\vec{x}_G = (r_i, \theta_j)$ se réécrit de la façon suivante :

$$\mathcal{J}_\rho(f)(r_i,\theta_j) = \frac{1}{2\pi}\sum_{k=0}^{N-1} \Delta\alpha \mathcal{P}(f)(r_i\cos\theta_j + \rho\cos\alpha_k,\ r_i\sin\theta_j + \rho\sin\alpha_k). \tag{4.10}$$

Il est à noter que dans cette dernière équation, les positions où la fonction f doit être évaluée sont exprimées dans les coordonnées cartésiennes du plan. La Figure 4.2 illustre cette définition avec $N = 5$ et $\vec{x}_G = (r_3, \theta_3)$.

Lorsqu'une position d'un point d'interpolation dans (4.10) se situe à l'extérieur du domaine de définition du plan poloïdal, nous projetons radialement cette position sur les bords du domaine :

- si $r < r_{\min}$ alors $\mathcal{P}(f)(r,\theta)$ est remplacé par $\mathcal{P}(f)(r_{\min},\theta)$,
- si $r > r_{\max}$ alors $\mathcal{P}(f)(r,\theta)$ est remplacé par $\mathcal{P}(f)(r_{\max},\theta)$.

Dans le cadre de cette thèse, les efforts de développement se sont concentrés sur la méthode d'interpolation d'Hermite. Ce choix vient du fait que cette méthode d'interpolation est moins contraignante que l'interpolation par spline cubique naturelle. La méthode par spline cubique nécessite de connaître l'ensemble des points dans la direction où on souhaite faire l'interpolation. Cette contrainte liée à la méthode de calcul n'est pas souhaitable dans un code parallèle. Néanmoins on peut noter que l'article [CLS07] propose une adaptation de cette méthode lorsque les données sont distribuées. Dans notre implémentation de l'interpolation d'Hermite, l'estimation des dérivées en un point se fait grâce aux valeurs de la fonction au voisinage de ce point [Li05]. Par rapport à l'interpolation par spline cubique, la précision globale est sensiblement moins bonne mais suffisante. L'aspect local de l'interpolation d'Hermite est un atout pour sa parallélisation.

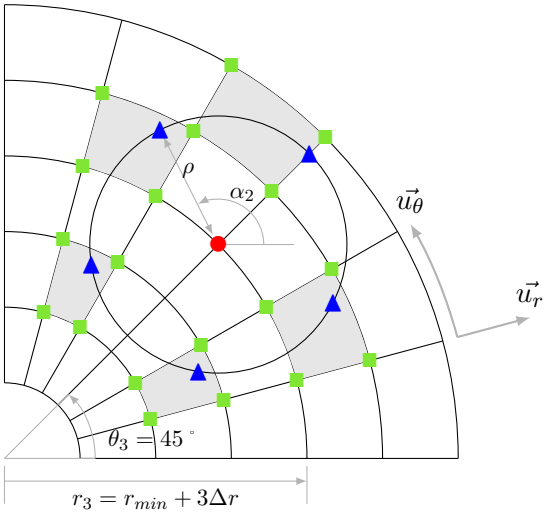


FIGURE 4.2 – Illustration du calcul de gyromoyenne basé sur une méthode d'interpolation. La gyromoyenne est ici calculée au point ● au centre du cercle. Pour ce faire, on calcule la moyenne de la valeur des points d'interpolation de la fonction f sur le cercle ▲. L'évaluation de la valeur d'un point d'interpolation ▲ se base sur des valeurs de f connues sur les coins de la cellule ■ à laquelle il appartient.

De plus, bien que nous utilisions cette méthode d'interpolation dans le plan poloïdal muni d'un système de coordonnées polaire, la méthode peut s'étendre à un système de coordonnées quelconque.

Nos travaux se sont donc concentrés autour du calcul de gyromoyenne basé sur l'interpolation d'Hermite. La suite de ce chapitre détaille cette méthode d'interpolation dans les cas $1D$ et $2D$ dans un plan polaire. Les optimisations de la gyromoyenne d'un plan $2D$ basées sur cette méthode d'interpolation sont présentées dans un second temps.

4.3.2 Interpolation par polynôme d'Hermite

L'interpolation d'Hermite reconstruit une fonction polynomiale par morceaux d'une fonction donnée f discrétisée. Sur chaque cellule du maillage qui intervient dans le calcul d'une gyromoyenne (zones grisée sur la Figure 4.2), l'interpolation d'Hermite reconstruit des polynômes d'ordre 3. Ces polynômes sont construits de telle manière que la valeur et la dérivée première de ces polynômes soient égales à celles de la fonction d'entrée discrète aux coins des cellules.

Interpolation sur un maillage $1D$

Nous détaillons ici l'interpolation d'Hermite d'ordre 3 dans le cas unidimensionnel avant de prendre en compte le cas $2D$ dans un système de coordonnées polaires. La méthode de construction résumée ici s'inspire de [Dub06] et [Fin04].

Considérons une fonction f définie sur un domaine $[x_{min}, x_{max}] \subset \mathbb{R}$ divisé en N cellules :

$$C_i = [x_i, x_{i+1}], \quad \text{avec } i \in [\![0, N-1]\!].$$

On suppose que le maillage est uniforme, c'est-à-dire pour tout indice i le pas d'espace Δx

vérifie :

$$\Delta x = x_{i+1} - x_i = \frac{b-a}{N}.$$

Sur chaque cellule C_i, on cherche à définir le polynôme d'Hermite P_i ; il respecte les deux contraintes suivantes : (i) il passe par les points $(x_i, f(x_i))$ et $(x_{i+1}, f(x_{i+1}))$ et (ii) la valeur de sa dérivée en ces deux points correspond à celle de la fonction d'entrée. Afin de calculer les coefficients du polynôme P_i, on se ramène classiquement à l'intervalle $[0,1] \subset \mathbb{R}$ grâce à un changement de variable. Soit g une fonction auxiliaire définie sur cet intervalle telle que :

$$g(\alpha) = f(x), \text{ avec } \alpha \in [0,1], x \in [x_i, x_{i+1}]$$

et

$$x = x_i + \alpha \Delta x, \qquad x \in [x_i, x_{i+1}]. \tag{4.11}$$

On cherche le polynôme $P_i(\alpha) = a_3\alpha^3 + a_2\alpha^2 + a_1\alpha + a_0$ qui interpole la fonction g sur l'intervalle $[0,1]$. Il respecte les contraintes :

$$\begin{aligned} P_i(0) &= g(0) & P_i(1) &= g(1) \\ P_i'(0) &= g'(0) & P_i'(1) &= g'(1). \end{aligned}$$

Les calculs qui nous permettent d'aboutir à l'expression du polynôme P_i sont dans l'annexe C.1. L'expression de P_i s'écrit à l'aide des fonctions de la base d'Hermite $(\varphi_i)_{1 \leqslant i \leqslant 4}$:

$$\begin{aligned} P_i(\alpha) = {} & \varphi_1(\alpha) f(x_i) + \varphi_2(\alpha) f(x_{i+1}) + \\ & \Delta x(\varphi_3(\alpha) f'(x_i) + \varphi_4(\alpha) f'(x_{i+1})). \end{aligned} \tag{4.12}$$

$$\text{avec } \begin{cases} \varphi_1(\alpha) = & 2\alpha^3 - 3\alpha^2 + 1 \\ \varphi_2(\alpha) = & -2\alpha^3 + 3\alpha^2 \\ \varphi_3(\alpha) = & \alpha^3 - 2\alpha^2 + \alpha \\ \varphi_4(\alpha) = & \alpha^3 - \alpha^2 \end{cases}$$

Les polynômes $(\varphi_i)_{1 \leqslant i \leqslant 4}$ de degré 3 forment la base d'Hermite. Cela signifie que tout polynôme de degré 3 peut se décomposer comme une combinaison linéaire des fonctions de cette base. Les polynômes (φ_i) satisfont les contraintes suivantes :

$$\begin{aligned} \varphi_1(0) &= 1, & \varphi_3(0) &= 0, & \varphi_2(0) &= 0, & \varphi_4(0) &= 0, \\ \varphi_1'(0) &= 0, & \varphi_3'(0) &= 1, & \varphi_2'(0) &= 0, & \varphi_4'(0) &= 0, \\ \varphi_1(1) &= 0, & \varphi_3(1) &= 0, & \varphi_2(1) &= 1, & \varphi_4(1) &= 0, \\ \varphi_1'(1) &= 0, & \varphi_3'(1) &= 0, & \varphi_2'(1) &= 0, & \varphi_4'(1) &= 1. \end{aligned}$$

L'allure des fonctions $(\varphi_i)_{1 \leqslant i \leqslant 4}$ est donné sur la Figure 4.3.

Le polynôme P_i est une approximation de la fonction f entre x_i et x_{i+1} :

$$f(x_i + \alpha \Delta x) \approx P_i(\alpha)$$

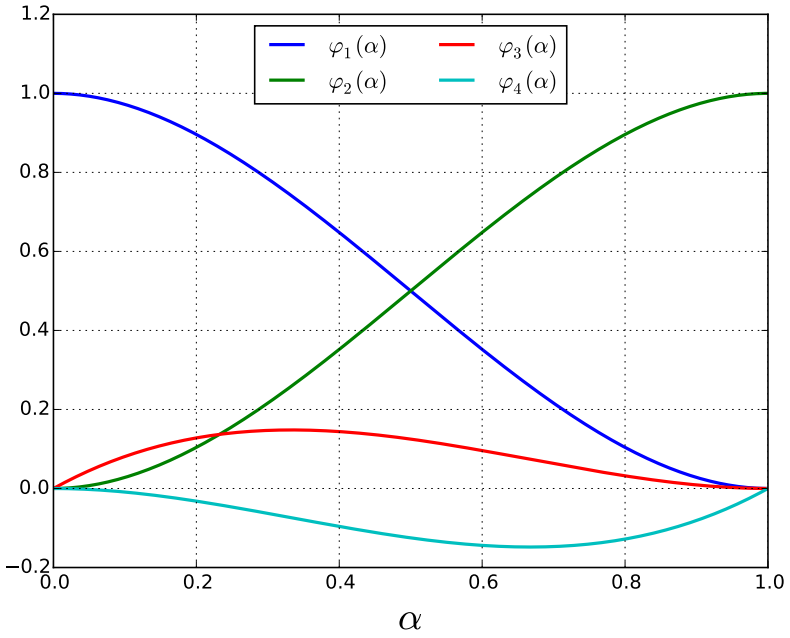


FIGURE 4.3 – Représentation graphique des fonctions de la base d'Hermite.

Estimation des dérivées sur un maillage $1D$

La construction des polynômes $(P_i)_{0\leqslant i\leqslant N-1}$ s'appuie sur la connaissance des pentes de la fonction f en chaque point de grille $(x_i)_{0\leqslant i\leqslant N}$ du maillage. Ces pentes doivent être reconstruites car nous ne disposons en entrée que des valeurs discrètes de f sur le maillage. Dans notre approche, nous estimons ces dérivées par différences finies. Cette méthode utilise un stencil de n points au voisinage du point dont on souhaite calculer la dérivée. Plus n est grand, meilleure sera l'estimation de la dérivée. L'article [Li05] fournit la démonstration du calcul de ces dérivées. Indépendamment de l'ordre n de reconstruction des dérivées, le polynôme d'Hermite reste d'ordre 3. Cette estimation des dérivées nous permet de conserver le caractère local de la méthode d'interpolation.

Dans notre cas, nous utilisons un stencil centré de d points, c'est-à-dire qu'il y a $d/2$ points à gauche et à droite du point dont on veut calculer la dérivée, dans le cas où d est pair. La démonstration qui permet d'aboutir à l'expression des coefficient de pondération ainsi que plusieurs exemples d'utilisation sur différents stencils sont donnés dans l'article [Li05].

Plus d est grand, meilleure est l'estimation de la dérivée. Néanmoins, l'article [SMC+15] montre que l'utilisation de grandes valeurs de d n'améliorent pas nécessairement la précision de l'opérateur de gyromoyenne.

On peut aussi noter le fait que dans le cas où d est impair, le stencil est décentré. On peut tirer profit d'un nombre impair de points pour calculer les dérivées en x_i^-, si le stencil compte un point de plus sur la partie à gauche par rapport à la partie à droite du point considéré, et en x_i^+ dans la situation opposée. Le polynôme P_i se réécrirait alors :

$$\begin{aligned} P_i(\alpha) = \varphi_1(\alpha)f(x_i) + \varphi_2(\alpha)f(x_{i+1}) + \\ \Delta x(\varphi_3(\alpha)f'(x_i^+) + \varphi_4(\alpha)f'(x_{i+1}^-)). \end{aligned}$$

Dans cette approche, généralement $f'(x_i^-) \neq f'(x_i^+)$. Ce qui implique que la fonction polynomiale par morceaux basée sur les P_i est uniquement C^0. Dans le cas où d est pair, cette fonction est C^1. C'est pourquoi nous utilisons une paire de d dans notre étude.

[Li05] fournit une série d'exemples explicites du calcul de dérivée sur différentes tailles de stencil et pour différentes positions du stencil par rapport au point où on souhaite estimer la dérivée. Le stencil centré avec $n = 5$ points correspond dans notre implémentation au cas $d = 4$. Avec cette taille de stencil, la dérivée en x_i se calcule de la façon suivante :

$$f'(x_i) \approx \frac{1}{12\Delta x}(f(x_{i-2}) - 8f(x_{i-1}) + 8f(x_{i+1}) - f(x_{i+2})).$$

Dans la suite de ce manuscrit, d vaut 4. Aux extrémités des directions non périodiques, nous avons choisi de réutiliser la dernière valeur autant de fois que nécessaire pour compléter le stencil. Par exemple en x_0 :

$$f'(x_0) \approx \frac{1}{12\Delta x}(f(x_0) - 8f(x_0) + 8f(x_1) - f(x_2)).$$

Interpolation sur un maillage polaire $2D$

Considérons maintenant le plan polaire uniforme défini sur le domaine $[r_{\min}, r_{\max}] \times [0, 2\pi]$ avec $N_r \times N_\theta$ cellules :

$$C_{ij} = [r_i, r_{i+1}] \times [\theta_j, \theta_{j+1}], \qquad \text{avec } i \in [\![0, N_r - 1]\!],\ j \in [\![0, N_\theta - 1]\!]$$

où

$$\begin{aligned} r_i &= r_{\min} + i\frac{r_{\max} - r_{\min}}{N_r}, & i &\in [\![0, N_r]\!] \\ \theta_j &= j\frac{2\pi}{N_\theta}, & j &\in [\![0, N_\theta]\!]. \end{aligned}$$

L'interpolation d'Hermite sur un plan $2D$ correspond à une succession d'interpolations unidimensionnelles. La Figure 4.4 illustre les étapes de calcul intermédiaires. Aux points ♦ (r, θ_j) et (r, θ_{j+1}), la fonction f est interpolée pour obtenir sa valeur et la valeur de sa dérivée suivant la direction θ en ces points[14]. Puis ces valeurs sont utilisées pour estimer f au point cible (r, θ) au sein d'une cellule.

Le polynôme $P_{i,j}$ peut s'exprimer comme une combinaison linéaire des fonctions :

$$f(r,\theta), \frac{\partial}{\partial r}f(r,\theta), \frac{\partial}{\partial \theta}f(r,\theta), \frac{\partial^2}{\partial r \partial \theta}f(r,\theta)$$

évaluées au 4 coins de la cellule $C_{i,j}$. Nous appellerons ce groupe de 4 valeurs W_f dans la suite de ce manuscrit. Les détails des calculs du polynôme $P_{i,j}$ sont disponibles dans l'annexe C.2.

De façon analogue à l'équation (4.12), les valeurs W_f évaluées aux 4 coins de la cellule pondèrent les fonctions de la base d'Hermite. En $2D$, les 16 fonctions de base sont de la forme $\big(\varphi_i(\alpha)\varphi_j(\beta)\big)_{1 \leqslant i,j \leqslant 4}$. L'allure de ces fonctions de base est donnée dans l'annexe C.3.

$P_{i,j}$ est une approximation de la fonction f sur $[r_i, r_{i+1}] \times [\theta_j, \theta_{j+1}]$:

$$f(r_i + \alpha\Delta r, \theta_j + \beta\Delta\theta) \approx P_{i,j}(\alpha, \beta).$$

14. Que les points intermédiaires ♦ soient pris sur les arêtes inférieure–supérieure ou les arêtes gauche–droite, ce choix ne change pas l'expression de $P_{i,j}$.

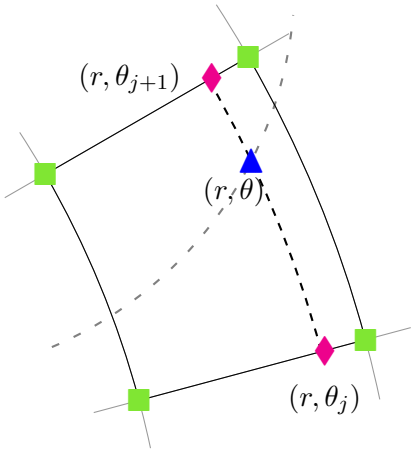


FIGURE 4.4 – Illustration des étapes intermédiaires de l'interpolation $2D$. Le point ▲ à l'intérieur de la cellule représente la position cible de l'interpolation. La valeur en ce point est obtenue grâce à une interpolation d'Hermite entre les points ♦ sur les arêtes inférieure et supérieure de la cellule. Les valeurs nécessaires aux points ♦ sont estimées par interpolation d'Hermite entre les points ■.

Remarque 1 : Soit M un point de l'espace de définition d'une fonction f multidimensionnelle. Selon le théorème de Schwarz [15], si une dérivée d'ordre $k \in \mathbb{N}$ est continue en M, alors la valeur de cette dérivée ne dépend pas de l'ordre des opérateurs de dérivation. Dans notre cas, cela se traduit par

$$\frac{\partial}{\partial r}\left(\frac{\partial f}{\partial \theta}\right) = \frac{\partial}{\partial \theta}\left(\frac{\partial f}{\partial r}\right).$$

Dans le contexte des fonctions discrètes à plusieurs variables, l'application de ce théorème est mise en œuvre dans [ASAH12] qui est la généralisation des formules de [Li05].

Remarque 2 : En pratique, le fait que le plan soit polaire est crucial lorsque pour une position d'interpolation donnée (▲), les valeurs α et β doivent être calculées.

L'évaluation du polynôme $P_{i,j}$ en (α, β) nous donne la valeur de la fonction f en un point d'interpolation. Pour calculer la gyromoyenne d'un point du maillage, N points d'interpolation sur le cercle de Larmor autour de ce dernier doivent être estimés. En réalité, ces points d'interpolation font intervenir un nombre restreint de cellules du maillage d'entrée, ce qui signifie que leur estimation fait intervenir un petit nombre de W_f du plan. De façon intuitive on peut entrevoir à cette étape des optimisations potentielles pour le calcul de gyromoyenne dans le cas où on souhaite appliquer ce calcul sur l'ensemble des points de grille d'un plan $2D$. La partie suivante donne une description matricielle de l'opérateur de gyromoyenne sur un plan.

15. https://en.wikipedia.org/wiki/Symmetry_of_second_derivatives

4.4 Description matricielle de l'opérateur de gyromoyenne basé sur l'interpolation d'Hermite

La section précédente décrit la méthode pour réaliser l'interpolation en un point. La gyromoyenne (4.10) faisant intervenir N points, un premier algorithme simple serait de répéter le calcul d'interpolation N fois. Néanmoins, on voit clairement que le calcul d'interpolation se résume à une combinaison linéaire de W_f qui sont potentiellement partagées par plusieurs points d'interpolation. Il suffit que 2 positions d'interpolation partagent la même cellule pour faire intervenir les mêmes valeurs W_f dans leur calcul.

Afin de clarifier de calcul de gyromoyenne sur un plan poloïdal, nous allons décrire dans cette section une représentation matricielle de cette opérateur. Cette formulation nous permettra d'identifier les propriétés exploitables pour l'optimisation sur l'algorithme du calcul lui même, mais aussi pour les structures de données.

4.4.1 Description matricielle

En considérant un plan poloïdal de $N_r \times N_\theta$ cellules, la gyromoyenne de ce plan à base d'interpolation d'Hermite peut s'écrire de la façon de suivante :

$$M_{final} = M_{coef} \times M_{fval}$$

$$\text{avec} \begin{cases} M_{final} & \in \mathcal{M}_{N_r+1,\ N_\theta}(\mathbb{R}), \\ M_{coef} & \in \mathcal{M}_{N_r+1,\ 4(N_r+1)N_\theta}(\mathbb{R}), \\ M_{fval} & \in \mathcal{M}_{4(N_r+1)N_\theta,\ N_\theta}(\mathbb{R}). \end{cases}$$

La matrice M_{final} représente le résultat de la gyromoyenne appliquée sur tous les points de grille du plan poloïdal. Tous les points de cette matrice vérifient la relation :

$$M_{final}(i,j) = \mathcal{J}_\rho(f)_{r_i,\theta_j}$$

$$\text{avec} \begin{cases} r_i = & r_{\min} + i\frac{r_{\max}-r_{\min}}{N_r}, & i \in [\![0, N_r]\!], \\ \theta_j = & j\frac{2\pi}{N_\theta}, & j \in [\![0, N_\theta - 1]\!]. \end{cases}$$

Le dernier point dans la direction θ est exclu puisqu'il partage la même position que le premier point dans cette direction périodique. Chaque ligne R_i de la matrice M_{coef} contient les coefficients de contribution associés au plan d'entrée en une position radial i donnée. Ces coefficients représentent la somme des contributions des N points d'interpolation pour une valeur W_f. Chaque colonne C_j de la matrice M_{fval} contient l'ensemble des valeurs W_f nécessaires dans l'interpolation d'Hermite.

Dans cette reformulation, le calcul de gyromoyenne en un point de grille se résume au produit scalaire suivant :

$$\mathcal{J}_\rho(f)_{r_i,\theta_j} = R_i(M_{coef}) \times C_j(M_{fval}). \tag{4.13}$$

Dans la suite de ce manuscrit, le couple d'indices (i,j) est lié au couple de valeurs réelles (r_i, θ_j).

4.4.2 Implémentation initial de l'opérateur de gyromoyenne

Par convenance, les notations matricielles introduites précédemment sont utilisées pour décrire l'implémentation initiale [SMC+15]. La méthode de calcul et les structures de données

qui y sont mises en œuvre ont la même sémantique que celles de la description matricielle, mais ne sont pas clairement identifiées ce qui peut entraîner une perte de performance de l'implémentation.

D'après la représentation matricielle de la gyromoyenne, les deux principales propriétés de la matrice M_{coef} sont *(i)* qu'elle est creuse (voir section 4.4.3) et *(ii)* qu'elle ne change pas durant toute la simulation. Cette matrice peut donc être calculée une fois pour toute durant une étape d'initialisation, alors que la matrice M_{fval} est recalculée à chaque gyromoyenne car ses valeurs dépendent des valeurs du plan poloïdal d'entrée.

L'implémentation se décompose en deux étapes : une première étape d'initialisation décrite par l'Algorithme 3 où on pré-calcule les poids des contributions (M_{coef}) et une seconde étape décrite par l'Algorithme 4 qui correspond à un appel à la *subroutine* `gyro_compute()` durant les itérations temporelles.

Entrées : N_r, N_θ, ρ, N
Sorties : Une représentation de M_{coef} – les poids et les indices correspondant à leur *rhs_values* dans la structure représentant M_{fval} pour chaque i.

pour $i \leftarrow 0$ **à** N_r **faire**
  pour $k \leftarrow 0$ **à** $N-1$ **faire**
    Calculer la $k^{\text{ième}}$ position du point de quadrature pt sur le cercle de Larmor de centre $(i, \theta = 0)$;
    Calculer/stocker les indices des coins de la cellule cell contenant pt ;
    Calculer/stocker les poids associés à la contribution de chaque coin de cell dans une représentation de M_{coef};
  fin
fin

Algorithme 3 : Calcul d'une représentation de M_{coef} (étape d'initialisation).

Entrées : Valeurs de la fonction f sur le plan poloïdal plan_2D
Sorties : Gyromoyennes des points de grille du plan poloïdal

pour *chaque point* pt (i, j) *de* plan_2D **faire**
  Calculer/stocker les dérivées premières dans chaque direction et la dérivée croisée au point pt (i, j) dans une représentation de $C_0(M_{fval})$;
fin
pour *chaque point* pt (i, j) *de* plan_2D **faire**
  Calculer la gyromoyenne au point pt (i, j)– effectuer le produit scalaire creux $R_i(M_{coef}) \times C_j(M_{fval})$;
  Stocker le résultat dans plan_2D en pt (i, j) ;
fin

Algorithme 4 : Calcul de gyromoyenne de la fonction f connue sur un plan poloïdal (durant un pas de temps).

4.4.3 Motif de la matrice M_{coef}

Pour se rendre compte de l'aspect creux de la matrice M_{coef}, regardons la Figure 4.2 (p. 66). La i-ème ligne de la matrice M_{coef} contient les coefficients de contribution de toutes les positions

du plan d'entrée pour la gyromoyenne au point $(i, j = *)$. Comme nous pouvons le voir sur la Figure 4.2, un petit nombre de points du plan (■) contribuent au calcul de la gyromoyenne du point au centre du cercle (●). Par conséquent le poids associé à un point du plan est non nul pour ce calcul de gyromoyenne uniquement s'il intervient dans un des calculs d'interpolation (▲). C'est le cas pour tout point du plan. Cela nous a mène à conclure que la matrice M_{coef} est creuse. La Figure 4.5 illustre le motif générique du i-ème vecteur ligne de M_{coef}.

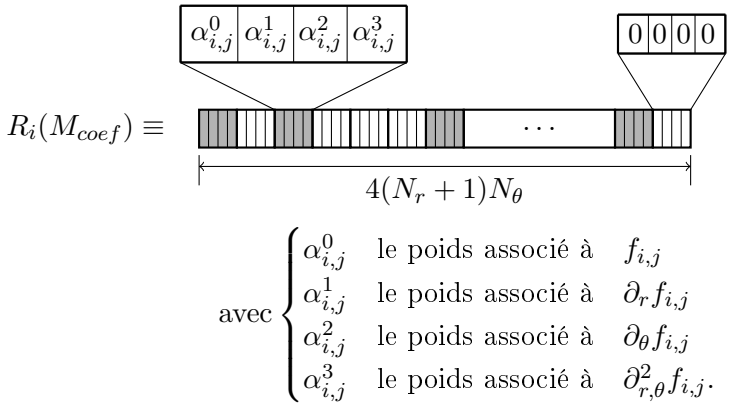


FIGURE 4.5 – Schéma de l'aspect creux d'un vecteur ligne $R_i(M_{coef})$. Sur cette exemple, seules les cellules grisées contiennent des valeurs non nulles $\alpha^*_{i,j}$.

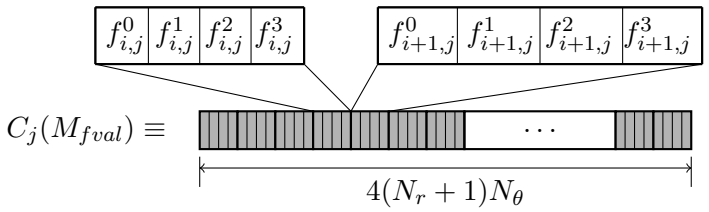
La valeur des coefficients de contribution est invariante par rapport à l'index dans la direction poloïdale j. Ceci est dû au fait que pour 2 points distincts du plan d'entrée $(i,\ j_1)$ et $(i,\ j_2)$, les coordonnées polaires des points d'interpolation intervenant dans leur calcul de gyromoyenne respectif sont les mêmes dans le système de coordonnées local le long de la direction poloïdale $(\vec{u_r},\ \vec{u_\theta})$. La valeur et la distribution des coefficients $\alpha^*_{i,j}$ dépendent uniquement de l'indice radial i.

Le nombre de valeurs non nulles de deux vecteurs lignes distincts de M_{coef} peut varier. Sur un maillage polaire donné, le motif des cellules qui contribuent au calcul de gyromoyenne dépend uniquement de l'indice radial i du point auquel on calcule la gyromoyenne. Généralement, les points proches du centre du maillage (i.e. des faibles valeurs de i) font intervenir un plus grand nombre de cellules par rapport aux cellules à l'extérieur du plan.

4.4.4 Motif de la matrice M_{fval}

La matrice M_{fval} stocke les valeurs nécessaires à l'interpolation d'Hermite, à savoir pour chaque point de la grille d'entrée, la valeur de la fonction, ses dérivées premières dans chaque direction ainsi que sa dérivée croisée (W_f). La Figure 4.6 illustre le contenu d'un des vecteurs colonne de cette matrice.

La taille d'un vecteur $C_j(M_{fval})$ est de $4(N_r+1)N_\theta$ et contient uniquement des valeurs non nulles. C'est donc une matrice dense. Dans la représentation actuelle (voir Figure 4.7), on peut remarquer que M_{fval} a un motif répétitif. La distribution des valeurs dans un vecteur $C_j(M_{fval})$ dépend de son indice poloïdal j. Les valeurs du vecteur $C_{j+1}(M_{fval})$ peuvent être obtenues par permutation circulaire de $4(N_r+1)$ positions des valeurs de $C_j(M_{fval})$. Le motif de la matrice M_{fval} est illustré par la Figure 4.7. À partir du premier vecteur colonne, les autres peuvent être



$$\text{avec} \begin{cases} f^0_{i,j} = f_{i,j} \\ f^1_{i,j} = \partial_r f_{i,j} \\ f^2_{i,j} = \partial_\theta f_{i,j} \\ f^3_{i,j} = \partial^2_{r,\theta} f_{i,j}. \end{cases}$$

FIGURE 4.6 – Schéma de l'agencement des éléments dans un vecteur colonne $C_j(M_{fval})$.

déduit par la relation :

$$C_{j\neq \mathrm{o}}(M_{fval})[i] = C_{\mathrm{o}}(M_{fval})[i - j_\theta \times 4(N_r+1)]. \tag{4.14}$$

La formulation matricielle de l'opérateur de gyromoyenne fournie dans cette section nous apporte une vision synthétique du calcul de gyromoyenne pour un plan. La section suivante présente les différentes optimisations mises en place.

4.5 Optimisation de l'opérateur de gyromoyenne basé sur l'interpolation d'Hermite

Le calcul de l'opérateur de gyromoyenne se décompose en 2 étapes, comme cela a déjà été mentionné dans la section 4.4.2. L'Algorithme 3 est calculé une seule fois durant l'initialisation. Le temps d'exécution attribué à ce calcul est considéré comme un coût d'amorçage, il ne fait pas l'objet du travail d'optimisation présenté ici. L'Algorithme 4 est appelé plusieurs fois durant une itération temporelle de GYSELA et représente le noyau de calcul principal de l'opérateur de gyromoyenne. Les optimisations qui ont été réalisées se concentrent sur la réduction du temps d'exécution de cet algorithme. Les structures de données qui représentent les matrices ont fait l'objet d'une attention particulière. Dans un second temps, des optimisations concernant les aspects caches sont mises en œuvre.

Dans cette section, nous décrivons plusieurs implémentations. La première implémentation présentée en 4.5.1 doit être considérée comme le point de départ pour les optimisations introduites en 4.5.2 et 4.5.3. Enfin cette section se conclut sur un position des optimisations mises en œuvres par rapport aux méthodes existantes en 4.5.4.

4.5.1 Structure de données pour un vecteur creux

Le choix des structures de données pour représenter M_{coef} et M_{fval} a un important impact sur les performances. Il a été montré que la matrice M_{coef} est creuse dans 4.4.3. Pour profiter de cette propriété, nous considérons uniquement les valeurs non nulles de M_{coef}. En évitant le stockage des valeurs nulles, la taille des vecteurs intervenant dans l'équation 4.13 est réduite, mais cela nécessite l'utilisation de structures dédiées pour réaliser l'algorithme 4.

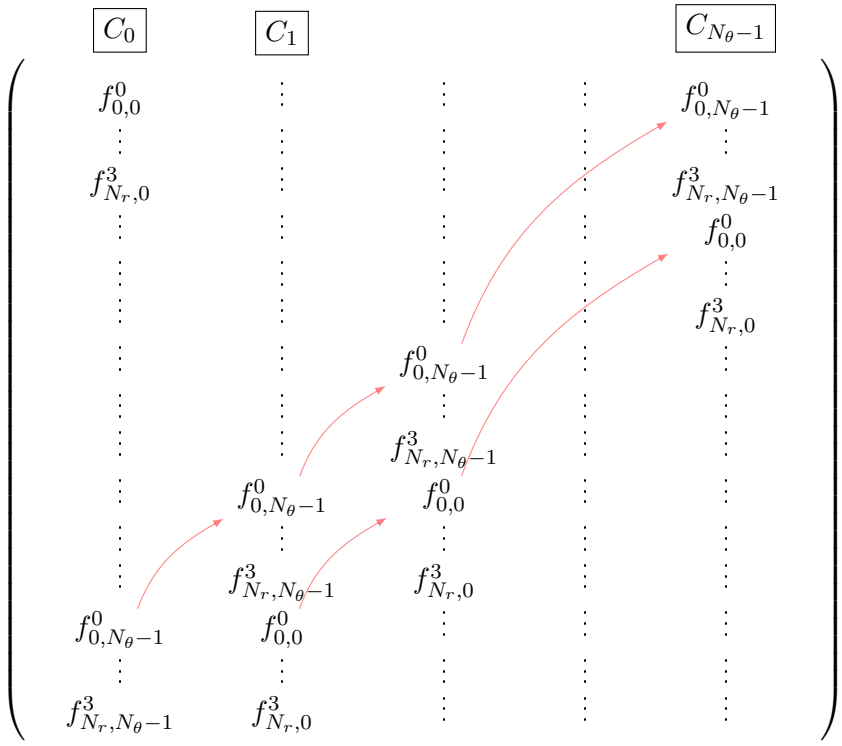


FIGURE 4.7 – Illustration de la permutation circulaire des valeurs W_f entre les colonnes de M_{fval}.

La principale contribution de l'optimisation présentée ici concerne le choix des structures de données. Parmi les différentes représentations possibles de M_{fval}, deux d'entre elles sont détaillées et comparées ci-dessous.

La représentation de la matrice M_{coef}

Pour profiter de l'aspect creux de la matrice M_{coef}, une structure dédiée `contribution_vector` a été créée pour la manipuler. Cette structure est définie de la façon suivante :

```
1  type :: gyro_vector
2    integer, dimension(:), pointer :: ind
3    real(8), dimension(:), pointer :: val
4  end type gyro_vector
5
6  type(gyro_vector), dimension(:), pointer :: contribution_vector
```

Dans cet extrait de code FORTRAN, la variable `contribution_vector` est un tableau de longueur $(N_r + 1)$ et de type `gyro_vector` qui est défini juste au-dessus. Chaque cellule de `contribution_vector` représente un vecteur ligne de la matrice M_{coef}. Pour un indice radial i donné, les indices des points du maillage sous-jacent qui contribuent au calcul de gyromoyenne du point $(i,\, j=0)$ sont calculés et stockés dans `contribution_vector(`i`)%ind`. Nous stockons la taille de ce vecteur dans une variable `nb_contribution_pt(`i`)`. Cette taille correspond au nombre de points de grille qui intervient dans le calcul de gyromoyenne pour le point $(i,\, j=0)$.

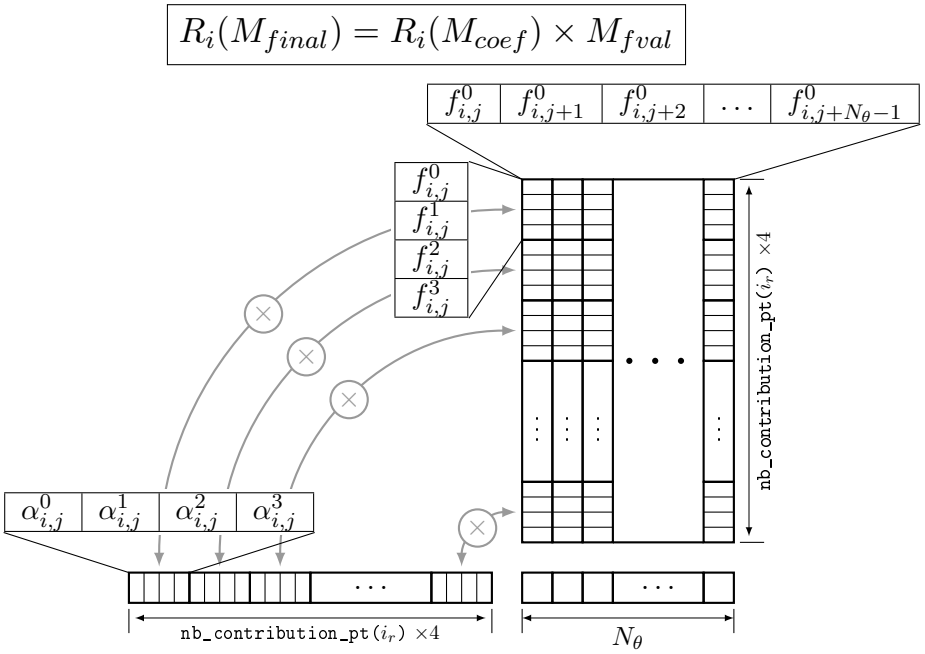


FIGURE 4.8 – Illustration du produit matrice-vecteur entre les poids (`contribution_vector(`i_r`)%val`) et les W_f du plan d'entrée (`rhs_product(`i_r`)%val`). Un produit calcule N_θ gyromoyennes.

Ce nombre de points varie en fonction de l'indice radial i (voir section 4.4.3). Les indices stockés dans `contribution_vector(`i`)%ind` sont utilisés pour construire une représentation de M_{fval} qui sera détaillée plus bas.

Les poids associés à ces positions sont calculés et stockés dans `contribution_vector(`i`)%val`. Dans le cas de l'interpolation d'Hermite, pour un point de grille, 4 poids sont stockés : un poids par valeur de W_f. Ce vecteur est donc 4 fois plus grand que `contribution_vector(`i`)%ind`.

Première représentation de la matrice M_{fval}

L'Algorithme 4 qui décrit le noyau de calcul principal de la gyromoyenne peut être décomposé en 2 parties : *(i)* le calcul des dérivées et *(ii)* le calcul du produit scalaire entre les poids et les W_f du plan d'entrée. À chaque appel à `gyro_compute`, les dérivées du plan d'entrée sont calculées et stockées dans une variable `rhs_product`. Cette variable représente la matrice M_{fval}. Dans cette implémentation, `rhs_product` est définie de la façon suivante :

```
1  type :: small_matrix
2  real(8), dimension(:, :), pointer :: val
3  end type small_matrix
4
5  type(small_matrix), dimension(:), pointer :: rhs_product
```

La variable `rhs_product` est un tableau de longueur $(N_r + 1)$; il contient des éléments de type `small_matrix` qui est une structure contenant simplement un pointeur $2D$. Les dimensions du tableau $2D$ `rhs_product(`i`)%val` sont $(4 \times$ `nb_contribution_pt(`i`)`, $N_\theta)$. Ce tableau est la matrice qui contient le sous-ensemble de valeurs de la matrice M_{fval}

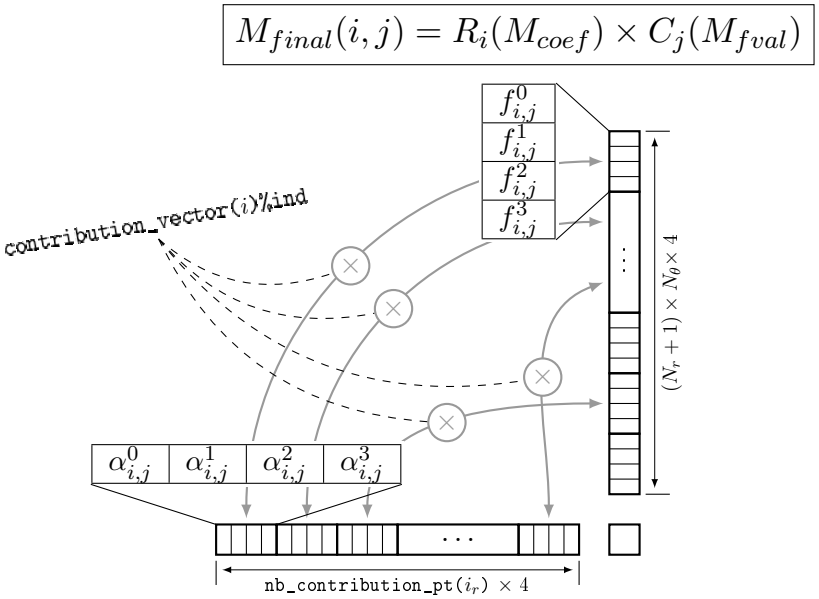


FIGURE 4.9 – Illustration du produit scalaire creux entre les poids (`contribution_vector(`i`)%val`) et les W_f du plan d'entrée (`rhs_product`). Les indirections dans le tableau `rhs_product` sont effectuées grâce aux positions enregistrées dans `contribution_vector(`i`)%ind`.

qui contribue à la gyromoyenne du point $(i,\ j\ =\ *)$. Son premier vecteur colonne `rhs_product(`i`)%val(:,`$j=0$`)` contient les W_f qui correspondent aux positions stockées dans le vecteur `contribution_vector(`i`)%ind`. Les autres colonnes `rhs_product(`i`)%val(:,`$j\neq 0$`)` sont déduites grâce à une permutation circulaire qui s'inspire de celle illustrée sur la Figure 4.7. Une fois que la matrice `rhs_product(`i`)%val` est construite, les tableaux `rhs_product(:)%ind` ne sont plus utilisés.

La Figure 4.8 donne une représentation graphique du produit matrice-vecteur entre les poids et les W_f. Ici, les W_f d'un vecteur `rhs_product(`i`)%val( : ,`j`)` sont ordonnées de telle sorte que leur position corresponde avec celle de leur poids stocké dans le vecteur `contribution_vector(`i`)%val`. De cette façon, la gyromoyenne peut être implémentée comme des produits matrice-vecteur qui sont des opérations efficaces sur les processeurs actuels. Chaque produit matrice-vecteur calcule la gyromoyenne de N_θ points. La gyromoyenne d'un plan poloïdal complet est réalisée grâce à (N_r+1) produits matrice-vecteur.

Seconde représentation de la matrice M_{fval}

Dans cette seconde représentation, la variable `rhs_product` représente toujours la matrice M_{fval}, mais elle est cette fois définie de la façon suivante :

```
1   real(8), dimension(:), pointer :: rhs_product
```

La variable `rhs_product` est ici un tableau $1D$. Il correspond exactement au premier vecteur colonne $C_0(M_{fval})$. Comme dans la section 4.4.4, ce vecteur contient toutes les valeurs W_f déduites du plan d'entrée. Sa taille est donc de $4(N_r+1)N_\theta$. On peut voir ce vecteur comme une représentation linéaire du plan $2D$ donné en entrée.

$N_r = N_\theta$		32	64	128	256	512
Structure 1 :	RHS	$1{,}5 \cdot 10^{-3}$	$1{,}1 \cdot 10^{-2}$	$8{,}5 \cdot 10^{-2}$	$5{,}1 \cdot 10^{-1}$	2,3
	Produit	$2{,}1 \cdot 10^{-4}$	$1{,}7 \cdot 10^{-3}$	$1{,}4 \cdot 10^{-2}$	$8{,}3 \cdot 10^{-2}$	$3{,}5 \cdot 10^{-1}$
	Total	$1{,}7 \cdot 10^{-3}$	$1{,}3 \cdot 10^{-2}$	$9{,}9 \cdot 10^{-2}$	$5{,}9 \cdot 10^{-1}$	2,7
Structure 2 :	RHS	$1{,}7 \cdot 10^{-4}$	$6{,}4 \cdot 10^{-4}$	$2{,}5 \cdot 10^{-3}$	$1{,}0 \cdot 10^{-2}$	$4{,}2 \cdot 10^{-2}$
	Produit	$7{,}9 \cdot 10^{-4}$	$5{,}7 \cdot 10^{-3}$	$4{,}6 \cdot 10^{-2}$	$2{,}7 \cdot 10^{-1}$	1,2
	Total	$9{,}7 \cdot 10^{-4}$	$6{,}3 \cdot 10^{-3}$	$4{,}9 \cdot 10^{-2}$	$2{,}8 \cdot 10^{-1}$	1,3

TABLE 4.1 – Comparaison de performances des 2 structures de données sur le cas test analytique. La 'Structure 1' correspond à la première représentation de M_{fval} et 'Structure 2' à la seconde. Les temps d'exécution sont donnés en secondes. Se sont les moyennes des temps sur 100 exécutions. ($\rho = 0.05$, $r_{min} = 0.1$, $r_{max} = 0.9$, $N = 32$).

La Figure 4.9 montre le produit entre les poids et les W_f en accord avec la présente implémentation. Les longueurs des vecteurs intervenant dans ce produit ne correspondent pas. Le lien entre les poids et les W_f se fait grâce aux positions des W_f concernées et stockées dans le vecteur `contribution_vector(`i`)%ind`. Ce lien est graphiquement représenté par les flèches. Les gyromoyennes des points (i, $j \neq 0$) sont calculées grâce à une permutation circulaire de $4j(N_r + 1)$ des indices de `contribution_vector(`i`)%ind`. Cette opération sera appelée *produit scalaire creux* dans les sections suivantes.

Pour réaliser la gyromoyenne du plan poloïdal en entier, $(N_r + 1)N_\theta$ produits scalaires creux sont nécessaires.

Benchmark pour comparer les structures de données

Pour comparer les 2 structures de données introduites plus haut, nous avons utilisé le cas test analytique comme microbenchmark (voir 5.1). Une exécution de ce programme consiste à calculer la gyromoyenne sur un plan initialisé par une fonction dont on connaît la formule analytique de la gyromoyenne. Considérons la fonction de complexité `comp(`N_r`,` N_θ`)` du produit entre les poids et les W_f. Cette fonction compte le nombre de multiplications requises par le calcul. Dans notre cas, elle s'écrit :

$$\texttt{comp}(N_r,\ N_\theta) = \sum_{i=0}^{N_r} 4 \times \texttt{nb\_contribution\_pt}(i) \times N_\theta.$$

Cette relation met en avant le lien fort entre le nombre de points de grille et la complexité en temps de la gyromoyenne. On peut aussi noter que pour le calcul d'une gyromoyenne, le nombre de multiplications en jeu est proportionnel au nombre de points du maillage d'entrée qui contribuent à la gyromoyenne. Ce nombre dépend du pas en espace du maillage et du nombre N.

Pour identifier et comparer le comportement des 2 implémentations de la gyromoyenne, nous avons fait varier la taille du plan poloïdal. Le Tableau 4.1 montre les temps d'exécution pour les différentes tailles. Les chiffres qui figurent dans ce tableau représentent la moyenne des temps obtenus sur 100 exécutions du cas test analytique. Le temps correspondant à la construction de la variable `rhs_product` est donné dans les lignes "RHS" et le temps correspondant aux produits entre les poids et les W_f dans les lignes "Produit".

La seconde implémentation de la structure de données est approximativement deux fois plus rapide que la première implémentation. En regardant le détail des résultats, la première implémentation est performante durant l'étape "Produit", par contre le temps nécessaire à la construction de `rhs_product` représente un surcoût conséquent. Puisque la seconde structure de données présente de meilleurs résultats, elle sert de base pour les optimisations suivantes.

Avec la seconde structure de données, les accès au tableau W_f ne sont pas contigus durant le produit scalaire creux (voir la Figure 4.9). Ceci est une conséquence du stencil irrégulier que fait apparaître l'opérateur de gyromoyenne (voir Figure 4.2) et de l'agencement des données dans la variable `rhs_product`. Ceci constitue une limite pour cette approche d'un point de vue performance.

Malgré le fait que la seconde structure de données soit plus délicate à manipuler pour le produit scalaire creux, cette structure nous permet d'avoir le contrôle sur l'agencement des données qu'elle contient. Cette possibilité représente le point clé de l'"optimisation par réagencement d'éléments" (détails en section 4.5.3).

Pour réaliser la gyromoyenne du plan complet, à chaque point de grille, un produit scalaire creux doit être effectué. Dans l'implémentation actuelle qui utilise la seconde structure de données pour M_{fval}, le plan poloïdal est parcouru grâce à 2 boucles "for" imbriquées où j est l'indice le plus interne. Géométriquement, le plan est exploré cercle après cercle, du plus petit au plus grand. Ce motif d'accès a la propriété de maximiser la réutilisation des poids (`contribution_vector`). Malgré cette propriété, ce parcours constitue une limite pour les performances et représente le point clé de l'"optimisation par blocage de boucle", détaillée dans la partie suivante. Nous nous référons à ce motif d'accès pour parcourir le plan sous le nom *parcours_plan* dans les sections suivantes.

4.5.2 Optimisation par blocage de boucle

Tel qu'il a été dit précédemment, pour réaliser le calcul de gyromoyenne sur l'intégralité du plan poloïdal, un produit scalaire creux doit être fait pour chaque point de grille. L'optimisation présentée ici se base sur l'implémentation de la seconde représentation de M_{fval} (voir 4.5.1). Cette optimisation se concentre sur l'amélioration du motif d'accès *parcours_plan* aux valeurs W_f du plan d'entrée inspiré par la technique de "blocage de boucle" [WL91].

Pour améliorer les performances, l'idée ici est de suivre un chemin *parcours_plan* qui diminue la fréquence des *caches miss* durant le calcul. Pour tout point du maillage (i,j), le calcul de la gyromoyenne requiert les données de `contribution_vector(`i`)` et de `rhs_product`. Tel qu'il a été dit dans la section 4.5.1, le calcul de gyromoyenne ce fait cercle après cercle pour assurer une bonne réutilisation des poids `contribution_vector(`i`)`. En ce qui concerne la variable `rhs_product`, ce parcours ne présente pas de propriété particulière. Les données accédées sont généralement non contiguës en mémoire et différentes pour chaque point du maillage. Néanmoins, puisque les données sont rapatriées par ligne de cache, le *taux d'accès* aux données en cache dépend du *parcours_plan* [Nys14]. Le but atteint dans cette optimisation a été d'augmenter la réutilisation des données disponibles dans le cache le long du *parcours_plan*.

Au lieu de parcourir le plan poloïdal cercle après cercle, le plan est parcouru en utilisant des carreaux $2D$ dans cette implémentation. Le plan poloïdal est partitionné en $NT_r \times NT_\theta$ carreaux. Ces carreaux sont de taille $BLOCK_r \times BLOCK_\theta$ et sont numérotés avec θ comme direction rapide, du centre vers l'extérieur du plan. La Figure 4.10 illustre la décomposition en carreaux d'une partie du maillage poloïdal. Sur cette exemple, $BLOCK_r = 3$ et $BLOCK_\theta = 4$. En pratique, les valeurs de $BLOCK_r$ et $BLOCK_\theta$ sont choisies de façon empirique grâce à des benchmarks sur plusieurs machines.

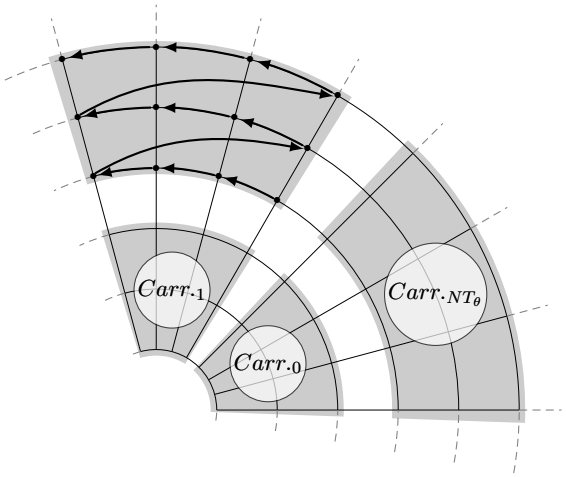


FIGURE 4.10 – Illustration de la technique de blocage de boucle sur une partie du plan poloïdal. Les carreaux qui partitionnent le maillage sont parcourus dans l'ordre croissant. Les flèches imbriquées à l'intérieur d'un carreau montre le parcours effectué sur ce dernier.

Le *parcours_plan* découpé ainsi en carreaux améliore la localité temporelle des données du tableau `rhs_product`. Cela augmente la réutilisation des données de ce tableau dans le cache.

4.5.3 Optimisation par ré-agencement d'éléments

Comme pour l'optimisation par blocage de boucle, l'optimisation présentée dans cette section se base sur l'implémentation de la seconde représentation de M_{fval} (voir 4.5.1). En particulier, le *parcours_plan* est identique dans l'optimisation présentée ici. L'optimisation de cette partie se concentre sur l'ordre des éléments dans le tableau `rhs_product`.

La démarche entreprise lors de la réalisation de cette optimisation se rapproche du travail présenté dans [YLPM05]. Ce dernier traite du problème de visualisation, ou plus précisément de traitement efficace de données à afficher. Les auteurs y exposent une méthode pour calculer l'ordre des éléments d'un tableau qui est consulté de façon intensive par différents algorithmes de rendu graphique.

Dans notre cas, pour un *parcours_plan* donné, l'arrangement des éléments de `rhs_product` a un fort impact sur le temps d'exécution. Nous ferons référence à cet agencement sous le nom *ré-agencement* de `rhs_product`.

Dans cette optimisation de l'agencement des éléments de `rhs_product`, le *parcours_plan* doit être considéré comme un paramètre d'entrée du problème. L'historique des accès aux éléments de `rhs_product` dépend du *parcours_plan*. L'idée de l'optimisation présentée ici est de permuter les éléments de `rhs_product`, donc de changer l'agencement des éléments, de telle sorte que la réutilisation des données dans le cache soit favorisée le long du *parcours_plan*.

En fonction de la valeur du rayon de giration ρ, le nombre de cellules $N_r \times N_\theta$, la valeur de r_{min} et r_{max}, le nombre de points sur le cercle de Larmor N et le *parcours_plan*, l'historique d'accès au tableau `rhs_product` est connu. Pour déterminer comment permuter les éléments de `rhs_product`, il est nécessaire d'introduire une métrique de distance d'accès temporel sur ces éléments.

Pour calculer la gyromoyenne au point (i,j), des W_f au voisinage de ce point sont nécessaires. Les points de ce voisinage sont les points de contributions (■) pour le calcul de gyromoyenne

point de contribution	liste des indices temporels d'accès	moyenne temporelle
(0, 0)	[0, 1, 8, 9, 10, 17]	7.0
(0, 1)	[0, 1, 8, 9, 10, 17, 18, 19, 26]	12.0
(0, 2)	[9, 10, 17, 18, 19, 26, 27, 28, 35]	21.0
(0, 3)	[18, 19, 26, 27, 28, 35, 36, 37, 44]	30.0
⋮	⋮	⋮
(1, 0)	[0, 1, 2, 9, 10, 11]	5.0
⋮	⋮	⋮

TABLE 4.2 – Pour chaque point du maillage, la liste des indices temporels d'accès le long de *parcours_plan* est donnée ainsi que la moyenne de ces indices.

au point (i, j). Le *parcours_plan* peut être vu simplement comme une liste ordonnée de la position des points du maillage. Un *indice temporel* d'un point du maillage correspond à un indice de *parcours_plan* pour lequel ce point de maillage contribue au calcul de gyromoyenne. Le Tableau 4.2 est un extrait de l'ensemble des associations entre des points de grilles et leurs indices temporels. Par exemple, grâce à ce tableau, on voit que le point `(0, 0)` est utilisé le long de *parcours_plan* durant les produit scalaires creux : 0, 1, 8, 9, 10 et 17. La seconde information fournit par ce tableau est la moyenne des indices temporels pour chaque point du maillage, que nous appelons *moyenne_temporelle*.

L'heuristique que nous avons implémentée ici consiste à trier les éléments de `rhs_product` par rapport à leur *moyenne_temporelle*. Par exemple, à partir des informations disponibles sur le Tableau 4.2, l'ordonnancement partiel des données dans le tableau `rhs_product` est [(1, 0), (0, 0), (0, 1), (0, 2), (0, 3), ...]. L'idée principale est de conserver les données accédées par 2 produits scalaires successifs aussi proches que possible pour augmenter les chances de réutilisation de données dans le cache.

Soit *tmp_moy*() une fonction qui retourne la *moyenne_temporelle* d'un point du plan. L'ordre des éléments proposé ici dans cette heuristique minimise la quantité :

$$\sum_{p=0}^{N_r \times N_\theta - 2} |tmp\_moy(parcours\_plan(p+1)) - tmp\_moy(parcours\_plan(p))|.$$

En pratique, le calcul des permutations des éléments de `rhs_product` est fait durant l'étape d'initialisation. Cela implique que le *parcours_plan* doit être connu dès l'étape d'initialisation. La correspondance entre les poids (`contribution_vector()%val`) et les W_f (`rhs_product`) durant le produit scalaire creux est fait grâce aux indices du tableau `contribution_vector()%ind`. Pour garantir la validité du calcul, les indices stockés dans `contribution_vector()%ind` doivent être conformes à ce nouvel agencement des éléments du tableau `rhs_product`.

4.5.4 Catégorisation algorithmique des optimisations des effets de cache

Les optimisations qui tentent de bénéficier des effets de cache se décompose en 2 familles : (*i*) les optimisations de type *cache aware* et (*ii*) les optimisations de type *cache oblivious*. Une méthode d'optimisation est dite *cache aware* si la taille des mémoires caches est prise en compte dans la construction de l'optimisation. La multiplication de matrices par blocage de boucle [WL91] est un exemple typique de l'esprit de cette famille d'optimisation. Une méthode d'optimisation qui est dite *cache oblivious* garantit une bonne localité des données sans tenir

compte de la taille des mémoires caches. Une optimisation de ce type pour la multiplication de matrice fut proposée dans [FLPR99].

Parmi les optimisations exposées précédemment, on peut identifier l'optimisation par blocage de boucle (section 4.5.2) comme étant une optimisation de type *cache aware* alors que l'optimisation par ré-agencement d'éléments (section 4.5.3) est plutôt de type *cache oblivious*. L'optimisation par blocage de boucle bénéficie des effets de cache en effectuant les calculs de gyromoyennes sur une partie restreinte du plan poloïdal. La taille du cache est un paramètre de cette méthode d'optimisation car plus elle est grande, plus la taille d'un carreau sur lequel les calculs sont réalisés peut être grande. De ce fait, cette méthode est dite de type *cache aware*. Ce n'est pas le cas de l'optimisation par ré-agencement d'éléments car dans cette approche, la taille du cache n'est pas un paramètre de la méthode d'optimisation. L'idée de l'optimisation par ré-agencement d'éléments est de minimiser la distance entre les éléments W_f qui participent aux différentes gyromoyennes selon un parcours donné du plan poloïdal. Puisque les caractéristiques matérielles n'interviennent pas dans le design de l'optimisation et que cette méthode accélère le calcul grâce aux effets de cache, cette méthode est dite de type *cache oblivious*.

Ces familles d'optimisation ont toutes deux pour but de réduire le temps d'exécution ; néanmoins elles se distinguent du point de vue performance et portabilité. Pour un problème donné, une implémentation optimisée de type *cache aware* peut se révéler plus rapide que son homologue de type *cache oblivious*. Ceci peut s'expliquer par le fait que la version *cache aware* peut être paramétrée par rapport au matériel sur lequel le programme s'exécute. C'est ce qui fait la force mais aussi la faiblesse de cette approche, car cela implique que pour obtenir les meilleures performances, des benchmarks doivent souvent être réalisés pour déterminer les paramètres de l'algorithme qui permettent d'obtenir les meilleurs performances sur une architecture donnée. À contrario, une optimisation de type *cache oblivious* s'affranchit des caractéristiques matérielles et garantie une amélioration des performances en moyenne sur plusieurs architectures.

La comparaison des avantages et des inconvénients entre ces deux domaines de recherche fait l'objet d'une grande quantité d'articles ([LFN02, BEF+14] par exemple). Dans le cadre de l'étude de la gyromoyenne, la comparaison de la performance des différentes implémentations fait l'objet du chapitre suivant.

Conclusion

Ce chapitre fournit une vue détaillée des méthodes de calcul par approximation de Padé et par interpolation d'Hermite de l'opérateur de gyromoyenne. Les différences entre ces deux méthodes sont profondes car la première s'appuie sur une interprétation dans l'espace de Fourier de la gyromoyenne, alors que la seconde méthode, plus directe, reste dans l'espace réel et se résume en grande partie à des calculs d'interpolation. Dans un second temps, une description formelle de type produit matrice–vecteur de l'opérateur de gyromoyenne par interpolation d'Hermite est donnée ainsi que le détail de différentes optimisations réalisées.

Les comparaisons de précision entre les deux méthodes de calcul sont traitées dans le chapitre suivant. De plus, l'opérateur de gyromoyenne est sollicité de nombreuses fois durant une itération temporelle de Gysela. Afin d'avoir une vision pertinente de l'impact des différentes implémentations de la gyromoyenne sur les performances de l'application Gysela, des tests sur différents types de simulations ont été réalisés. L'influence du choix de la méthode pour la gyromoyenne est aussi discuté.

Chapitre 5

Étude de l'opérateur de gyromoyenne à base d'interpolation d'Hermite

Sommaire

Ce chapitre expose les études qui ont été faites sur les aspects précision et performance des différentes méthodes de gyromoyenne, à savoir par approximation de Padé ou par interpolation d'Hermite. Ces études permettent de faire un bon compromis pour l'approche par interpolation d'Hermite.

Nous nous concentrons dans un premier temps sur l'étude de la précision de l'opérateur de gyromoyenne. Nous exhibons un type de fonctions pour lequel on peut calculer la gyromoyenne de façon analytique. Ce type de fonctions nous permet d'étudier de façon précise l'écart entre la gyromoyenne exacte et la gyromoyenne obtenue numériquement dans différentes configurations. Dans un second temps, nous comparons les temps d'exécution des différentes versions de l'opérateur de gyromoyenne intégrées à GYSELA sur de petites simulations. Finalement, afin de valider les résultats numériques obtenus par le nouvel opérateur de gyromoyenne, nous regardons les résultats obtenus sur un benchmark de référence de la communauté de physique des plasmas : le cas Cyclone DIII-D.

5.1 Définition d'un type de fonctions à gyromoyenne analytique

Dans cette partie, nous définissons un type de fonctions pour lequel on peut calculer l'opérateur de gyromoyenne de façon analytique. La proposition suivante donne l'expression analytique de la gyromoyenne des fonctions de type Fourier-Bessel.

Proposition

Soient $m \in \mathbb{N}$, $z \in \mathbb{C}$ et soit C_m une fonction de Bessel de première espèce (noté J_m) ou de seconde espèce (noté Y_m) (voir [Kre12]). Soit f une fonction en coordonnées polaires de $(r, \theta) \in \mathbb{R}^+ \times \mathbb{R} \mapsto \mathbb{C}$ définie par :

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

Pour un rayon de Larmor $\rho \in \mathbb{R}^+$ donné, la gyromoyenne de f en (r_0, θ_0) vaut :

$$\mathcal{J}_\rho(f)(r_0, \theta_0) = J_0(\rho)f(r_0, \theta_0).$$

Cette proposition a été établie par Steiner dans [SMC+15]. La démonstration donnée ici est une réécriture différente de celle qui a été proposée initialement.

Démonstration. Par définition (voir Equation (4.1)), la gyromoyenne est l'intégrale de la fonction f à 2 variables le long d'une courbe paramétrée [Tor09] :

$$\mathcal{J}_\rho(f)(\vec{x_0}) = \frac{1}{L} \oint_{\mathcal{C}} f(\vec{x})dl \tag{5.1}$$

où $\mathcal{C}$ désigne le cercle d'intégration, L le périmètre de ce cercle à savoir $2\pi\rho$ et dl l'élément de longueur de la courbe paramétrique $\mathcal{C}$. Le contour $\mathcal{C}$ est défini comme le cercle de centre $\vec{x_0}$ et de rayon ρ. Il se décrit par l'ensemble suivant :

$$\mathcal{C} = \{\vec{x}(\alpha) : \alpha \in [0, 2\pi]\}$$
$$\text{avec} \begin{cases} \vec{x}(\alpha) & = \vec{x_0} + \vec{\rho}(\alpha) \\ \vec{x_0} & = r_0(cos(\theta_0), sin(\theta_0)) \\ \vec{\rho}(\alpha) & = \rho(cos(\alpha), sin(\alpha)). \end{cases}$$

L'équation (5.1) se ré-exprime comme :

$$\mathcal{J}_\rho(f)(\vec{x_0}) = \frac{1}{L} \int_0^{2\pi} f(\vec{x}(\alpha)) \left\| \frac{d\vec{x}}{d\alpha}(\alpha) \right\|_2 d\alpha.$$

On calcule :

$$\frac{d\vec{x}}{d\alpha}(\alpha) = \frac{d\vec{\rho}}{d\alpha}(\alpha) = \rho(-sin(\alpha), cos(\alpha)) \tag{5.2}$$

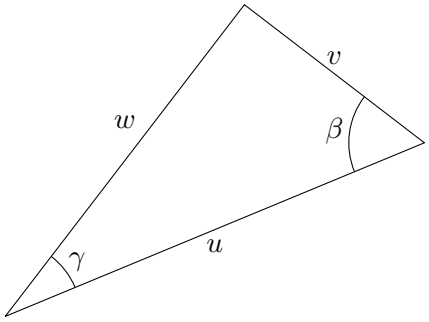
d'où :

$$\left\| \frac{d\vec{x}}{d\alpha}(\alpha) \right\|_2 = \sqrt{\rho^2((-sin(\alpha))^2 + (cos(\alpha))^2)} = \rho.$$

En utilisant cette dernière expression dans (5.2), on a :

$$\mathcal{J}_\rho(f)(\vec{x_0}) = \frac{1}{2\pi} \int_0^{2\pi} f(\vec{x}(\alpha))d\alpha. \tag{5.3}$$

Le théorème d'additivité de Graf pour les fonctions de Bessel (voir [AS+66]) affirme que dans le cas des nombres réels, si u, v et w sont les longueurs d'un triangle et β, γ les angles du triangle dans la configuration que montre la figure suivante :



alors pour tout $m \in \mathbb{N}$:

$$J_m(w)e^{im\gamma} = \sum_{k=-\infty}^{\infty} J_{m+k}(u)J_k(v)e^{ik\beta}.$$

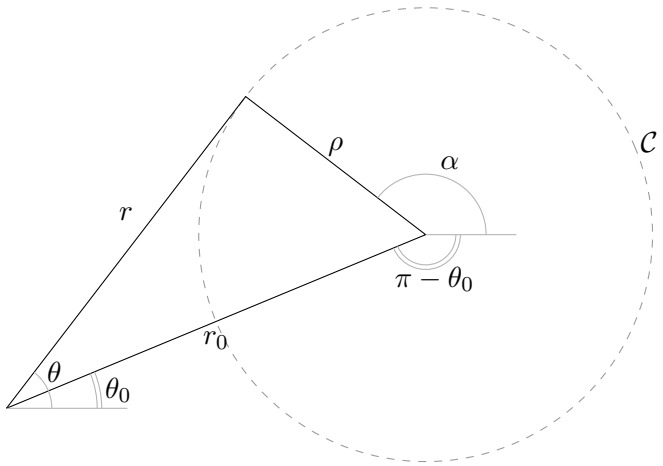
Dans le cas général où u, v, w, γ et β sont des complexes, la relation de Graf est la suivante :

$$C_m(w)e^{im\gamma} = \sum_{k=-\infty}^{\infty} C_{m+k}(u)J_k(v)e^{ik\beta}, \quad \text{si } |ve^{\pm i\gamma}| < |u|.$$

Soit $z \in \mathbb{C}$, on peut étendre la relation précédente en :

$$C_m(zw)e^{im\gamma} = \sum_{k=-\infty}^{\infty} C_{m+k}(zu)J_k(zv)e^{ik\beta}, \quad \text{si } |ve^{\pm i\gamma}| < |u|. \tag{5.4}$$

De façon similaire à la figure précédente, on peut représenter la relation entre les différents paramètres de l'opérateur de gyromoyenne grâce au graphique suivant :



Par identification des paramètres entre les deux figures, on a $u = r$, $v = \rho$, $w = r_0$, $\gamma = \theta - \theta_0$

et $\beta = \pi + \theta_0 - \alpha$. À partir des relations (5.3) et (5.4) on obtient :

$$\begin{aligned}\mathcal{J}_\rho(f)(\vec{x_0}) &= \frac{1}{2\pi}\int_0^{2\pi} f(\vec{x}(\alpha))d\alpha \\ &= \frac{e^{im\theta_0}}{2\pi}\int_0^{2\pi} C_m(zr(\alpha))e^{im(\theta(\alpha)-\theta_0)}d\alpha \\ &= \frac{e^{im\theta_0}}{2\pi}\int_0^{2\pi}\sum_{k=-\infty}^{\infty}\Big(C_{m+k}(zr_0)J_k(z\rho)e^{ik(\pi+\theta_0-\alpha)}\Big)d\alpha. \end{aligned} \tag{5.5}$$

Soit $K \in \mathbb{N}$, considérons la somme finie S_K définie comme suit :

$$S_K(\alpha) = \sum_{k=-K}^{K} C_{m+k}(zr_0)J_k(z\rho)e^{ik(\pi+\theta_0-\alpha)}d\alpha, \quad \text{avec } \alpha \in [0, 2\pi].$$

En considérant l'intégrale de S_K sur $[0, 2\pi]$:

$$\begin{aligned}\int_0^{2\pi} S_K(\alpha) &= \int_0^{2\pi}\sum_{k=-K}^{K} C_{m+k}(zr_0)J_k(z\rho)e^{ik(\pi+\theta_0-\alpha)}d\alpha \\ &= \sum_{k=-K}^{K} C_{m+k}(zr_0)J_k(z\rho)\int_0^{2\pi} e^{ik(\pi+\theta_0-\alpha)}d\alpha\end{aligned}$$

et en utilisant le fait que :

$$\int_0^{2\pi} e^{ik(\pi+\theta_0-\alpha)}d\alpha = 2\pi\delta_{k,0}.$$

on a, pour tout (K, α) :

$$\int_0^{2\pi} S_K(\alpha) = 2\pi C_m(zr_0)J_0(z\rho).$$

Par définition, en passant à la limite, on a :

$$\lim_{K\to\infty}\int_0^{2\pi} S_K(\alpha) = \int_0^{2\pi}\sum_{k=-\infty}^{\infty}\big(C_{m+k}(zr_0)J_k(z\rho)e^{ik(\pi+\theta_0-\alpha)}\big)d\alpha = 2\pi C_m(zr_0)J_0(z\rho)$$

En utilisant le résultat précédent dans (5.5), on peut conclure que :

$$\mathcal{J}_\rho(f)(r_0, \theta_0) = \frac{2\pi e^{im\theta_0}}{2\pi}C_m(zr_0)J_0(z\rho)$$

soit :

$$\boxed{\mathcal{J}_\rho(f)(r_0, \theta_0) = J_0(z\rho)f(r_0, \theta_0)}$$

□

À partir de ce résultat, nous sommes à même de construire des fonctions pour lesquelles nous connaissons l'expression analytique de leur gyromoyenne ; c'est le sujet de la section suivante.

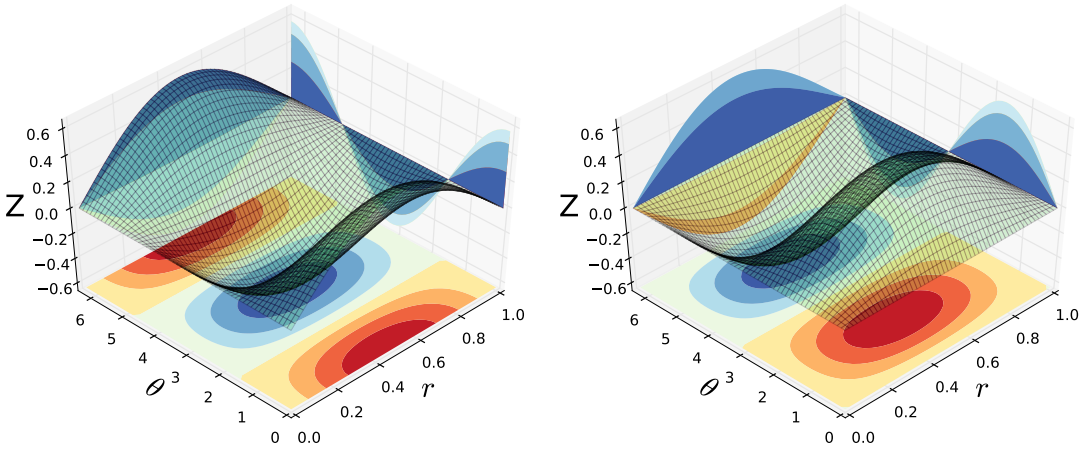


FIGURE 5.1 – Les parties réelle et imaginaire de la fonction test f_1 avec $m = 1$, $k = 1$, $r_{min} = 0$ et $r_{max} = 1$.

5.2 Exemples de fonctions pour différentes conditions limites

À partir du type de fonctions de base fournit par la section précédente, nous sommes à même de construire des fonctions dont on connait la gyromoyenne de façon analytique. Ce type de fonctions constitue la brique de base qui nous permet de construire 3 exemples de fonctions qui se résument à des combinaisons linéaires de fonctions appartenant à ce type. Des conditions limites différentes sont imposées pour chacune d'elles.

Les fonctions définies ci-dessous ont un domaine de définition borné non périodique dans la direction radiale. Nous discuterons dans la section suivante des différentes stratégies qui sont envisageables pour calculer en pratique la gyromoyenne basée sur une méthode d'interpolation pour les points dont le cercle de Larmor associé sort du domaine de définition de la fonction.

Exemple 1 $r_{\min} = 0$ *et avec une condition homogène de Dirichlet en* r_{max}.

Dans cette exemple, pour $m \in \mathbb{N}$, nous considérons la fonction f_1 définie sur le disque $[0, r_{\max}] \times [0, 2\pi]$ telle que :

$$f_1(r, \theta) = J_m \left(j_{m,k} \frac{r}{r_{\max}} \right) e^{im\theta}$$

où $j_{m,k}$ est la k^{ieme} racine de la fonction J_m sur $\mathbb{R}$. La fonction f_1 vérifie la condition de Dirichlet :

$$f_1(r_{\max}, \theta) = 0, \text{avec } 0 \leq \theta < 2\pi$$

et sa fonction gyromoyennée s'écrit :

$$\mathcal{J}_\rho(f_1)(r, \theta) = J_0 \left(j_{m,k} \frac{\rho}{r_{\max}} \right) f_1(r, \theta).$$

La Figure 5.1 donne l'allure des parties réelle et imaginaire de la fonction f_1 avec les paramètres $m = 1$, $k = 1$, $r_{min} = 0$ et $r_{max} = 1$.

Exemple 2 *Conditions de Dirichlet homogènes en* $r_{\min} > 0$ *et* $r_{\max}$.

Pour $m \in \mathbb{N}$, nous considérons la fonction f_2 définie sur l'anneau $[r_{\min}, r_{\max}] \times [0, 2\pi]$ telle que :

$$f_2(r,\theta) = \Big(J_m(\chi_{m,k}) Y_m\Big(\chi_{m,k}\frac{r}{r_{\max}}\Big) - Y_m(\chi_{m,k}) J_m\Big(\chi_{m,k}\frac{r}{r_{\max}}\Big)\Big) e^{im\theta}$$

où $\chi_{m,k}$ est la k^{ieme} racine de

$$y \mapsto J_m(y) Y_m\left(r_{\min}\frac{y}{r_{\max}}\right) - Y_m(y) J_m\left(r_{\min}\frac{y}{r_{\max}}\right)$$

sur $\mathbb{R}$.

La fonction f_2 vérifie les conditions de Dirichlet suivantes :

$$f_2(r_{\min},\theta) = 0, \quad f_2(r_{\max},\theta) = 0,\ 0 \le \theta < 2\pi$$

et sa fonction gyromoyennée s'écrit :

$$\mathcal{J}_\rho(f_2)(r,\theta) = J_0\left(\chi_{m,k}\frac{\rho}{r_{\max}}\right) f_2(r,\theta). \tag{5.6}$$

Exemple 3 *Conditions de Neumann homogènes en* $r_{\min} > 0$ *et* $r_{\max}$.

Pour $m \in \mathbb{N}$, nous considérons la fonction f_3 définie sur l'anneau $[r_{\min}, r_{\max}] \times [0, 2\pi]$ telle que :

$$f_3(r,\theta) = \Big(J'_m(\eta_{m,k}) Y_m\Big(\eta_{m,k}\frac{r}{r_{\max}}\Big) - Y'_m(\eta_{m,k}) J_m\Big(\eta_{m,k}\frac{r}{r_{\max}}\Big)\Big) e^{im\theta}$$

où $\eta_{m,k}$ est la k^{ieme} racine de

$$y \mapsto J'_m(y) Y'_m\Big(r_{\min}\frac{y}{r_{\max}}\Big) - Y'_m(y) J'_m\Big(r_{\min}\frac{y}{r_{\max}}\Big)$$

sur $\mathbb{R}$.

La fonction f_3 vérifie les conditions de Neumann suivantes :

$$\partial_r f_3(r_{\min},\theta) = 0, \quad \partial_r f_3(r_{\max},\theta) = 0,\ 0 \le \theta < 2\pi.$$

En r_{min}, ce résultat se justifie par le fait que :

$$\partial_r f_3(r,\theta) = \frac{\eta_{m,k}}{r_{\max}} \Big(J'_m(\eta_{m,k}) Y'_m\Big(\eta_{m,k}\frac{r}{r_{\max}}\Big) - Y'_m(\eta_{m,k}) J'_m\Big(\eta_{m,k}\frac{r}{r_{\max}}\Big)\Big) e^{im\theta}.$$

Par définition de $\eta_{m,k}$:

$$\partial_r f_3(r_{\min},\theta) = 0.$$

Sa fonction gyromoyennée s'écrit :

$$\mathcal{J}_\rho(f_3)(r,\theta) = J_0\Big(\eta_{m,k}\frac{\rho}{r_{\max}}\Big) f_3(r,\theta).$$

La construction de ces différents exemples nous montre premièrement qu'il est possible de fixer des contraintes et de fabriquer des fonctions pour lesquelles on sait calculer la gyromoyenne de façon analytique. Deuxièmement, ces exemples peuvent servir de base pour la construction de programme test pour l'étude des erreurs de la version numérique de l'opérateur de gyromoyenne. La connaissance sous forme analytique de la fonction gyromoyennée permet d'étudier l'écart entre l'évaluation numérique de la gyromoyenne et sa valeur exacte ; c'est le sujet de la section suivante.

5.3 Précision de l'opérateur de gyromoyenne

La précision de l'opérateur de gyromoyenne basé sur l'interpolation d'Hermite dépend principalement de 3 paramètres :

— le nombre de cellules du maillage $N_r \times N_\theta$;
— la taille du rayon de Larmor ρ ;
— le nombre de points d'intégration N.

Intuitivement, on s'attend à ce que le calcul de la gyromoyenne soit d'autant plus précis que le maillage est raffiné ($N_r \times N_\theta$ grand), que le contour d'intégration est petit (ρ petit) ou que le nombre de points d'interpolation sur le cercle d'intégration est grand (N grand).

Nous avons créé un code de test basé sur le second type de fonction f_2 fourni dans la section 5.2 pour étudier la précision de l'opérateur de gyromoyenne basée sur l'interpolation d'Hermite. Nous avons fait ce choix car ce type de fonction correspond au cadre d'utilisation de GYSELA, c'est-à-dire $r_{min} > 0$ et des conditions de Dirichlet aux extrémités du domaine radial. Pour toutes les études qui sont exposées dans cette partie, nous considérons uniquement la partie réelle de la fonction f_2.

Les différentes études réalisées se basent sur un même programme de test. Ce programme réalise les opérations suivantes :

1. lecture des paramètres d'entrés :
   N_r : le nombre de points dans les deux directions du maillage ($N_r = N_\theta$) ;
   ρ : le rayon de Larmor ;
   N : le nombre de points d'intégration ;
   m : l'ordre des fonctions de Bessel utilisé ;
   k : l'indice de la racine utilisée,
2. initialisation du plan poloïdal par la fonction f_2 ;
3. calcul numérique de la gyromoyenne ;
4. affichage de diverses grandeurs fin d'évaluer la précision des calculs.

Pour les points aux extrémités radiales du domaine de définition de la fonction f_2, des points d'interpolation peuvent sortir du domaine de définition de la fonction. Pour gérer ces conditions limites, on peut appliquer une des stratégies de la liste non exhaustive suivante :

1. on considère que la fonction est nulle à l'extérieur de son domaine de définition ;
2. on considère la projection radiale du point d'interpolation en r_{min} ou r_{max} suivant le cas ;
3. on ne considère pas ces points dans le calcul de gyromoyenne. Par exemple, si $N = 4$ et un des points d'interpolation est à l'extérieur du domaine de définition, alors on calcule uniquement l'interpolation des 3 points dans le domaine de définition et on en fait la moyenne.

Dans le cas du calcul de la gyromoyenne basée sur l'interpolation d'Hermite, nous avons choisi d'utiliser la seconde stratégie, à savoir la projection radiale (voir section 4.3.1).

Afin d'avoir une vision d'ensemble du comportement de l'erreur de l'opérateur de gyromoyenne basé sur l'interpolation d'Hermite, 3 études ont été réalisées avec le programme test. Chaque étude se distingue par les paramètres d'entrées utilisés. Pour chaque étude, un sous-ensemble de ces paramètres est fixé.

5.3.1 $1^{ère}$ étude : l'influence de ρ

Dans cette étude, les paramètres fixés sont les suivants :

$$\boxed{N = 1024,\ m = 1 \text{ et } k = 1}$$

Nous nous intéressons ici à l'influence du rayon du cercle d'intégration ρ sur la précision de la gyromoyenne basée sur les interpolation d'Hermite en fonction de la résolution du maillage $N_r \times N_\theta$.

Pour chaque couple (rayon d'intégration – résolution), on regarde la valeur de la norme L^2 de la différence entre la gyromoyenne numérique, $\tilde{\mathcal{J}}_\rho$, et la gyromoyenne analytique, $\mathcal{J}_\rho$. Seuls sont considérés les points du plan dont le cercle de Larmor associé est contenu dans le plan, c'est-à-dire les points qui ne sont pas trop proches des extrémités de la direction radiale. Ces parties du plan qui ne sont pas considérées dans notre étude sont appelées *zones buffer*.

Soit $\tilde{P}$ l'ensemble des points du plan poloïdal hors zones buffer. Soit $\mathcal{E}$ la grandeur qui nous permet de quantifier l'écart entre la solution exacte et la solution numérique. Elle se calcule de la façon suivante :

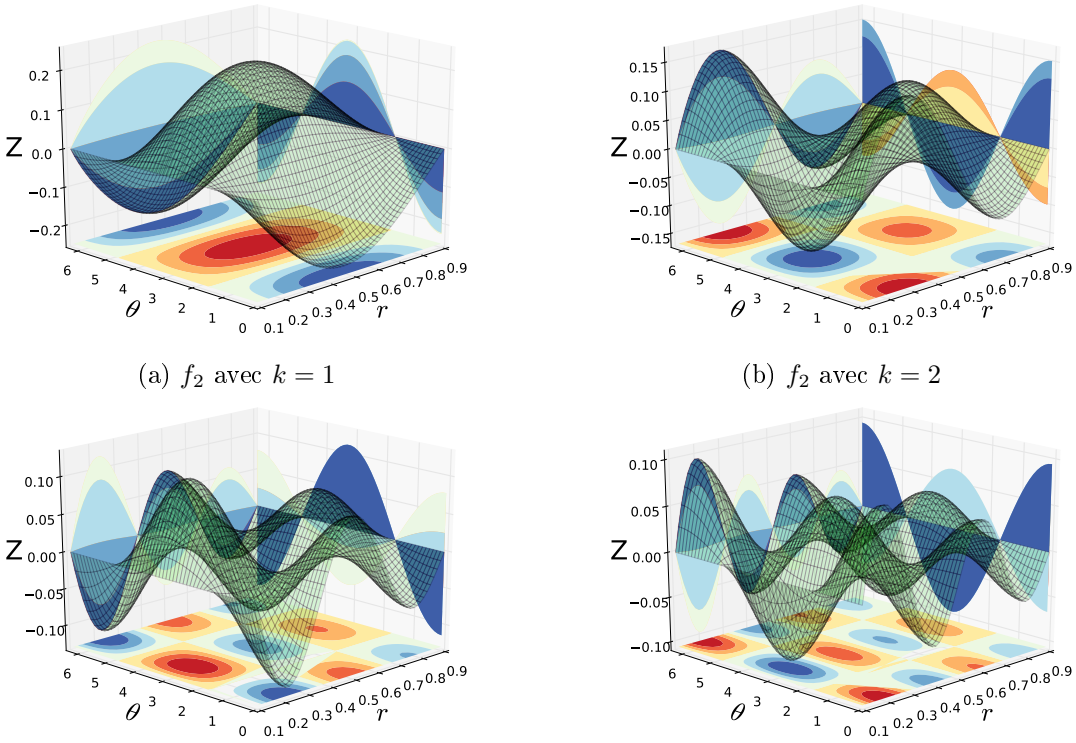
$$\begin{aligned}\mathcal{E} &= \left(\sum_{\vec{x}\in\tilde{P}} \left(\tilde{\mathcal{J}}_\rho(f_2)(\vec{x}) - \mathcal{J}_\rho(f_2)(\vec{x}) \right)^2 \right)^{1/2} \\ &= \left(\sum_{\vec{x}\in\tilde{P}} \left(\tilde{\mathcal{J}}_\rho(f_2)(\vec{x}) - J_0\big(\chi_{m,k}\frac{\rho}{r_{\max}}\big) f_2(\vec{x}) \right)^2 \right)^{1/2}.\end{aligned}$$

Le Tableau 5.1 montre la valeur de $\mathcal{E}$ pour différentes tailles de maillage et différentes valeurs de ρ.

$N_r = N_\theta$	16	25	40	64
$\rho = 1.10^{-3}$	$2.52 \cdot 10^{-3}$	$8.90 \cdot 10^{-4}$	$2.89 \cdot 10^{-4}$	$9.20 \cdot 10^{-5}$
$\rho = 5.10^{-3}$	$2.52 \cdot 10^{-3}$	$8.86 \cdot 10^{-4}$	$2.87 \cdot 10^{-4}$	$9.36 \cdot 10^{-5}$
$\rho = 1.10^{-2}$	$2.51 \cdot 10^{-3}$	$8.78 \cdot 10^{-4}$	$2.88 \cdot 10^{-4}$	$9.87 \cdot 10^{-5}$
$\rho = 5.10^{-2}$	$2.24 \cdot 10^{-3}$	$7.72 \cdot 10^{-4}$	$2.66 \cdot 10^{-4}$	$1.20 \cdot 10^{-4}$
$\rho = 1.10^{-1}$	$2.14 \cdot 10^{-3}$	$9.32 \cdot 10^{-4}$	$4.63 \cdot 10^{-4}$	$2.51 \cdot 10^{-4}$

TABLE 5.1 – Comparaison de $\mathcal{E}$ pour différentes tailles de maillage et différentes valeurs de ρ avec la gyromoyenne basée sur l'interpolation d'Hermite. Paramètres : $r_{\min} = 0.1$, $r_{\max} = 0.9$, $N = 1024$, $m = 1$ et $k = 1$.

On constate que pour un maillage grossier, la valeur du paramètre ρ n'affecte pas beaucoup la précision du calcul. Dans notre cas avec $N_r = N_\theta = 16$, l'ordre de l'erreur $\mathcal{E}$ est de 10^{-3} quel que soit ρ. La discrétisation grossière pénalise clairement la précision du calcul.



(a) f_2 avec $k = 1$

(b) f_2 avec $k = 2$

(c) f_2 avec $k = 3$

(d) f_2 avec $k = 4$

FIGURE 5.2 – Illustration de la fonction d'entrée f_2 pour $k \in 1, 2, 3, 4$.

Dans le cas $N_r = N_\theta = 64$, plus ρ est grand, moins le calcul est précis. Cette tendance se confirme sur de plus grands maillages. Cette tendance s'inscrit dans le cadre d'utilisation de GYSELA car les maillages poloïdaux utilisés par les simulations sont toujours de taille supérieure à 64×64.

5.3.2 $2^{ième}$ étude : l'influence de N

Dans cette étude, les paramètres fixés sont les suivants :

$$\boxed{N_r \times N_\theta = 128 \times 128,\ \rho = 0.1 \text{ et } m = 1}$$

Nous nous intéressons ici à l'influence du nombre de points d'interpolation N sur la précision de la gyromoyenne basée sur l'interpolation d'Hermite.

Les expériences réalisées dans cette partie se basent sur une réinterprétation de la formule (5.6). Cette formule relie analytiquement la fonction f_2 et sa gyromoyenne. Dans le cas où la gyromoyenne est évaluée numériquement, on peut réécrire cette formule comme suit :

$$\frac{\tilde{\mathcal{J}}_\rho(f_2)(r_0, \theta_0)}{f_2(r_0, \theta_0)} \approx J_0\big(\chi_{m,k} \frac{\rho}{r_{\max}}\big). \tag{5.7}$$

Sous cette forme, le calcul numérique de la gyromoyenne nous donne une approximation de la fonction de Bessel J_0.

L'étude réalisée dans cette partie a pour but d'estimer la qualité de la reconstruction de cette fonction de Bessel. Le protocole de l'expérience est le suivant. Pour N et k fixés, on calcule la

valeur $J_0\left(\chi_{m,k}\frac{\rho}{r_{\max}}\right)$ pour l'ensemble des points appartenant à $\tilde{P}$ et enfin on calcule la moyenne de l'ensemble de ces valeurs.

Pour une valeur de k donnée, on calcule alors l'approximation de J_0 en un point, c'est-à-dire pour un antécédent donné. Pour obtenir plusieurs points de la courbe de J_0, on réitère l'expérience précédente pour différentes valeurs de k. On peut noter que les racines $\chi_{m,k}$ ont un comportement quasi-linéaire par rapport à l'indice k, ce qui est un avantage dans notre étude car cela signifie que les abscisses pour lesquelles J_0 est estimée sont réparties de façon uniforme. Un autre point important à noter est que plus la valeur de k est élevée, plus les structures spatiales de la fonction d'entrée f_2 sont fines. L'ensemble de Figures 5.2 donne l'allure de la fonction f_2 pour $k \in \{1, 2, 3, 4\}$.

Pour construire une courbe à N fixé pour $x \in [0, 15]$, l'indice k parcourt l'ensemble $[\![1, 39]\!]$. Plus la valeur en abscisse est grande, plus la fonction en entrée admet de variations dans la direction radiale sur son domaine de définition qui est lui de taille constante. Le tracé de plusieurs courbes pour différentes valeurs de N nous permet de nous rendre compte de l'influence de ce paramètre sur la précision de l'opérateur de gyromoyenne.

L'ensemble de Figures 5.3 nous permet d'évaluer la qualité de la reconstruction de la fonction J_0 en fonction de N. L'abscisse x représente le produit $k_\perp\rho$ dans l'application Gysela. Sur chacune des figures, on retrouve la courbe de la solution exacte J_0 en bleu, la courbe de l'approximation de Padé en vert et la courbe discrétisée de l'approximation obtenue par calcul de la gyromoyenne par interpolation d'Hermite en rouge. Ces figures permettent de se rendre compte de la convergence de la méthode en fonction du nombre de points d'intégration N.

On peut voir que pour de faibles valeurs de N, à partir d'une abscisse donnée, l'approximation de la fonction de Bessel obtenue est assez éloignée de la valeur escomptée. Ces écarts signifient que l'opérateur de gyromoyenne est précis pour des fonctions d'entrée dont les variations spatiales sont assez lentes pour être capturées par N points d'interpolation. Pour des fonctions avec des variations spatiales plus rapides, la gyromoyenne calculée en un point sera en moyenne éloignée de la valeur attendue.

Comme on peut s'y attendre, plus N est grand, meilleure est la qualité de la reconstruction. On peut noter que la convergence de la reconstruction est rapide par rapport au nombre de points d'interpolation utilisés. Pour $N = 16$ points d'interpolation, l'approximation de la fonction J_0 sur l'intervalle $[0, 12]$ est très bonne par rapport à l'approximation de Padé.

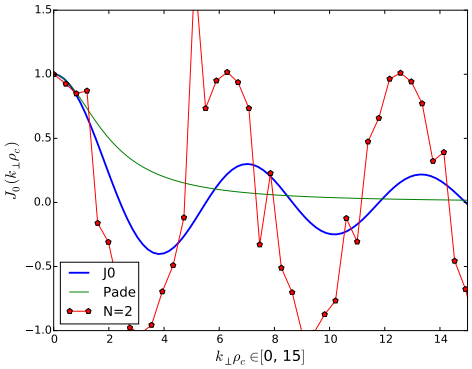
Dans le cadre de l'application Gysela, la précision de la gyromoyenne sur l'intervalle $[0, 1]$ est cruciale car cet intervalle contient les petites longueurs d'onde de la fonction de distribution. Ces longueurs d'onde jouent un rôle important dans le développement des instabilités simulées par Gysela. Si $x \in [0, 1]$ représente l'intervalle d'intérêt pour les effets physiques simulés, l'opérateur de gyromoyenne basé sur Hermite avec $N = 4$ points d'interpolation apporte une précision bien supérieure à l'approximation de Padé. Si $x \in [0, 8]$ représente l'intervalle d'intérêt, dans le cas par exemple où des longueurs d'onde plus grandes ont des effets significatifs sur le phénomène physique simulé, $N = 12$ points d'interpolation seront nécessaires pour assurer une bonne précision de la gyromoyenne.

5.3.3 $3^{ième}$ étude : l'influence de la résolution du maillage

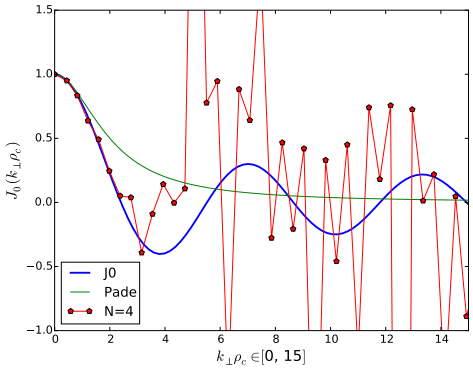
Dans cette étude, les paramètres fixés sont les suivants :

$$\boxed{\rho = 0.1,\ m = 1 \text{ et } k = 51}$$

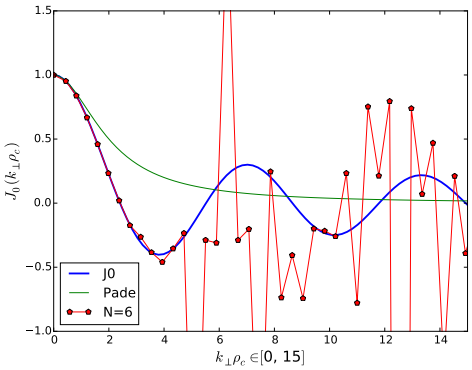
La grande valeur du paramètre $k = 51$ sort du cadre d'utilisation standard de l'opérateur de gyromoyenne dans Gysela puisque l'abscisse $k_\perp\rho$ correspondant à cette valeur vaut $k_\perp\rho \approx 20$.



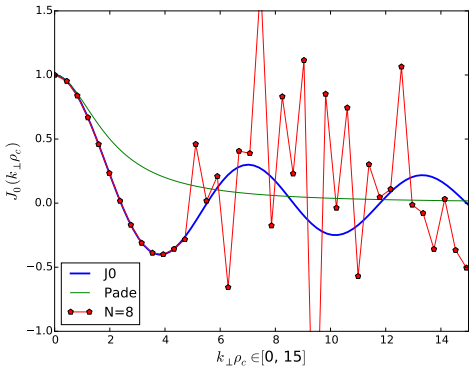
(a) Hermite avec $N = 2$



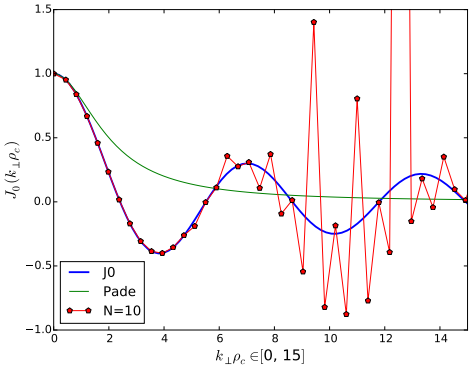
(b) Hermite avec $N = 4$



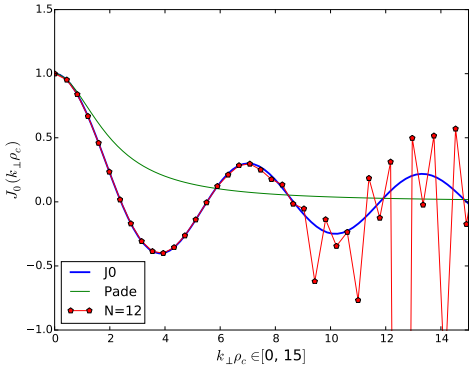
(c) Hermite avec $N = 6$



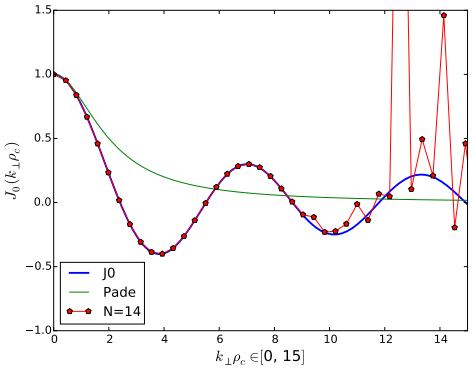
(d) Hermite avec $N = 8$



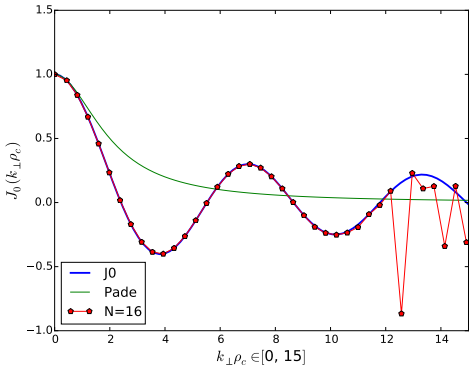
(e) Hermite avec $N = 10$



(f) Hermite avec $N = 12$



(g) Hermite avec $N = 14$



(h) Hermite avec $N = 16$

FIGURE 5.3 – Comparaison de la gyromoyenne basée sur l'approximation de Padé et sur l'interpolation d'Hermite. Convergence de l'opérateur en fonction du nombre de points d'intégration. Paramètres : $m = 1$, $N_r = N_\theta = 128$, $r_{min} = 0.1$, $r_{max} = 0.9$ et $\rho = 0.1$.

Cette grande valeur de k sous-entend un grand nombre de variations dans la direction radiale de la fonction d'entrée. Le choix d'une si grande valeur pour k a été pris pour pouvoir mettre en évidence l'influence de la résolution du maillage sur la précision de la gyromoyenne.

Cette étude est le prolongement de la deuxième étude. Les définitions de l'ensemble de points $\tilde{P}$ et de l'erreur $\mathcal{E}$ sont identiques à la première étude 5.3.1. La deuxième étude nous a montré que pour de grandes valeurs de k, la reconstruction de la fonction de Bessel pour l'ensemble des points de $\tilde{P}$ est bonne si on prend un nombre suffisant de points d'interpolation.

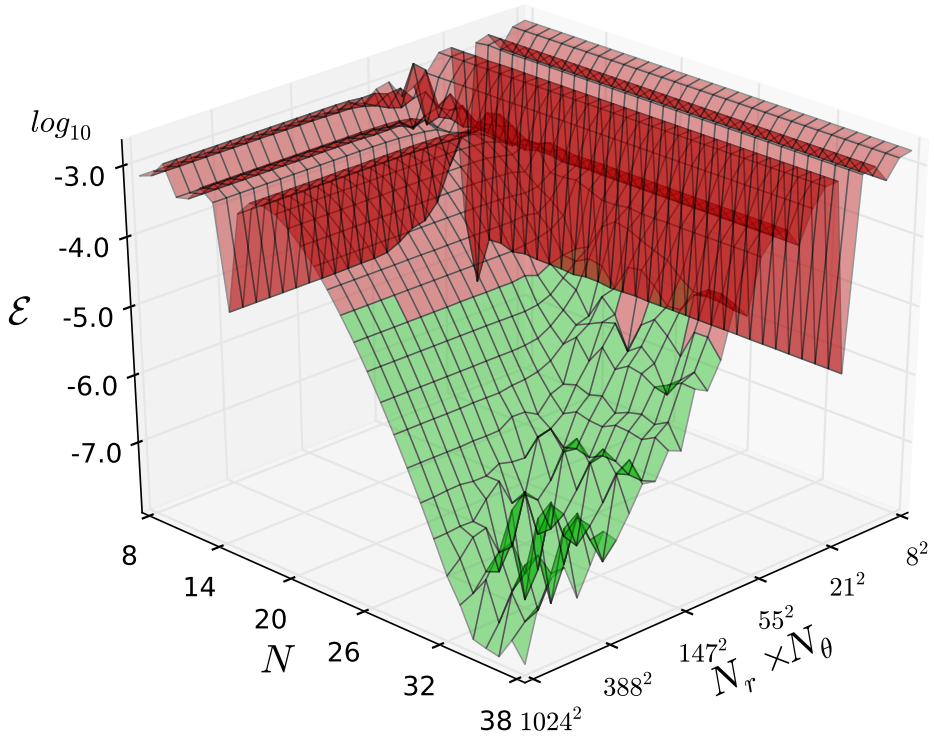


FIGURE 5.4 – Etude de l'erreur $\mathcal{E}$ de la gyromoyenne basée sur l'interpolation d'Hermite en échelle logarithmique à base 10 en fonction du nombre de points d'interpolation N et de la résolution du maillage. La fonction d'entrée est paramétrée avec $m = 1$ et $k = 51$. Cette valeur élevée en k indique la présence de très fines structures spatiales de la fonction f_2.

Sur la Figure 5.4 nous avons tracé l'erreur $\mathcal{E}$ en échelle logarithmique à base 10 en fonction du nombre de points d'interpolation N et de la résolution du maillage. On s'attend naturellement à ce que l'erreur diminue lorsque la résolution augmente, ce qui est effectivement vérifié. Pour faciliter la lecture du graphique, les cellules vertes indiquent que la valeur de l'erreur aux 4 coins de la cellule sont en-dessous du seuil 10^{-4}. Ce seuil a été fixé de façon arbitraire.
Ce graphique met en évidence différentes tendances concernant la précision de l'opérateur. On peut identifier deux régions du graphique où l'erreur semble ne pas diminuer. Lorsqu'on regarde cette figure pour de faibles valeurs de N, on se rend compte que la variation de la résolution n'a pas d'influence sur la précision. De façon symétrique, le même raisonnement s'applique lorsqu'on regarde la figure pour de faibles résolutions $N_r \times N_\theta$.

En revanche, pour $N = 38$, l'impact de la résolution sur l'erreur $\mathcal{E}$ se voit nettement. Il en est de même lorsqu'on regarde la courbe obtenue en fixant la résolution à $N_r \times N_\theta = 1024^2$. L'idée

intuitive de la convergence de l'opérateur en fonction de N et de $N_r \times N_\theta$ est donc bien vérifiée.

La Figure 5.4 met aussi en avant le fait que la convergence de l'erreur $\mathcal{E}$ s'obtient avec un nombre raisonnable de points d'interpolation ($\max(N) = 38$) alors qu'un ensemble de valeurs bien plus grand doit être considéré pour observer le même comportement en fonction du pas d'espace ($\max(N_r \times N_\theta) = 1024^2$).

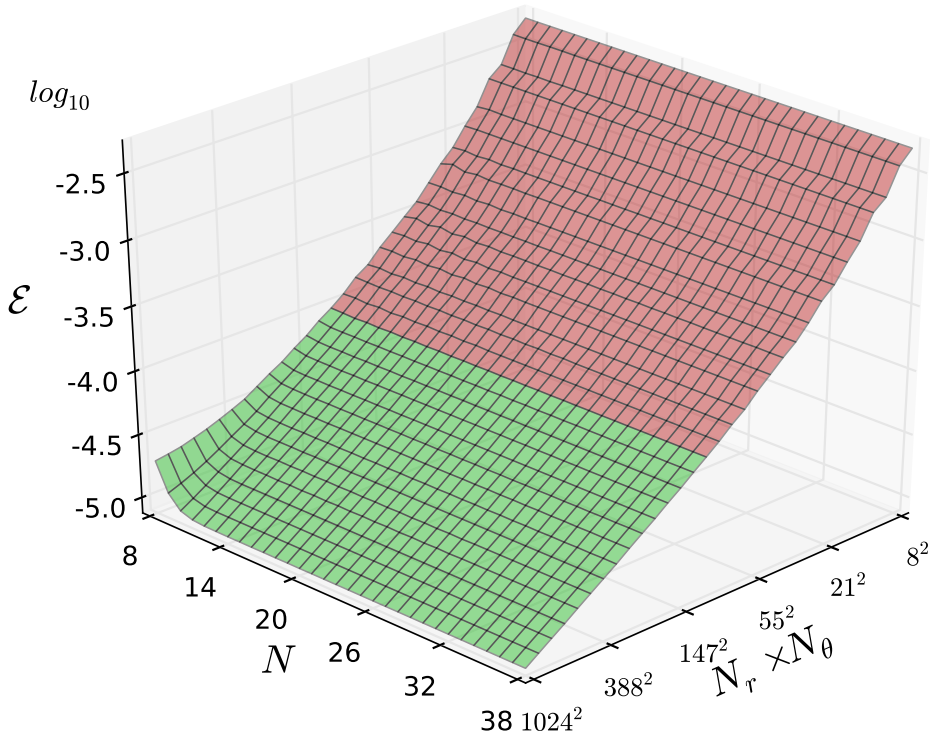


FIGURE 5.5 – Etude de l'erreur $\mathcal{E}$ de la gyromoyenne basée sur l'interpolation d'Hermite en échelle logarithme à base 10 en fonction du nombre de points d'interpolation N et de la résolution du maillage. La fonction d'entrée est paramétrée avec $m = 1$ et $k = 2$. Cette faible valeur de k signifie que la fonction d'entrée f_2 n'admet pas un grand nombre de variations dans sa direction radiale.

En complément de cette étude, on peut reproduire la Figure 5.4 avec des paramètres moins exigeants :

$$\boxed{\rho = 0.1,\ m = 1 \text{ et } k = 2}$$

Sur la Figure 5.5 on peut donc voir l'évolution de l'erreur $\mathcal{E}$ en fonction de la résolution du maillage et du nombre de points d'interpolation N pour cette nouvelle fonction f_2. On constate que l'erreur ne dépend quasiment pas du nombre de points d'interpolation mais entièrement de la finesse du maillage.

5.3.4 Bilan des études sur la précision de la gyromoyenne

La seconde étude (voir section 5.3.2) permet de comparer aisément la gyromoyenne basée sur l'approximation de Padé et celle basée sur l'interpolation d'Hermite. On voit clairement que l'implémentation basée sur l'interpolation est plus précise que celle basé sur l'approximation de

Padé. Ce gain de précision est accompagné d'une augmentation du temps d'exécution comme on le verra dans la section suivante.

Afin que les simulations qui utilisent l'opérateur de gyromoyenne basé sur l'interpolation d'Hermite ne soient pas pénalisées d'un point de vue performance, différentes optimisations ont été réalisées (voir section 4.5). Des comparaisons de performance des différentes versions de l'opérateur de gyromoyenne sont faites dans la section suivante.

5.4 Comparaison des temps d'exécution : petits cas

Plusieurs simulations ont été réalisées pour comparer les performances des différentes versions de l'opérateur de gyromoyenne et l'efficacité des différentes optimisations. Dans le cadre des benchmarks présentés ici, les simulations ont été réduites à uniquement 4 itérations pour garder un temps d'exécution raisonnable. Ces tests ont été réalisés sur la machine HELIOS [16]. Les nœuds de calcul de cette machine sont équipés de 2 processeurs Intel Xeon E5-2450 2.10GHz, soit 16 cœurs par nœud. Sur le Tableau 5.2, les différents cas tests ont été réalisés sur les maillages suivants :

$$N_r = 256,\ N_\theta = 256,\ N_\varphi = 32,\ N_{v_\parallel} = 16,\ N_\mu = 4 \tag{5.8}$$

$$N_r = 512,\ N_\theta = 512,\ N_\varphi = 32,\ N_{v_\parallel} = 16,\ N_\mu = 4. \tag{5.9}$$

Pour mettre en évidence les différences de performance entre les différentes implémentations, la taille du plan poloïdal est multipliée par 4 entre les deux maillages. Nous utilisons $N = 16$ points d'interpolation pour les gyromoyennes basées sur l'interpolation d'Hermite. Ce nombre de points assure une bonne précision des résultats comme on a pu le constater dans la section précédente. Les fluctuations des temps d'exécution dues à l'utilisation de ressources partagées (telles que le réseau inter-nœud et le système de fichiers parallèle) sont minimisées. Nous avons configuré les simulations de sorte à ce qu'elles s'exécutent sur un seul nœud de calcul afin d'éviter l'utilisation du réseau. Afin de réduire au minimum l'utilisation du système de fichiers parallèle, nous avons désactivé l'écriture des fichiers de résultats.

Nous avons utilisé 16 processus MPI et 1 thread par processus pour ces simulations. Cette configuration n'est pas représentative de celles utilisées en production mais sont pratiques dans le cadre d'une étude de performance. Cette configuration permet d'être sûr que la simulation ne souffre pas d'effets NUMA (Non Uniform Memory Access). D'autre part, cette configuration ne profite pas de la parallélisation OPENMP pour minimiser l'empreinte mémoire de la simulation. Des configurations plus réalistes sont utilisées dans les tests de la section 5.5.

Le Tableau 5.2 donne les temps d'exécution de GYSELA utilisant l'opérateur de gyromoyenne initial par approximation de `Padé` ainsi que les autres versions basées sur l'interpolation d'Hermite. Ces dernières se décrivent succinctement de la façon suivante :

- `initiale` : implémentation initiale basée sur l'interpolation d'Hermite ;
- `compacte` : optimisation qui utilise des structures de données compactes (voir section 4.5.1) ;
- `ré-arrangement` : optimisation qui réorganise les éléments d'une structure utilisée lors du produit scalaire creux (voir section 4.5.3) ;
- `blocage` : optimisation qui calcule la gyromoyenne du plan poloïdal en le parcourant carreau par carreau (voir section 4.5.2).

Il est possible d'effectuer des simulations en désactivant l'opérateur de gyromoyenne. Cette configuration correspond à la colonne «`Désactivé`» du Tableau 5.2. Elle ne donne pas de résultats

16. IFERC-CSC HELIOS super calculateur à Rokkasho – Japan : http ://www.top500.org/system/177449

correctes, bien sûr, mais elle est très utile pour quantifier le coût réel associé à l'opérateur de gyromoyenne durant une simulation.

Comme on peut s'y attendre, le temps de calcul augmente avec la taille du plan poloïdal. L'optimisation `compacte` qui est une étape intermédiaire dans le processus d'optimisation améliore déjà sensiblement le temps d'exécution par rapport à l'implémentation `initiale`. Néanmoins, le pourcentage du temps d'exécution associé à l'implémentation `compacte` est 6 à 10 fois supérieur à l'implémentation `Padé`. L'implémentation `ré-arrangement` montre de meilleurs résultats par rapport à l'étape intermédiaire. Sur le plus grand maillage, cette optimisation réduit le temps d'exécution d'environ 16 % par rapport à la version `compacte`. Ces chiffres démontrent l'impact de l'organisation des données en mémoire sur les performances d'un programme.

Les meilleures performances sont obtenues par l'optimisation `blocage` qui réduit le coût cumulé de l'opérateur de gyromoyenne de 40% par rapport à la version `initiale`. Néanmoins, cette implémentation est toujours approximativement 4 fois plus coûteuse que l'implémentation basée sur l'approximation de Padé sur le plus grand maillage. Ceci est en grande partie dû au nombre de points d'interpolation $N = 16$ utilisés pour ces simulations.

Les temps que montre le Tableau 5.3 ont été obtenus avec $N = 8$ points d'interpolation pour les mêmes simulations que dans le Tableau 5.2. En comparant ces deux tableaux, on voit clairement l'incidence du nombre de points d'interpolation sur le temps d'exécution. Avec deux fois moins de points d'interpolation, le temps d'exécution associé à la version `blocage` est réduit à 17.8 %, ce qui contribue à réduire son écart avec la version `Padé` qui représente 7.1 % du temps d'exécution total.

Le contrecoup de la diminution du nombre de point d'interpolation est la dégradation de la précision de l'opérateur de gyromoyenne. Le compromis entre précision et coût de calcul de l'opérateur de gyromoyenne utilisé dans GYSELA est discuté dans la section suivante.

5.5 Validation numérique : le cas test Cyclone

La gyromoyenne constitue un opérateur essentiel dans la théorie gyrocinétique. Afin de vérifier la validité de l'opérateur de gyromoyenne basé sur l'interpolation d'Hermite, nous avons reproduit des simulations du cas "Cyclone DIII-D".

Ce cas physique sert de support pour la comparaison de différents codes de simulation [Da00]. Il permet de confronter les grandeurs numériques d'un code de simulation aux grandeurs physiques mesurées lors d'une expérience du tokamak Doublet III-D, alias DIII-D. Ces

		Méthode de l'opérateur gyromoyenne					
		`Padé`	Hermite				`Désactivé`
			`Initiale`	`Compacte`	`Ré-arrangement`	`Blocage`	
Maillage (5.8) :	Exéc.	110,20	150,60	146,35	132,91	131,39	`106,18`
	%	3.8 %	41.8 %	37.8 %	25.2 %	23.7 %	–
Maillage (5.9) :	Exéc.	464,62	629,46	627,80	557,49	550,38	`433,35`
	%	7.2 %	45.3 %	44.9 %	28.6 %	27.0 %	–

TABLE 5.2 – Les lignes «Exéc» donnent le temps d'exécution de GYSELA en seconde avec les différentes versions de la gyromoyenne et les lignes «%» donnent le pourcentage du temps d'exécution associé à l'opérateur de gyromoyenne utilisé. Ces temps sont donnés pour 2 tailles différentes du plan poloïdal avec $N = 16$ points d'interpolation.

		Méthode de l'opérateur gyromoyenne					
		Padé	Hermite				Désactivé
			Initiale	Compacte	Ré-arrangement	Blocage	
Maillage (5.8) :	Exéc.	110,00	131,42	129,11	123,12	120,47	105,60
	%	4.2 %	24.4 %	22.3 %	16.6 %	14.1 %	–
Maillage (5.9) :	Exéc.	464,20	544,07	538,38	525,21	510,44	433,49
	%	7.1 %	25.5 %	24.2 %	21.2 %	17.8 %	–

TABLE 5.3 – Les lignes «Exéc» donnent le temps d'exécution de GYSELA en secondes avec les différentes versions de la gyromoyenne et les lignes «%» donnent le pourcentage du temps d'exécution associé à l'opérateur de gyromoyenne utilisé. Ces temps sont donnés pour 2 différentes tailles du plan poloïdal avec $N = 8$ points d'interpolation.

comparaisons donnent un point de repère qui permet de valider les résultats obtenus numériquement. Ce cas est communément appelé "Cyclone DIII-D base case". Les détails du jeu de paramètres que nous avons utilisé sont disponibles à la section 4 de l'article [GSG+08].

Les résultats que nous avons obtenus sur le cas Cyclone correspondent à ceux obtenus précédemment dans [GSG+08]. La comparaison des résultats obtenus avec l'opérateur de gyromoyenne par interpolation d'Hermite avec ceux obtenus précédemment nous permet de le valider et de voir son influence sur le comportement global des simulations.

Nous poursuivons la comparaison d'un point de vue performance entre les différentes versions de gyromoyenne dans un premier temps. Les simulations effectuées dans cette partie utilisent plusieurs nœuds de calcul. Elles utilisent donc des ressources partagées qui peuvent produire du bruit sur les temps d'exécution mesurés. Nous avons exécuté ces simulations plusieurs fois pour nous assurer que les temps d'exécution obtenus sont reproductibles. Nous nous concentrons sur le mode le plus instable $(m, n) = (14, -10)$ avec les paramètre suivants : $\mu_{min} = 0.143$, $\mu_{max} = 7$ (ces paramètres sont décrits dans [SMC+15], p. 10).

Le maillage utilisé est le suivant :

$$N_r = 256,\ N_\theta = 256,\ N_\varphi = 64,\ N_{v_\parallel} = 48,\ N_\mu = 8. \tag{5.10}$$

Un processus de vérification standard des résultats numériques consiste à extraire le taux de croissance des modes les plus instables durant la phase linéaire de la simulation.

Méthode	Padé	Hermite					
		N=2	N=3	N=4	N=6	N=8	N=16
Taux de croissance γ	.10436	.12246	.09420	.09625	.09643	.09644	.09644

TABLE 5.4 – Taux de croissance du mode le plus instable $(m, n) = (14, -10)$ sur le cas test Cyclone avec différentes versions de l'opérateur de gyromoyenne et différents nombres de points d'interpolation pour la méthode Hermite (normalisé de la même façon que sur la Figure 1 de [Da00]).

Le Tableau 5.4 donne le taux de croissance du mode $(14, -10)$ (normalisé de la même façon que [Da00], Figure 1) observé avec différentes configurations de l'opérateur de gyromoyenne. Ce taux est caractéristique du comportement des principales grandeurs du code durant la phase

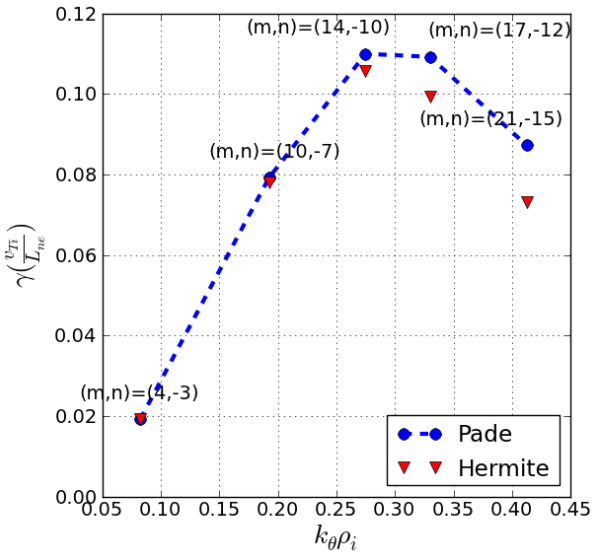


FIGURE 5.6 – Comparaison des taux de croissance linéaire sur le cas Cyclone entre l'utilisation de la gyromoyenne par approximation de Padé (●) et la gyromoyenne basée sur l'interpolation d'Hermite (▼).

linéaire. On peut voir que les taux obtenus avec la méthode Hermite convergent lorsque N augmente. Ce comportement indique une bonne convergence des simulations. De plus, la valeur asymptotique correspond à la valeur attendue. D'autre part, pour $N = 2$, $N = 3$ et pour la méthode de Padé, les taux de croissance obtenus sont moins précis. En pratique, $N = 8$ devrait être utilisé dans les simulations de production. Ce choix représente un bon compromis entre précision et performance.

Dans la littérature (voir par exemple [Bro08], p. 39-40), des travaux théoriques et appliqués montrent que $N = 8$ points d'interpolation pour l'opérateur de gyromoyenne est souvent suffisant. La section 5.3 montre aussi que ce choix est suffisant pour que la gyromoyenne basée sur l'interpolation d'Hermite soit une meilleure approximation de $J_0(k_\perp \rho_i)$ sur l'intervalle $k_\perp \rho_i \in [0, 5]$ que l'approximation de Padé. Les configurations avec $N = 16$ sont utiles dans des cas spécifiques où une plus grande précision serait nécessaire. Par exemple lorsque les particules avec $k_\perp \rho_i > 5$ jouent un rôle important pour une simulation.

La Figure 5.6 montre les différents taux de croissance qu'on obtient pour les 5 modes les plus instables de la simulation. La courbe en pointillés bleu correspond aux simulations effectuées avec la gyromoyenne par approximation de Padé et les points rouges ▼ correspondent à celles effectuées avec la gyromoyenne basée sur l'interpolation d'Hermite. On peut constater dans un premier temps que les points obtenus sont proches. On peut aussi voir que pour les modes les plus élevés, les taux de croissance obtenus par la méthode d'Hermite sont inférieurs à ceux obtenus par approximation de Padé. Ceci est cohérent avec la seconde étude de la section 5.3.2 où on peut voir que sur l'intervalle $[1, 5]$ l'approximation de Padé surestime les valeurs de la fonction de Bessel J_0.

Le Tableau 5.5 détaille le temps d'exécution en donnant le temps passé dans les parties *Diagnostique* et *Solveur champs* du code. Ces parties sont impactées par le choix de la méthode pour l'opérateur de gyromoyenne. Les versions optimisées `ré-arrangement` et `blocage` divisent par 2 le surcoût de l'interpolation d'Hermite dans la partie *Diagnostiques* par rapport à

Code \ Méthode	Padé	Hermite					
		N=8			N=16		
		Initiale	Ré-arrangement	Blocage	Initiale	Ré-arrangement	Blocage
Solveur champ	28 (0%)	32 (+14%)	31 (+10%)	31 (+11%)	35 (+27%)	32 (+13%)	32 (+16%)
Diagnostiques	96 (0%)	123 (+29%)	108 (+12%)	110 (+15%)	147 (+54%)	114 (+19%)	120 (+26%)
Total	629 (0%)	662 (+5.3%)	659 (+4.8%)	651 (+3.6%)	689 (+9.6%)	670 (+6.6%)	666 (+5.8%)

TABLE 5.5 – Temps d'exécution en secondes cumulés de certaines portions du code qui sont modifiées par le coût d'exécution de la gyromoyenne. Ces temps ont été obtenus sur des simulations de 60 itérations en utilisant 512 cœurs. Le pourcentage donné entre parenthèses correspond au surcout d'exécution par rapport à l'utilisation de la gyromoyenne par approximation de Padé.

l'implémentation `initiale`. Sur le temps total d'exécution, la meilleure méthode a un surcoût de 4% par rapport à l'approximation de Padé avec $N = 8$ points et seulement 6% avec $N = 16$. Ce surcout est tout à fait acceptable pour l'utilisation de cette méthode sur des simulations de production. Les optimisations réalisées sur la gyromoyenne ont permis de réduire son temps d'exécution d'environ 40 % par rapport à la version `initiale`.

Conclusion

Nous avons comparé dans ce chapitre la gyromoyenne existante par approximation de Padé et la gyromoyenne basée sur l'interpolation d'Hermite du point de vue de la précision des résultats et du point de vue des performances. Nous avons défini un cadre analytique qui nous a permis d'étudier la sensibilité du calcul de la gyromoyenne par méthode Hermite en fonction des données d'entrée. Cette première étude a permis d'établir que la méthode Hermite est une meilleure approximation de la fonction de Bessel J_0 que la méthode Padé sur un intervalle plus large que $k_\perp \rho \in [0, 1]$. Dans un second temps, nous avons reproduit les simulations du cas de référence Cyclone avec les deux méthodes de gyromoyenne. Nous avons comparé les performances et les résultats obtenus sur les différentes simulations. Cette étude nous permet de conclure que la méthode Hermite réalise un bon compromis entre précision et temps d'exécution puisque pour un surcoût de l'ordre de 5%, les simulations bénéficient d'un opérateur de gyromoyenne plus précis.

Pour comparer l'impact de ces versions de l'opérateur de gyromoyenne dans GYSELA, différentes étapes intermédiaires ont dues être réalisées. La première étape a consisté à intégrer la gyromoyenne à base d'interpolation d'Hermite dans GYSELA. Durant sa conception, nous souhaitions réaliser l'implémentation de l'opérateur de gyromoyenne son forme d'une bibliothèque. C'est une bonne pratique du point de vue génie logiciel, car cela nécessite de définir clairement les interfaces entre l'application cliente, GYSELA, et la bibliothèque, la gyromoyenne. Dans un second temps, nous avons rendu l'implémentation de la gyromoyenne *thread-safe* afin de pouvoir utiliser les fonctions de calcul de gyromoyenne au sein de zones parallèles OPENMP. Cela nous permet de calculer de façon concurrente la gyromoyenne de plusieurs plans poloïdaux. Ces travaux d'ingénierie sont indispensables pour la réalisation des études présentées dans ce manuscrit.

Le bénéfice de la gyromoyenne base sur l'interpolation d'Hermite ne s'arrête pas à l'amélioration de la précision numérique des calculs. Cette méthode qui se base sur des interpolations locales permet d'envisager une parallélisation du calcul de la gyromoyenne par bloc

$2D$ dans le plan poloïdal. Cette parallélisation représente un enjeu important pour l'application GYSELA. L'opérateur de gyromoyenne par approximation de Padé, qui est l'implémentation actuellement utilisée par défaut, impose la connaissance de l'intégralité du plan poloïdal car il s'appuie sur des transformées de Fourier dans la direction θ (voir section 4.2.3). Cette contrainte représente un verrou technologique comme nous le verrons dans le chapitre suivant. La parallélisation de la gyromoyenne basée sur l'interpolation Hermite permettrait de supprimer ce verrou technologique. Ce point fera l'objet du chapitre suivant.

Chapitre 6

Parallélisation en MPI de l'opérateur de gyromoyenne

Sommaire

Capturer le comportement de structures très fines dans la fonction de distribution permettrait à GYSELA d'accéder à des simulations plus réalistes. Une plus grande finesse de maillage serait à l'heure actuelle surtout plus importante dans les directions r et θ. L'augmentation de cette résolution poloïdale du maillage est actuellement limitée d'une part par certains opérateurs qui couplent fortement ces directions r et θ (dont l'opérateur de gyromoyenne) et empêche ainsi une parallélisation efficace dans ces dimensions, et d'autre part par une empreinte mémoire trop large liée à la présence en mémoire de plan poloïdaux complets pour simplifier un certain nombre de calculs.

Dans GYSELA, l'opérateur de gyromoyenne est mis à contribution dans le solveur de Poisson (calcul du champ électrique), mais aussi dans les fonctions de diagnostiques. Durant une itération temporelle, la fonction de distribution est le plus souvent décomposée de sorte à ce que chaque processus MPI soit responsable du sous-domaine défini[17] par $\bar{f}(r = [i_{start}, i_{end}], \theta = [j_{start}, j_{end}], \varphi = *, v_{\|} = *, \mu = \mu_{value})$, avec les valeurs $i_{start}, i_{end}, j_{start}, j_{end}, \mu_{value}$ localement connues par chaque processus (voir section 1.3.2, p. 11). Actuellement, certains opérateurs de GYSELA, dont la gyromoyenne, nécessitent le rapatriement de plans poloïdaux ($r = *, \theta = *$) complets pour pouvoir s'exécuter. Pour que les processus aient accès localement à l'intégralité des points dans les directions (r, θ), il est nécessaire de transposer la fonction de distribution de sorte à obtenir une décomposition de domaine de la forme $\bar{f}(r = *, \theta = *, \varphi = [k_{start}, k_{end}], v_{\|} =$

17. La notation $*$ représente toutes les valeurs d'une dimension donnée.

$[l_{start}, l_{end}], \mu = \mu_{value})$, avec les valeurs $k_{start}, k_{end}, l_{start}, l_{end}$ localement connues par chaque processus. Cette transposition implique des communications collectives qui ne sont pas favorables à la scalabilité.

L'utilisation de l'opérateur de gyromoyenne parallèle qui est proposé dans ce chapitre permettrait plusieurs avancées. Dans les diagnostiques, cela éviterait des communications collectives coûteuses qui reconstituent par transposition les plans poloïdaux, et donc par suite d'améliorer la scalabilité de l'application. Du point de vue de la consommation mémoire, il s'avère que la gyromoyenne utilise des structures de données $2D$ dont la taille est proportionnelle au plan poloïdal. Une version parallèle de la gyromoyenne conduirait à supprimer ces structures qui ont été identifiées comme pénalisantes pour le passage à plus grande échelle (voir Tableau 3.3, p. 52). L'utilisation de la gyromoyenne parallèle dans Gysela autorisera des maillages plus raffinés dans le plan poloïdal et donc facilitera l'accès à la dynamique des électrons cinétiques. Ce dernier point est un objectif visé par les physiciens utilisateurs de Gysela.

Pour toutes ces raisons, les bénéfices directement liés à l'utilisation d'un opérateur de gyromoyenne parallèle dans Gysela sont conséquents. Le développement d'une version parallèle de l'opérateur de gyromoyenne étant invasif dans un code de production tel que Gysela, nous avons développé une application prototype qui nous a permis de tester cette extension de la gyromoyenne et de réaliser les études de performances présentées dans ce chapitre. La section 6.1 décrit la solution parallèle retenue. La section 6.2 présente les courbes de scalabilité en temps d'exécution de cette gyromoyenne parallèle. La section 6.3 décrit les contraintes d'utilisation liées à la parallélisation de la gyromoyenne que nous avons identifiées et enfin la section 6.4 conclut ce chapitre.

6.1 Solution pour un opérateur de gyromoyenne parallèle

Dans le chapitre 4, nous avons défini l'opérateur de gyromoyenne basé sur l'interpolation d'Hermite. Cette méthode d'interpolation a la propriété d'être très locale en espace et se prête donc bien à une parallélisation en mémoire distribuée via MPI. Nous avons développé une application prototype qui correspond à une version parallèle du cas test analytique de la gyromoyenne (voir section 5.1, p. 83). Ceci nous permet d'identifier précisément les contraintes d'utilisation de cette solution parallèle. L'approche décrite dans ce chapitre représente le travail préparatoire avant l'intégration de l'opérateur de gyromoyenne parallèle dans Gysela.

Nous présentons la décomposition de domaine adoptée dans la section 6.1.1. La section 6.1.2 donne des éléments pour l'estimation du nombre de points fantômes nécessaires pour la réalisation des calculs de gyromoyenne en parallèle. La section 6.1.3 décrit l'application prototype dans son ensemble et l'algorithme utilisé.

6.1.1 Décomposition de domaine du plan poloïdal

La version parallèle de la gyromoyenne présentée ici adopte la même décomposition de domaine du plan poloïdal que celle utilisée durant la majeure partie d'une itération temporelle dans Gysela, à savoir une décomposition par blocs dans les directions (r, θ). La Figure 6.1 illustre la décomposition d'un petit plan poloïdal avec 4 processus MPI. Une section grisée désigne une portion du plan poloïdal associée à un processus MPI. Cette portion de maillage propre à un processus MPI sera appelée *sous-domaine propre* par la suite. L'ensemble des sous-domaines propres partitionne le plan poloïdal complet. Pour chaque nœud du maillage poloïdal, le calcul de sa gyromoyenne est réalisé par le processus qui inclut ce nœud.

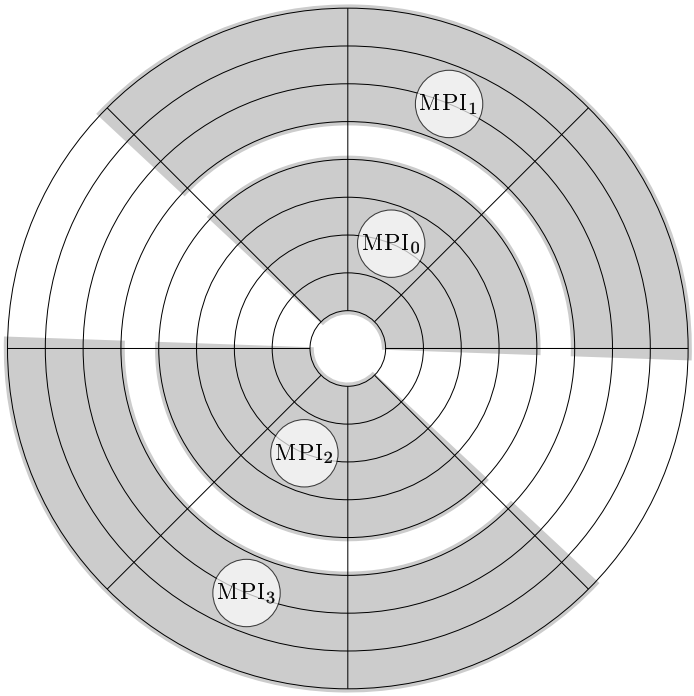


FIGURE 6.1 – Illustration de la décomposition d'un petit maillage poloïdal en 4 processus MPI. Paramètres : $N_r = 9$, $N_\theta = 8$.

Pour concevoir un algorithme parallèle en mémoire distribué, il est nécessaire d'analyser finement les dépendances de données manipulées. Cette identification permet de mettre en place les communications et transferts de données. La solution parallèle a été réalisée en s'appuyant sur la version optimisée par blocage de boucle (voir section 4.5.2, page 79), car c'est la méthode qui présentait les meilleurs performances.

6.1.2 Calcul de la taille des zones fantômes

Dans le cas du calcul de gyromoyenne par interpolation d'Hermite, des points fantômes sont nécessaires pour calculer l'ensemble de valeurs W_f (voir section 4.3.2, p. 66). Pour rappel, pour une grandeur f définie sur le plan poloïdal, cet ensemble de valeurs désigne pour un point (r_i, θ_j) du maillage les quantités suivantes :

$$f(r_i,\theta_j), \frac{\partial}{\partial r} f(r_i,\theta_j), \frac{\partial}{\partial \theta} f(r_i,\theta_j), \frac{\partial^2}{\partial r \partial \theta} f(r_i,\theta_j).$$

Pour expliciter les informations nécessaires au calcul des valeurs W_f, intéressons nous au premier point du sous-domaine (r, θ) propre au processus MPI_0. Il est représenté par ● sur la Figure 6.2. Ce point se situe sur une bordure du domaine attribué au processus MPI_0. De plus, il a la particularité d'être le premier point dans la direction radiale du plan poloïdal. Les points d'interpolation sont représentés par ▲. Dans le cas où l'un de ces points se situe à l'extérieur du plan poloïdal, on considère sa projection radiale représentée par ▼ lors du calcul de l'interpolation. La gyromoyenne du point ● peut se calculer à condition de connaître les valeurs W_f aux 4 coins des cellules (■) qui contiennent la position d'au moins un point d'interpolation.

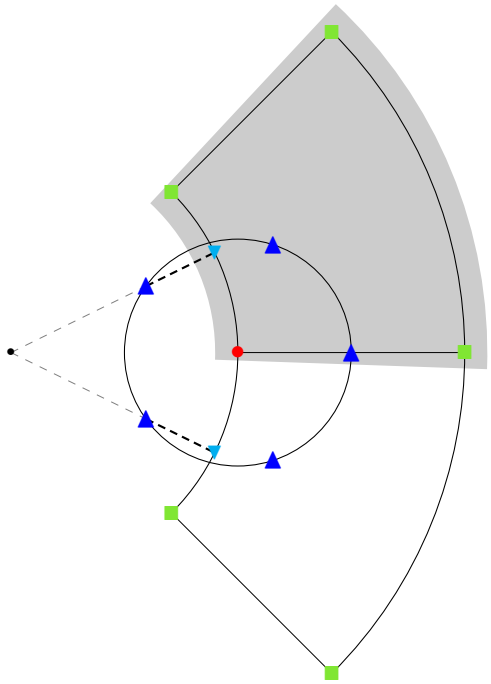


FIGURE 6.2 – Gyromoyenne du point au coin inférieur gauche (●) du sous-domaine propre du processus MPI_0 avec 5 point d'interpolation (▲). Les points d'interpolations à l'extérieur du maillage sont projetés radialement (▼). Les points potentiellement fantômes du maillage sous-jacent qui contribuent au calcul sont marqués par ■. Le point • représente le centre du maillage poloïdal.

Sur le cas illustré par la Figure 6.2, on se rend compte qu'en raison des conditions limites que l'on utilise dans la direction r (la projection radiale), le nombre de points de maille (■) qui intervient dans le calcul n'est pas le même suivant les directions (r, θ). On peut voir sur la figure qu'il est nécessaire de connaître les W_f de deux points de maillage extérieurs au processus MPI_0.

La section 4.3.2 (p. 66) donne la formule du calcul de dérivée dans le cas unidimensionnel. Dans le cas de l'interpolation $2D$, il est nécessaire de calculer les dérivées partielles selon deux directions du plan poloïdal et la dérivée croisée. Afin de donner les expressions de ces dérivées, nous réutilisons les notations introduites précédemment ; chaque point du plan poloïdal est identifié par un couple de valeur (r_i, θ_j) tel que :

$$\begin{aligned} r_i &= r_{\min} + i\frac{r_{\max} - r_{\min}}{N_r}, & i &\in [\![0, N_r]\!] \\ \theta_j &= j\frac{2\pi}{N_\theta}, & j &\in [\![0, N_\theta - 1]\!]. \end{aligned}$$

En un point du plan (r_i, θ_j), le calcul des dérivées partielles est très similaire à la formule du cas $1D$:

$$\frac{\partial f}{\partial r}(r_i, \theta_j) \approx \frac{1}{12\Delta r}\Big(f(r_{i-2}, \theta_j) - 8f(r_{i-1}, \theta_j) + 8f(r_{i+1}, \theta_j) - f(r_{i+2}, \theta_j)\Big)$$

et

$$\frac{\partial f}{\partial \theta}(r_i, \theta_j) \approx \frac{1}{12\Delta \theta}\Big(f(r_i, \theta_{j-2}) - 8f(r_i, \theta_{j-1}) + 8f(r_i, \theta_{j+1}) - f(r_i, \theta_{j+2})\Big).$$

L'expression de la dérivée croisée s'exprime de la façon suivante :

$$\frac{\partial^2 f}{\partial\theta\partial r}(r_i,\theta_j) = \frac{\partial}{\partial\theta}\frac{\partial f}{\partial r}(r_i,\theta_j)$$
$$\approx \frac{1}{12\Delta\theta}\Big(\frac{\partial f}{\partial r}(r_i,\theta_{j-2}) - 8\frac{\partial f}{\partial r}(r_i,\theta_{j-1}) + 8\frac{\partial f}{\partial r}(r_i,\theta_{j+1}) - \frac{\partial f}{\partial r}(r_i,\theta_{j+2})\Big).$$

Pour pouvoir calculer l'ensemble des valeurs W_f en un point (r_i,θ_j) du plan, il est donc nécessaire de connaître des valeurs dans un voisinage de ce dernier. La Figure 6.3 illustre la forme carrée du stencil nécessaire pour l'évaluation des valeurs W_f en un point ■ du plan.

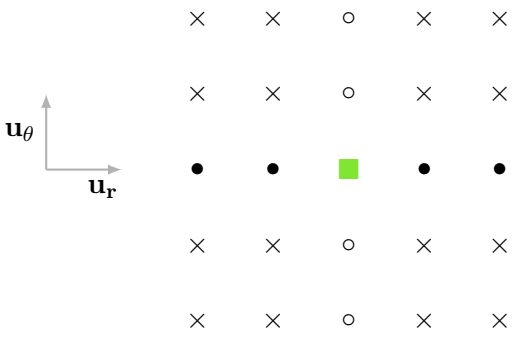


FIGURE 6.3 – Stencil autour d'un point source ■. Au point ■, les valeurs de la fonction f aux points «•» dans la direction radiale sont nécessaires pour calculer $\frac{\partial f}{\partial r}$; de même les points «○» dans la direction poloïdale sont nécessaires pour calculer $\frac{\partial f}{\partial \theta}$ et ainsi que les points «×» qui sont nécessaires pour calculer $\frac{\partial^2 f}{\partial r\partial\theta}$.

Le calcul de la gyromoyenne en un point dépend :
— de la position des points sources (■) qui contribuent au calcul ;
— de la forme du stencil pour calculer les valeurs W_f en un point source (■).
A partir de ces informations, il est possible de déduire l'ensemble des points du plan poloïdal qui est nécessaire aux calculs de gyromoyenne en chaque point du maillage. Pour le calcul de la gyromoyenne du point • illustré par la Figure 6.2, il est nécessaire de connaître les valeurs des points du maillage sous-jacent sur un stencil carré de taille 7×7 et dont le point central est le point cible •.

En réalisant ce travail pour l'ensemble des points en bordure d'un domaine MPI, on en déduit les dimensions dans les directions (r,θ) de la zone fantôme nécessaire pour ce domaine. Dans l'application prototype décrite dans la section suivante, tous les processus ont des zones fantômes de même dimension.

6.1.3 Algorithme parallèle de l'opérateur de gyromoyenne

Le calcul de la gyromoyenne dans sa version séquentielle s'effectue en deux temps. Durant une première phase d'initialisation, on calcule les poids associés aux valeurs W_f du plan pour chaque rayon r_i (voir Algorithme 3, p. 72). Durant un appel à la fonction `gyro_compute()`, les valeurs W_f sont calculées en chaque point du plan d'entrée et des produits scalaires creux permettent de calculer la valeur des gyromoyennes en chaque point du plan (voir Algorithme 4, p. 72).

Le développement de l'extension en MPI de ces deux composants de la gyromoyenne a nécessité des changements par rapport à l'implémentation séquentielle. Parmi les différences

les plus notables, on peut citer la réécriture des algorithmes qui tiraient profit d'une vision d'ensemble du plan poloïdal. Par exemple, lors du calcul de la position d'un point d'interpolation, nous considérons sa projection radiale dans le cas où ce dernier est à l'extérieur du domaine de définition du plan poloïdal. Dans une décomposition par blocs du plan poloïdal, pour pouvoir savoir si un point d'interpolation appartient ou non au domaine de définition, il est nécessaire de connaître la position relative du sous-domaine propre à un processus MPI par rapport au domaine globale.

Entrée : Un plan poloïdal
Sorties : Gyromoyenne en chaque point du plan d'entrée

1 Calcul de la taille des zones fantômes;
2 Initialisation du module de gyromoyenne;
3 Initialisation à la valeur analytique des différents blocs du plan poloïdal;
4 Communication des zones fantômes entre les processus frontaliers;
5 Calcul de la gyromoyenne sur le bloc local : calcul des dérivées W_f sur le sous-domaine propre et sur la zone fantôme, et produits scalaires creux sur le sous-domaine propre;
6 Calcul de l'erreur par rapport à la solution analytique.

Algorithme 5 : Algorithme du cas test analytique parallèle.

Pour illustrer la solution parallèle, l'Algorithme 5 donne le pseudo-algorithme du cas test analytique parallèle que nous avons développé. Par rapport à la version séquentielle, on peut remarquer l'apparition des étapes relatives aux zones fantômes en lignes 1 et 4. Durant l'étape d'initialisation du module de gyromoyenne en ligne 2, chaque processus MPI calcule les poids associés aux valeurs W_f de leur sous-domaine propre. C'est aussi durant cette étape que la taille de la zone fantôme est calculée pour chaque processus MPI. Durant le calcul de gyromoyenne parallèle à l'étape 5, chaque processus effectue le calcul sur son sous-domaine propre.

Ce programme permet de vérifier efficacement la validité des calculs. Nous nous appuyons sur ce dernier pour réaliser les études de scalabilité de l'opérateur de gyromoyenne. Les résultats sont donnés dans la section suivante.

6.2 Étude de performances

Dans le domaine du HPC, deux tests sont souvent utilisés pour estimer la qualité de la parallélisation d'un code : le *strong* et le *weak scaling*. Leur définition et leur protocole expérimental sont donnés dans les sections suivantes. Les benchmarks ont été réalisés sur le cas test parallélisé sur la machine Poincare. La configuration de la machine Poincare est donnée dans le tableau 6.1.

Nombre de nœuds	92
Quantité de mémoire par nœud	32 Go
Nombre de processeurs par nœud	2
Famille de processeur	Intel(R) Xeon(R) CPU E5-2670 (2.60 GHz)
Nombre de cœurs par processeur	8

Table 6.1 – Principales caractéristiques de la machine Poincare.

Après avoir testé différentes implémentations MPI sur cette machine, nous avons retenu la

bibliothèque OpenMPI 1.6.3 avec laquelle nous avons obtenu les meilleurs temps d'exécution sur nos tests de performances [18].

Les sections 6.2.1 et 6.2.2 présentent respectivement l'étude en *strong* et en *weak scaling*. Un bilan est donné et différentes perspectives d'amélioration de l'opérateur de gyromoyenne parallèle sont discutées dans la section 6.2.3.

6.2.1 Strong scaling

Pour un programme donné, l'étude en *strong scaling* consiste à relever les temps d'exécution du programme pour une instance de taille fixe en faisant varier le nombre de processus MPI. Par exemple, si pour une instance de taille T, le temps d'exécution du programme est de $t_{\text{réf}}$ avec P processus MPI, on peut s'attendre à ce que ce temps d'exécution soit divisé par n si on exécute le programme avec $n \times P$ processus sur la même instance. Si c'est effectivement le cas, on dit alors que le *scaling* est parfait. En pratique, ce n'est généralement pas le cas à cause d'un certain nombre de surcoûts.

Protocole

Dans le cadre de l'étude de la gyromoyenne, la taille de l'instance d'entrée correspond à la taille du plan poloïdal $N_r \times N_\theta$, qui est ici fixée à $\mathbf{2^{14} \times 2^{14}}$, soit $\mathbf{16384 \times 16384}$. Entre deux simulations, on double le nombre de processus dans la direction r et θ ; le nombre de processus total est donc multiplié par 4. Nous avons effectué les simulations avec 1, 4, 16 et 64 processus MPI. Afin de disposer d'un maximum de mémoire par processus, chaque processus dispose d'un nœud complet [19]. Pour l'ensemble des simulations présentées, nous utilisons 32 points d'interpolation sur le cercle de Larmor et un plan poloïdal avec $r_{min} = 0.1$ et $r_{max} = 0.9$.

L'ensemble de Figures 6.4 montre les courbes des temps d'exécution et des efficacités relatives obtenues durant cette étude. Ces courbes correspondent aux opérations suivantes de l'application prototype parallèle :

- `send` : temps de communication des points fantômes entre processus MPI ;
- `init.` : temps de l'initialisation du plan poloïdal. Chaque processus MPI initialise son sous-domaine propre par une fonction dont on sait calculer analytiquement la gyromoyenne ;
- `deriv.` : temps de calcul des valeurs W_f sur l'ensemble des points du plan. Chaque processus MPI calcule les valeurs W_f sur son sous-domaine propre ainsi que sur sa zone fantôme ;
- `interp.` : temps de calcul de la moyenne des interpolations sur l'ensemble des points du plan. Chaque processus MPI calcule les produits scalaires creux en chaque point de son sous-domaine propre.

Il est important de noter que localement, un processus MPI dispose des points du maillage dont il est chargé de calculer la gyromoyenne, ainsi que des points fantômes nécessaires pour calculer l'ensemble des dérivées qui interviennent dans les calculs de gyromoyenne. Dans cette version parallèle de la gyromoyenne, la taille de la zone fantôme du maillage est identique pour l'ensemble des processus MPI, ce qui simplifie les communications de données [20]. La taille de la zone fantôme d'un processus dépend du Larmor ρ et de la forme des cellules du maillage. Dans une étude en *strong scaling*, la taille du maillage est fixée, la forme des cellules du maillage

18. Les deux autres implémentations testées étaient IntelMPI et Mvapich2.
19. La mémoire disponible pour un processus utilisateur est limitée à 28 Go sur un nœud de Poincare.
20. Cette hypothèse peut être remise en cause néanmoins pour réduire le volume de communication global.

est donc identique pour l'ensemble des simulations. Dans ce cas, le nombre de points fantômes dépend uniquement du rayon de Larmor. Pour observer son impact sur notre prototype parallèle, nous avons effectué des tests avec les valeurs $\rho = 0.01$, $\rho = 0.03$ et $\rho = 0.099$.

Dans le cas d'une étude en *strong scaling*, et pour une opération donnée, l'efficacité relative se calcule à partir de la simulation qui utilise le moins de processus MPI qui est considérée comme un cas de référence. L'efficacité relative en pourcentage se calcule à partir de la formule suivante :

$$\text{Efficacité} = \frac{\#proc_{\text{réf}} \cdot t_{\text{réf}}}{\#proc_{cible} \cdot t_{cible}} \times 100.$$

Dans le cas où le *scaling* est parfait, la courbe d'efficacité relative est horizontale, égale à 100%. Pour les courbes `init.`, `deriv.` et `interp.`, la simulation de référence est celle utilisant 1 processus, alors que pour la courbe `send`, nous prenons la simulation qui utilise 4 processus comme référence car c'est la plus petite simulation qui effectue des communications.

Interprétation

Pour simplifier l'interprétation des Figures 6.4, nous allons commenter les courbes `init.`, `deriv.`, `interp.` et `send` une par une.

L'initialisation parallèle du plan poloïdal ne dépend pas de la valeur de ρ mais uniquement du nombre de processus utilisés. Les temps d'initialisation sont identiques quelle que soit la valeur de ρ. La courbe `init.` est donc la même sur les Figures 6.4a, 6.4c et 6.4e. On peut l'utiliser comme courbe de référence pour observer les tendances des autres courbes en fonction de ρ. On peut voir que la courbe `init.` est effectivement décroissante lorsque le nombre de processus MPI augmente.

ρ	0.01	0.03	0.099
# points fantômes (r, θ)	$(209, 262)$	$(619, 794)$	$(2032, 3710)$

TABLE 6.2 – Évolution du nombre de points fantômes en fonction de ρ.
Paramètres : $N_r = N_\theta = 2^{14}$.

Sur les Figures 6.4a, 6.4c et 6.4e, la courbe `deriv.` correspond au temps d'exécution du calcul des dérivées aux points locaux à un processus MPI. Comme on peut s'y attendre dans le cas d'une bonne scalabilité, elle est décroissante lorsque le nombre de processus augmente. En regardant son évolution par rapport à ρ, on voit que cette opération est de plus en plus coûteuse lorsque ρ augmente. En outre, la pente de cette courbe est de moins en moins accentuée lorsque ρ augmente. Ceci se confirme clairement lorsqu'on regarde son efficacité relative sur les Figures 6.4b, 6.4d et 6.4f. Ce comportement est dû au fait que le calcul des dérivées se fait sur le sous-domaine propre d'un processus MPI, plus sur ses points fantômes. Hors, le nombre de points fantômes augmente avec ρ comme on peut le voir sur le Tableau 6.2 ; pour une valeur de ρ donnée, ce tableau indique la largeur de la zone fantôme dans les directions r et θ. La zone fantôme d'un processus, dont la taille dépend donc de ρ, constitue un coût de calcul séquentiel pour le calcul des dérivées. Ceci explique l'augmentation du temps d'exécution en fonction de ρ. Ce même argument explique la baisse de l'efficacité relative lorsque ρ augmente. Plus la taille de la zone fantôme est importante par rapport à la taille du sous-domaine propre, moins bonne est l'efficacité relative.

La courbe `interp.` présente de loin le meilleur comportement en terme de scalabilité. Malgré le fait que ce soit l'opération la plus coûteuse en terme de temps de calcul (Figures 6.4a, 6.4c,

6.4e), on peut voir sur les Figures 6.4b, 6.4d, 6.4f que cette opération affiche une efficacité relative supérieure à 100%. Ceci est dû au fait que la taille du second membre lors du produit scalaire creux diminue lorsqu'on augmente le nombre de processus MPI (voir Figure 4.9, p. 77). Dans le cas séquentiel, ce vecteur est idéalement de taille $4 \times (N_r + 1) \times N_\theta$, alors que dans le cas parallèle, la taille de ce vecteur est proportionnelle à la taille du sous-domaine propre d'un processus. Lorsque le nombre de processus utilisés augmente, la taille du sous-domaine propre diminue et donc la taille du vecteur qui intervient lors des produits scalaires creux diminue aussi. Plus le vecteur est petit, plus les produits scalaires creux bénéficient des effets de mémoire cache. Comme il a été dit plus haut, plus ρ est grand, plus la zone fantôme est grande. Cette zone croît en fonction du nombre de processus ; c'est un surcoût parallèle incontournable qui se traduit par un affaissement des courbes d'efficacité relative lorsque ρ augmente.

Enfin, sans considérer le cas mono-processus sur les Figures 6.4a, 6.4c, 6.4e, pour les deux premières valeurs de ρ, on voit que le temps des communications MPI est faible et relativement constant avec le nombre de processus utilisés (courbe `send`). Dans une étude en *strong scaling*, pour une valeur de ρ donnée, la taille de la zone fantôme d'un processus est fixe et donc indépendante du nombre de processus MPI. Le schéma de communication de notre application parallèle est plutôt simple car chaque processus MPI communique avec ses 4 processus frontaliers. En sachant qu'un processus MPI correspond à un nœud de calcul, ce comportement est cohérent avec le fait que la bande passante totale disponible augmente avec le nombre de nœuds utilisés. Malgré le fait que le volume global de communication augmente linéairement avec le nombre de processus utilisés, ceci est compensé par le fait que la bande passante du réseau utilisable augmente aussi de façon linéaire. D'autre part, on peut noter que les coûts de communication augmentent avec la valeur de ρ comme on peut s'y attendre car le volume de données à échanger entre processus augmente avec la valeur de ρ.

Les opérations liés au calcul de la gyromoyenne, `deriv.` et `interp.`, montrent dans cette étude une bonne scalabilité.

6.2.2 Pseudo weak scaling

Pour un programme donné, l'étude en *weak scaling* consiste à relever les temps d'exécution du programme en faisant varier de façon proportionnelle la taille de l'instance d'entrée et le nombre de processus MPI utilisés. Par exemple, si pour une instance de taille T, le temps d'exécution du programme est t avec P processus MPI, l'exécution suivante dans l'étude consistera à utilisé kP processus MPI sur une instance de taille kT. Dans le cas où le *scaling* est parfait, le temps de calcul reste constant sur toutes les simulations. Ici, l'objectif est de conserver une taille de domaine globale proportionnelle au nombre d'unités de calcul utilisées.

Protocole

Pour réaliser cette étude, nous faisons varier la taille du plan poloïdal de $\mathbf{2^{11}} \times \mathbf{2^{11}}$ à $\mathbf{2^{14}} \times \mathbf{2^{14}}$ et le nombre de processus MPI de 16 à 1024 comme on peut le voir dans le Tableau 6.3. Un processus MPI correspond à un cœur de calcul[21] et nous déployons 16 processus MPI par nœud sur Poincare. De façon similaire à l'étude précédente, nous utilisons 1, 4, 16 et 64 nœuds de la machine. De façon identique à l'étude précédente, nous nous intéressons aux opérations `init.`, `deriv.`, `interp.` et `send` ; nous utilisons 32 points d'interpolation sur le cercle de Larmor

21. Dans cette étude, un processus dispose localement de 16 fois moins de mémoire que dans l'étude en *strong scaling*.

	# nœuds	1	4	16	64
	# proc MPI	16	64	256	1024
	$N_r = N_\theta$	2^{11}	2^{12}	2^{13}	2^{14}
ρ	0.01	$(30, 35)$	$(56, 68)$	$(107, 133)$	$(209, 262)$
	0.03	$(81, 101)$	$(158, 200)$	$(312, 398)$	$(619, 794)$
	0.099	$(258, 466)$	$(511, 930)$	$(1018, 1856)$	$(2032, 3710)$

TABLE 6.3 – Évolution du nombre de points fantômes dans les directions (r, θ) en fonction de ρ et de la taille du maillage $N_r \times N_\theta$. Le nombre de nœuds de calcul et le nombre de processus MPI sont indiqués.

et un plan poloïdal avec $r_{min} = 0.1$ et $r_{max} = 0.9$. Nous avons fait varier le rayon de Larmor avec les valeurs suivantes $\rho = 0.01$, $\rho = 0.03$ et $\rho = 0.099$.

L'étude que nous présentons ici ne correspond pas strictement à la définition d'une étude en *weak scaling*. Un des aspects importants d'une étude en *weak scaling* est de maintenir une taille de sous-domaine propre constante par processus MPI entre les simulations, hors ce n'est pas le cas dans l'étude que nous présentons ici. Comme on peut le voir sur le Tableau 6.3, pour une valeur ρ donnée, le nombre de points fantômes augmente entre deux simulations. Ceci vient du fait que la forme des cellules du maillage deviennent de plus en plus petite lorsqu'on augmente la résolution du plan poloïdal. Plus un maillage est fin, plus un cercle de Larmor intersecte un nombre de cellules important. C'est pourquoi nous qualifions cette étude de *pseudo weak scaling*.

En revanche, la taille des sous-domaines propres à chaque processus est constante lorsqu'on augmente le nombre de processus. Dans les tests réalisés cette taille est de $2^9 \times 2^9$, soit 512×512.

Interprétation

L'ensemble de Figures 6.5 montre l'évolution des temps d'exécution des opérations `init.`, `deriv.`, `interp.` et `send` en fonction du nombre de processus MPI et de la taille du rayon de Larmor.

De façon identique à l'étude précédente, le temps d'exécution de l'initialisation du plan ne dépend pas du rayon de Larmor ; la courbe `init.` peut donc être utilisée comme référence pour observer la tendance des autres courbes en fonction de ρ. Aussi, le temps d'initialisation relevé ici correspond au temps que prend un processus pour initialiser son sous-domaine propre. La taille des sous-domaines propres étant constante sur les différentes simulations, on observe sur les Figures 6.5a, 6.5b et 6.5c que la courbe `init.` est proche de l'horizontale.

Pour une valeur de ρ donnée, on peut voir une légère croissance de la courbe `interp.` lorsque le nombre de processus MPI augmente. Ceci est dû au fait que la taille du second membre des produits scalaires creux augmente avec le nombre de processus utilisés. Comme expliqué dans l'étude précédente, la taille de ce vecteur est proportionnelle au nombre de points du sous-domaine propre et de la zone fantôme d'un processus. Hors le nombre de points fantômes augmente lorsqu'on raffine le maillage, et donc la taille du second membre. Ce comportement n'est pas favorable aux effets de cache, ce qui explique aussi la croissance modérée de la courbe `interp.`. Néanmoins, on peut voir que la position relative de cette courbe par rapport à la courbe `init.` évolue très peu lorsque la valeur de ρ augmente.

Les courbes `deriv.` et `send` montrent en revanche une évolution spectaculaire en fonction de la valeur de ρ, et particulièrement avec $\rho = 0.099$. Pour une valeur de ρ donnée, la croissance de ces courbes est due au fait que le nombre de points fantômes augmente avec le nombre

de processus utilisés. Plus la quantité de points fantômes est grande (voir Tableau 6.3), plus le coût de ces opérations est élevé. Sur les simulations réalisées avec $\rho = 0.099$, on peut voir que le calcul des dérivées domine le coût de l'interpolation sur les plus grandes simulations. La simulation $\rho = 0.099$ et *#proc* MPI $= 1024$ représente le pire scénario, car dans ce cas le coût des communications (3.75 s.) représente plus de temps que la somme des coûts de calcul ($0.12 + 0.42 + 2.96 = 3.5$ s.). Néanmoins, dans les cas $\rho = 0.01$ et $\rho = 0.03$, les courbes de scalabilité sont suffisamment bonnes pour que l'opérateur de gyromoyenne parallèle soit utilisé en production. Ces cas font partie du cadre d'utilisation de GYSELA comme l'explique la section suivante.

6.2.3 Bilan et perspectives

Les opérations qui relèvent purement du calcul de la gyromoyenne, `deriv.` et `interp.`, ont une bonne scalabilité comme on peut le voir sur l'étude en *strong scaling*. Néanmoins, le calcul des dérivées étant sensible au nombre de points fantômes, on peut voir une forte évolution du temps d'exécution de ces calculs en fonction de la taille de cette zone fantôme sur l'étude en *pseudo weak scaling*. Concernant le coût des communications, il dépend bien entendu du nombre de points fantômes à communiquer et il doit être maîtrisé.

Dans le cadre de l'application GYSELA, les valeurs de ρ explorées recouvrent l'ensemble des utilisations de l'application. La taille du rayon de Larmor ρ est liée à la valeur de l'invariant adiabatique μ par la relation suivante :

$$\rho = \frac{\sqrt{2\mu}}{a}.$$

Le Tableau 6.4 montre les valeurs de μ qui correspondent aux valeurs de ρ qui ont été utilisées pour réaliser les études.

ρ	0.01	0.03	0.099
μ	0.5	4.5	49

TABLE 6.4 – Correspondance entre ρ et μ.

En pratique, les valeurs de μ utilisées pour les simulations varient classiquement dans l'intervalle $[0, 12]$. Le plus grand rayon que nous avons utilisé présente le comportement le plus pathologique pour la scalabilité de l'opérateur de gyromoyenne, mais il n'est jamais utilisé en pratique. Dans le cadre d'une étude de la gyromoyenne parallèle, cette valeur permet néanmoins d'identifier les comportements les plus limitants qui sont encore à améliorer.

Pour améliorer l'implémentation actuelle de la gyromoyenne parallèle, on peut envisager de calculer les tailles de zone fantôme processus par processus. Dans l'implémentation actuelle du prototype parallèle, l'ensemble des processus partage la même taille de zone fantôme, ce qui simplifie la régularité du schéma de communication comme il a été dit précédemment. Dans le cas où on calculerait de façon locale la taille de la zone fantôme relative à la position de la portion du plan propre à chaque processus, il serait alors possible de réduire globalement le volume de données échangé. Les processus qui sont proches du centre du plan poloïdal nécessitent les plus grandes zones fantômes, car plus on est proche du centre du plan, plus petite est la taille d'une cellule du plan ; pour un rayon ρ donné, un cercle d'intégration fait intervenir un plus grand nombre de cellules au centre du plan qu'à sa périphérie. Dans cette approche, les processus éloignés du centre ont une zone fantôme de plus petite taille et bénéficient donc de gains de

performances. Les gains de performances liés à cette approche n'est donc pas garantie car les volumes de communication sont déséquilibrés en fonction des processus.

L'implémentation présentée ici est de type *full*-MPI. Pour aller plus loin dans la parallélisation de l'opérateur de gyromoyenne, il sera nécessaire d'envisager l'approche hybride MPI/OpenMP. Dans cette approche, les calculs de dérivée et d'interpolation (`deriv.` et `interp.`) bénéficieraient d'une parallélisation en mémoire partagée via OpenMP. Avec cette solution, le plus grand cas en *weak scaling* avec *#proc* MPI = 1024 se traduirait par l'utilisation de 64 processus MPI avec 16 threads par processus. Ce déploiement permettrait de réduire la consommation mémoire globale de l'application, puisque le nombre de zones fantômes est moindre, et dans le même temps, de diminuer le volume global de communication MPI. Cette approche est un bon compromis entre les deux déploiements que nous avons utilisés pour réaliser les études de *strong* et *pseudo weak scaling*. Dans la première étude, un processus dispose de l'ensemble de la mémoire disponible sur un nœud, mais n'exploite qu'un seul cœur ; dans la seconde étude, toute la puissance d'un nœud est utilisée, mais les surcoûts mémoire liés aux zones fantômes sont importants ainsi que le coût des communications. L'approche hybride permettrait donc d'exploiter au mieux les ressources disponibles sur le nœud et de diminuer le volume de communication global.

6.3 Contraintes d'intégration

La section 6.1.3 présente l'algorithme de l'application parallèle que nous avons utilisé pour réaliser les études de performances dans ce chapitre. Nous avons vu que le nombre de points fantômes joue un rôle capital. Le calcul du nombre de points fantômes dans les directions (r, θ) est en pratique fait durant l'initialisation du module de gyromoyenne, à la ligne 2 de l'Algorithme 5. Ces dimensions déterminent les tailles de tableaux utilisés ainsi qu'une partie des calculs à réaliser pour effectuer la gyromoyenne d'un plan poloïdal en parallèle, d'où la nécessité de connaître au plus tôt les dimensions de la zone fantôme d'un processus.

Afin d'intégrer une version parallèle de l'opérateur de gyromoyenne à Gysela, il sera premièrement nécessaire de calculer les dimensions de la zone fantôme des processus tôt dans l'exécution de l'application. Deuxièmement les efforts de développement qu'il faudra mettre en œuvre pour intégrer une version parallèle dans le flux d'exécution de Gysela seront importants. Typiquement durant les diagnostiques, cette évolution permettrait de supprimer deux communications collectives liées aux transpositions de la fonction de distribution, mais nécessiterait la réécriture d'une partie du code parallélisée en OpenMP. Les modifications à apporter aux diagnostiques seront comparables aux efforts à fournir dans le solveur de Poisson. Un modification du schéma de parallélisation d'une application de l'envergure de Gysela est généralement un processus long. Néanmoins, ces modifications seront bénéfiques pour la scalabilité de l'application, mais aussi pour permettre de simuler la dynamique des électrons cinétiques.

6.4 Conclusion

Dans ce chapitre, nous avons abordé les points cruciaux du développement de la version parallèle de la gyromoyenne, à savoir sa décomposition de domaine, ainsi que le calcul des points fantômes. L'algorithme global de l'application prototype que nous avons utilisée pour réaliser les tests de performances de l'implémentation de la gyromoyenne parallèle actuelle est aussi donné.

Les tests de *strong scaling* montrent de bons résultats, la gyromoyenne parallèle a donc une bonne scalabilité. Le calcul des dérivées ainsi que le calcul des interpolations constituent les

deux noyaux de calcul de la gyromoyenne. Ils ont tous les deux un comportement favorable à la scalabilité de l'opérateur, ce qui est un résultat encourageant pour les futurs développements. L'étude en *pseudo weak scaling* nous a permis d'identifier des aspects limitants pour la scalabilité de la solution actuelle, et nous avons discuté de différentes modifications qui pourraient potentiellement remédier à ces limitations. Enfin, les contraintes liées au portage de l'opérateur parallèle dans GYSELA sont discutées.

Sur l'ensemble des tests réalisés, on peut noter que la taille maximale du plan poloïdal utilisé ($2^{14} \times 2^{14}$, soit 16384×16384) est très nettement au dessus des résolutions maximales utilisées (généralement $2^9 \times 2^9$, soit 512×512) quotidiennement par la communauté des utilisateurs de GYSELA. Les résultats de performances obtenus démontrent pleinement la bonne scalabilité globale de l'opérateur de gyromoyenne parallèle *full*-MPI sur des maillages poloïdaux très fins et sur un large intervalle de valeurs de ρ. Ces résultats sont prometteurs pour les futurs développements de cette solution. La prochaine évolution de l'opérateur de gyromoyenne consistera à utiliser une parallélisation hybride en MPI/OPENMP pour tirer profit de la totalité de la puissance de calcul d'un nœud de calcul, tout en réduisant le volume global de communications MPI par rapport à l'implémentation *full*-MPI actuelle.

Enfin, l'intégration à moyen terme d'une version parallèle de l'opérateur de gyromoyenne permettra de lever un verrou technique et d'améliorer la scalabilité globale de l'application, en particulier en supprimant des communications MPI collectives de type transposition, et en réduisant la quantité de structures de données limitantes pour la scalabilité mémoire. De plus, l'opérateur de gyromoyenne parallèle permettra d'augmenter la résolution du maillage dans le plan poloïdal et donc d'accéder à la simulation des électrons cinétiques. Le travail réalisé dans ce chapitre représente un réel pas en avant vers la simulation de phénomènes physiques plus riches, encore inexplorés par l'application GYSELA.

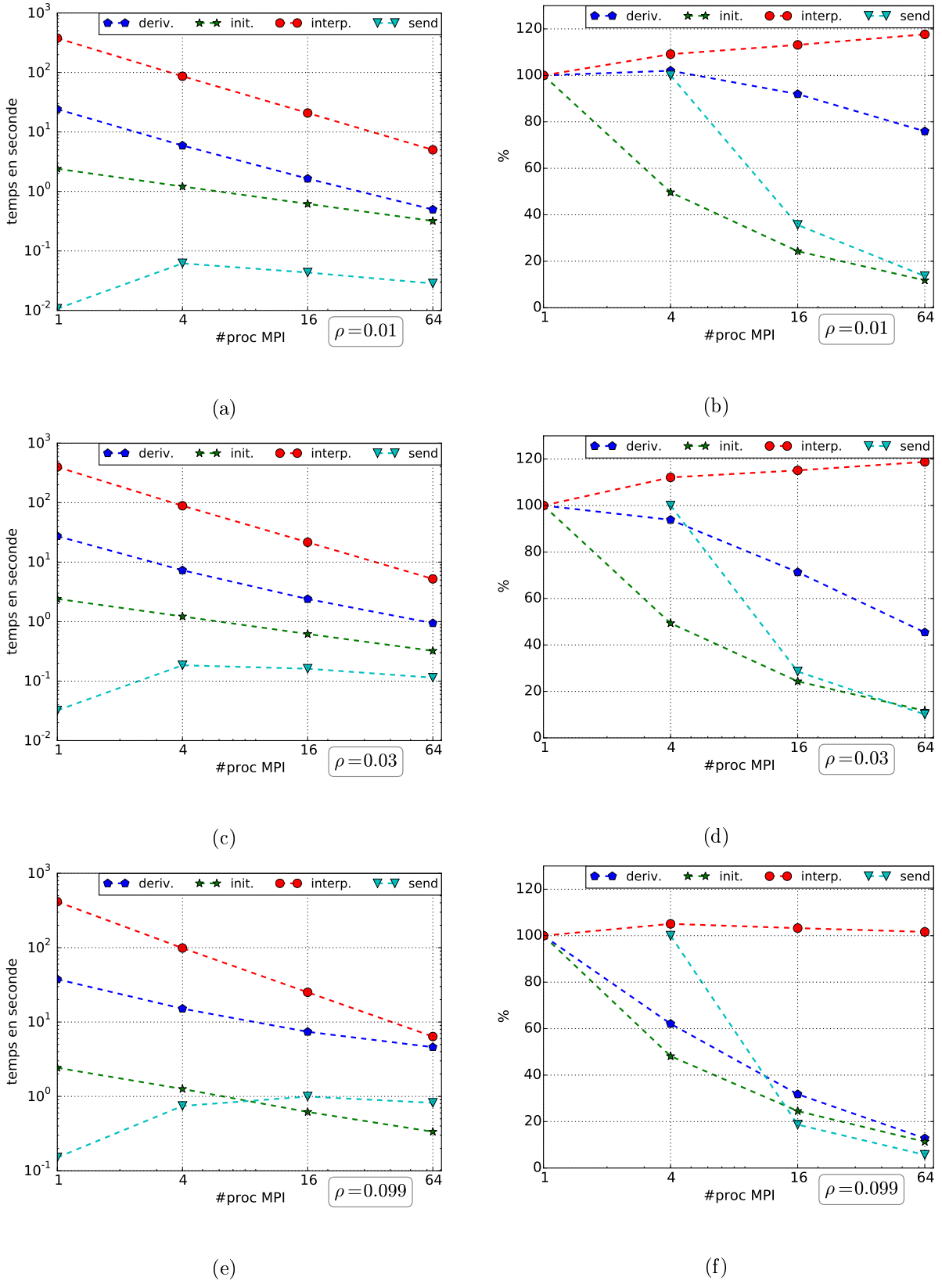
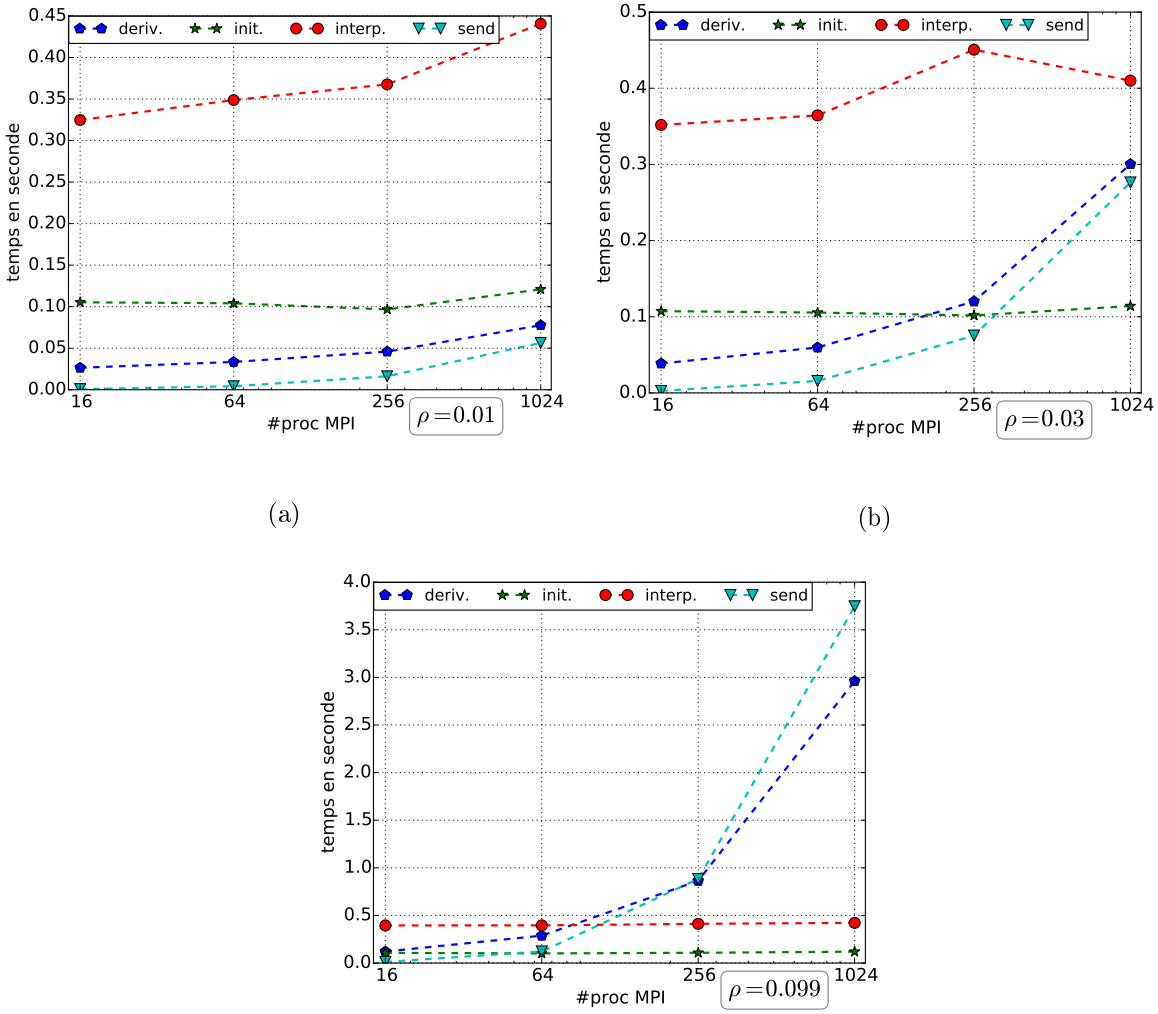


FIGURE 6.4 – Pour différentes valeurs du rayon de Larmor ρ, les Figures 6.4a, 6.4c, 6.4e représentent les temps d'exécution en secondes et les Figures 6.4b, 6.4d, 6.4f donnent les efficacités relatives en pourcentage en fonction du nombre de processus MPI.



(a) (b)

(c)

FIGURE 6.5 – Pour différentes tailles du rayon de Larmor ρ, les Figures 6.5a, 6.5b, 6.5c représentent les temps d'exécution en seconde en fonction du nombre de processus MPI.

Conclusion et perspectives

La simulation de turbulence de plasma est un outil essentiel pour la compréhension des phénomènes qui s'opèrent au sein d'un tokamak. L'utilisation efficace d'un supercalculateur par un code de simulation est un travail de longue haleine et nécessite l'utilisation d'indicateurs de performance et le développement d'outils d'analyse adaptés. L'amélioration de la scalabilité globale d'une application est une vaste problématique qui requiert souvent des collaborations interdisciplinaires. Dans cette problématique, les aspects informatiques, algorithmiques et relatifs aux mathématiques appliquées cohabitent. La première partie de la thèse décrit la libMTM qui est un outil informatique qui permet d'étudier et d'améliorer la scalabilité mémoire d'une application. Cette bibliothèque a été conçue de sorte à ce qu'elle soit utilisable dans d'autres applications, ce qui en fait un outil transverse aux disciplines qui utilisent la simulation numérique. Son utilisation en production dans l'application Gysela témoigne d'une certaine maturité. La seconde partie de ce manuscrit décrit les différentes solutions mises au point pour améliorer la méthode de calcul de l'opérateur de gyromoyenne, pour l'optimiser, l'intégrer et le comparer à l'opérateur existant. Une version parallèle de cette opérateur a aussi été développée et étudiée dans un programme prototype. Ce développement représente un travail amont et a pour objectif de préparer l'intégration d'une gyromoyenne parallèle dans Gysela.

Bilan

La première partie de ce manuscrit est consacrée à la description et l'utilisation de la bibliothèque libMTM qui permet d'étudier la consommation mémoire d'une application parallèle. Les outils qu'elle fournit sont un réel support pour l'étude de l'empreinte mémoire, pour sa réduction et sa prédiction. Bien que le choix d'instrumenter une application au niveau du code soit une méthode intrusive, c'est la seule à notre connaissance qui permette de récupérer les informations nécessaires pour la réalisation de prédictions mémoire précises. Nous avons vu dans le chapitre 3 différentes utilisations des outils de la libMTM. L'utilisation de l'outil de visualisation et les modifications incrémentales apportées à Gysela ont permis de réduire sa consommation mémoire, mais aussi d'améliorer sa scalabilité mémoire. Pour la plus grande simulation de notre jeu de tests, avec 32K cœurs, nous avons prédit et observé une réduction de la consommation mémoire de 50% par rapport à la version initiale. Le maillage utilisé pour réaliser les tests de strong scaling mémoire est de taille supérieure aux maillage utilisés en production ($N_r = 512$, $N_\theta = 1024$, $N_\varphi = 128$, $N_{v_\parallel} = 128$, $N_\mu = 2$). Mise à part la réduction effective de l'empreinte mémoire et l'amélioration de la scalabilité de Gysela, l'utilisation de la libMTM est bénéfique pour mieux comprendre le fonctionnement interne d'une grande application. La libMTM permet d'aborder une application du point de vue de sa consommation mémoire, ce qui est susceptible de donner des indices sur les modifications à effectuer afin d'améliorer les performances d'une application. Cette approche peut guider les améliorations à effectuer. Dans le contexte interdisciplinaire de la simulation HPC, cet outil est aussi pédagogique et peut

améliorer le dialogue entre les différentes communautés.

Dans un second temps, nous nous sommes concentrés sur l'opérateur de gyromoyenne qui est fondamental dans le modèle gyrocinétique. Initialement, l'opérateur existant était basé sur une approximation de Padé. Bien que son coût d'exécution soit faible, la méthode de calcul induite par cette approximation est peu précise et représente un frein pour la scalabilité de l'application. Pour pallier à ce défaut, nous avons développé une méthode de calcul alternative qui se base sur une intégration directe dans l'espace réel (versus l'utilisation de transformées de Fourier pour l'approximation de Padé) ; elle fait essentiellement intervenir des interpolations dans un plan poloïdal. Cette nouvelle méthode de calcul est plus précise que la précédente, mais en contrepartie elle est plus coûteuse en terme de temps de calcul. Afin de limiter les surcoûts, nous avons réalisé des optimisations. Dans un premier temps, nous avons rendu l'implémentation des fonctions de calcul de gyromoyenne *thread safe* ; il est donc possible d'appeler de façon concurrente les fonctions de calcul de gyromoyenne au sein d'une zone OpenMP. Ensuite, une description formelle de type produit matrice–vecteur de l'opérateur de gyromoyenne nous a permis d'identifier différentes approches d'optimisation qui ont été conçues pour favoriser les effets de cache. La meilleure optimisation réduit de près de 40% le temps d'exécution de l'opérateur non optimisé. Ce travail nous a permis d'obtenir un opérateur de gyromoyenne plus précis avec un coût de calcul similaire à l'opérateur existant. Dans un second temps, nous avons validé les résultats obtenus par ce nouvel opérateur sur le cas Cyclone qui est un cas de référence dans le domaine de la physique des plasmas. Enfin, nous avons développé une version de la gyromoyenne parallélisée en MPI. Grâce à cette implémentation, nous avons pu accéder à des résolutions du plan poloïdal (au maximum $2^{14} \times 2^{14}$, soit 16384×16384) qui sont bien supérieures aux résolutions actuellement utilisées en production (généralement $2^9 \times 2^9$, soit 512×512) par Gysela. Les études de scalabilité qui ont été réalisées démontrent la bonne scalabilité de l'opérateur de gyromoyenne pour les valeurs de μ utilisées par les simulations. Ce développement réalisé sur une application prototype représente un travail préparatoire pour l'intégration future d'une gyromoyenne parallèle dans Gysela. La parallélisation de cette opérateur permettra d'améliorer la scalabilité globale de Gysela en supprimant des communications MPI et représente un premier pas vers la simulation des électrons cinétiques.

Les deux parties distinctes de ce manuscrit contribuent à un objectif commun qui est l'amélioration de la scalabilité d'une application déjà massivement parallèle. Cet objectif a souvent nécessité d'intervenir à plusieurs niveaux afin de conserver une cohérence d'ensemble. Dans le cadre de cette thèse, la réduction de l'empreinte mémoire peut être vue comme une contribution de haut niveau alors que le changement de l'opérateur de gyromoyenne est une contribution spécifique à Gysela et proche de la physique.

Perspectives

Plusieurs extensions sont envisageables pour enrichir la libMTM. Un outil de visualisation plus interactif permettrait d'explorer les données contenues dans les trace `MTM` plus rapidement et d'augmenter de fait la fluidité des développements. De plus, à l'heure actuelle, le script de post-traitement exploite de façon partielle le contenu des traces. Il serait intéressant d'afficher plus d'informations telles que le nom de fichier et la ligne où se situe l'allocation ou la libération mémoire.

Pour aller plus loin dans l'analyse de la consommation mémoire, il serait intéressant de savoir si une zone mémoire est favorable ou non à la scalabilité d'une application. Dans notre cas, les données distribuées entre les différents processus MPI sont favorables à la scalabilité de l'application, alors que les quantités dont la consommation mémoire ne bénéficie pas de la

décomposition de domaine sont pénalisantes pour la scalabilité. Pour accéder à ce type d'analyse, le format de trace `MTM` et le script de post-traitement sont amenés à évoluer.

Dans le contexte de l'analyse de la consommation mémoire, une autre extension d'intérêt serait de coupler la libMTM avec une bibliothèque qui intercepte les appels aux fonctions primitives de gestion mémoire (allocation, libération, réallocation, alignement mémoire, etc). Ce mécanisme d'interception permettrait de mesurer les allocations mémoire qui ne sont pas directement dues à l'application elle-même, mais aux bibliothèques qu'elle utilise ; actuellement la libMTM se limite à l'enregistrement d'informations au niveau de l'application. En revanche, dans l'approche par interception, il est très difficile de concevoir une solution qui permettrait de récupérer suffisamment d'informations pour pouvoir réaliser la prédiction hors-ligne de la consommation mémoire. Cette approche est donc complémentaire à celle de la libMTM. De plus, elle s'adresse à une plus large classe d'applications que celles qui sont à même d'utiliser l'outil de prédiction. Dans le cadre de l'utilisation de supercalculateurs, les applications parallèles utilisent une pile logicielle qui leur permet de profiter des ressources (par exemple utilisation de plusieurs nœuds et/ou de plusieurs cœurs de calcul). Typiquement l'utilisation de bibliothèques qui implémentent les normes MPI et OpenMP est largement répandue dans la communauté HPC. Sur de grandes configurations, la mesure de la consommation mémoire de toutes ces briques logicielles peut faciliter l'interprétation du comportement d'une application [RPH+14].

Concernant l'opérateur de gyromoyenne, nous avons introduit dans la fin de ce manuscrit une version parallélisée en MPI. Pour poursuivre l'amélioration de la scalabilité de l'opérateur, l'utilisation d'un niveau de parallélisme supplémentaire en OpenMP est nécessaire.

La version parallélisée en MPI présentée dans le chapitre 6 adopte une décomposition de domaine par bloc $2D$ dans le plan poloïdal. Cette décomposition qui est inspirée de celle utilisée dans Gysela facilitera son intégration dans l'application. L'utilisation de cette version parallèle de la gyromoyenne constitue un premier pas vers la suppression de certaines communications MPI globales qui sont limitantes pour la scalabilité au delà de 128K cœurs en *strong scaling*. Par effet de bord, les structures temporaires qui limitent aujourd'hui la scalabilité mémoire de Gysela devraient bénéficier de la parallélisation de la gyromoyenne car la taille de ces tableaux est actuellement proportionnelle à $N_r \times N_\theta \times N_{threads}$. En bénéficiant d'une décomposition de domaine en (r, θ), ces structures ne devraient plus représenter de difficultés pour le passage à plus grande échelle de Gysela.

Enfin, la décomposition de domaine du plan poloïdal permettra d'utiliser de plus grands maillages poloïdaux. Cette évolution de Gysela est un cap majeur pour la simulation des électrons cinétiques car ils nécessitent l'utilisation de maillages poloïdaux plus fins que ceux utilisés actuellement. La simulation de la dynamique des électrons cinétiques permettra d'enrichir le modèle physique résolu et donc d'observer des phénomènes physiques plus riches, encore inexplorés par l'application Gysela.

Bibliographie

[Abi12] J. Abiteboul. *Transport turbulent et néoclassique de quantité de mouvement toroïdale dans les plasmas de tokamak*. PhD thesis, Aix-Marseille, 2012.

[AMGRT13] C. Aulagnon, D. Martin-Guillerez, F. Rué, and F. Trahay. Runtime function instrumentation with EZTrace. In *Euro-Par 2012 : Parallel Processing Workshops*, pages 395–403. Springer, 2013.

[AS+66] M. Abramowitz, I. A Stegun, et al. Handbook of mathematical functions. *Applied Mathematics Series*, 55 :62, 1966.

[ASAH12] R. B. Albadarneh, N. T. Shawagfeh, and Z. S. Abo Hammour. General (n+1)-explicit finite difference formulas with proof. *Applied Mathematical Sciences*, 2012.

[BEF+14] M. A. Bender, R. Ebrahimi, J. T. Fineman, G. Ghasemiesfeh, R. Johnson, and S. McCauley. Cache-adaptive algorithms. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 958–971. SIAM, 2014.

[BGA03] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *[CGO 2003]*, pages 265–275. IEEE, 2003.

[BGL+13] J. Bigot, V. Grandgirard, G. Latu, C. Passeron, F. Rozar, and O. Thomine. Scaling GYSELA code beyond 32k-cores on BlueGene/Q. In *ESAIM : Proceedings*, volume 43, pages 117–135. EDP Sciences, 2013.

[BH07] A. J. Brizard and T. S. Hahm. Foundations of nonlinear gyrokinetic theory. *Reviews of modern physics*, 79(2) :421, 2007.

[Bri89] A. Brizard. Nonlinear gyrokinetic Maxwell-Vlasov equations using magnetic coordinates. *Journal of plasma physics*, 41(03) :541–559, 1989.

[Bro08] I. Broemstrup. *Advanced Lagrangian Simulation Algorithms for Magnetized Plasma Turbulence*. PhD thesis, Univ. of Maryland, 2008.

[BSKM11] G. Barootkoob, M. Sharifi, E. M. Khaneghah, and S. L. Mirtaheri. Parameters affecting the functionality of memory allocators. In *Communication Software and Networks (ICCSN), 2011 IEEE 3rd International Conference on*, pages 499–503. IEEE, 2011.

[Car13] S. Carrez. Optimization with Valgrind Massif and Cachegrind, march 2013. website `http://blog.vacs.fr/vacs/blogs/post.html?post=2013/03/02/Optimization-with-Valgrind-Massif-and-Cachegrind`.

[CLS07] N. Crouseilles, G. Latu, and E. Sonnendrücker. Hermite spline interpolation on patches for parallelly solving the Vlasov-Poisson equation. *International Journal of Applied Mathematics and Computer Science*, 17(3) :335–349, 2007.

[CMS10] N. Crouseilles, M. Mehrenberger, and H. Sellama. Numerical solution of the gyroaverage operator for the finite gyroradius guiding-center model. *Communications in Computational Physics*, 8(3) :484, 2010.

[CQ13] C. Caldini-Queiros. *Analyse mathématique et numérique de modeles gyrocinétiques.* PhD thesis, Université de Franche-Comté, 2013.

[Da00] A. M. Dimits and al. Comparisons and physics basis of tokamak transport models and turbulence simulations. *Physics of Plasmas*, 7(3) :969–983, 2000.

[Dar05] O. Darrigol. *Les équations de Maxwell.* Belin, Paris, 2005.

[DBC+05] L. Djoudi, D. Barthou, P. Carribault, C. Lemuet, J. Acquaviva, W. Jalby, et al. MAQAO : Modular assembler quality analyzer and optimizer for itanium 2. In *The 4th Workshop on EPIC architectures and compiler technology, San Jose*, 2005.

[DE98] Leonardo Dagum and Rameshm Enon. OpenMP : an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1) :46–55, 1998.

[Dub06] F. Dubois. Interpolation de Hermite, august 2006. link http://www.math.u-psud.fr/~fdubois/cours/iacs/iacs-chap08.pdf.

[EMFB14] D. Elias, R. Matias, M. Fernandes, and L. Borges. Experimental and theoretical analyses of memory allocation algorithms. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pages 1545–1546. ACM, 2014.

[Eva06] J. Evans. A scalable concurrent malloc (3) implementation for FreeBSD. In *Proc. of the BSDCan Conference, Ottawa, Canada.* Citeseer, 2006.

[FBDR+98] 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.

[FCY99] M. Folk, A. Cheng, and K. Yates. HDF5 : A file format and I/O library for high performance computing applications. In *Proceedings of Supercomputing*, volume 99, 1999.

[Fin04] D. Finn. Ma 323 geometric modelling, course notes : Day 09, quintic Hermite interpolation, december 2004. Available online at https://www.rose-hulman.edu/~finn/CCLI/Notes/day09.pdf.

[FLPR99] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 285–297. IEEE, 1999.

[FMMA11] T. B. Ferreira, R. Matias, A. Macedo, and L. B. Araujo. An experimental study on memory allocators in multicore and multithreaded applications. In *Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2011 12th International Conference on*, pages 92–98. IEEE, 2011.

[GAB+15] 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.

[GBB+06] V. Grandgirard, M. Brunetti, P. Bertrand, N. Besse, X. Garbet, Ph. Ghendrih, G. Manfredi, Y. Sarazin, O. Sauter, E. Sonnendrücker, J. Vaclavik, and L. Villard. A drift-kinetic semi-Lagrangian 4D code for ion turbulence simulation. *Journal of Computational Physics*, 217(2) :395–423, 2006.

[GLB+11] 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.

[GLDS96] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel computing*, 22(6) :789–828, 1996.

[Glo06] W. Gloger. Ptmalloc, 2006. link http://www.malloc.de/en/.

[GM09] S. Ghemawat and P. Menage. Tcmalloc : Thread-caching malloc, 2009. link http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[gme02] GMemLogger - general memory logger, 2002. link http://www.nongnu.org/gmemlogger/.

[Gor04] M. Gorman. *Understanding the Linux virtual memory manager*. Prentice Hall Upper Saddle River, 2004.

[Gör09] T. Görler. Multiscale effects in plasma microturbulence. *PhD, Ulm*, 2009.

[gpe09] Gperftool - heap profiler, 2009. link http://goog-perftools.sourceforge.net/doc/heap_profiler.html.

[GR12] S. C. Gupta and K. Rangarajan. Optimizing heap memory usage, may 2012. US Patent App. 13/462,766.

[GSG+06] V. Grandgirard, Y. Sarazin, X. Garbet, G. Dif-Pradalier, P. Ghendrih, N. Crouseilles, G. Latu, E. Sonnendrücker, N. Besse, and P. Bertrand. GYSELA, a full-f global gyrokinetic semi-Lagrangian code for ITG turbulence simulations. In *AIP Conference Proceedings*, volume 871, page 100. IOP INSTITUTE OF PHYSICS PUBLISHING LTD, 2006.

[GSG+08] V. Grandgirard, Y. Sarazin, X. Garbet, G. Dif-Pradalier, P. Ghendrih, N. Crouseilles, G. Latu, E. Sonnendrücker, N. Besse, and P. Bertrand. Computing ITG turbulence with a full-f semi-Lagrangian code. *Communications in Nonlinear Science and Numerical Simulation*, 13(1) :81–87, 2008.

[GWW+10] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr. The SCALASCA performance toolset architecture. *CCPE*, 22(6) :702–719, 2010.

[Hah88] T. S. Hahm. Nonlinear gyrokinetic equations for tokamak microturbulence. *Physics of Fluids (1958-1988)*, 31(9) :2670–2673, 1988.

[HJ91] R. Hastings and B. Joyce. Purify : Fast detection of memory leaks and access errors. In *In Proc. of the Winter 1992 USENIX Conference*. Citeseer, 1991.

[HKM07] R. Hatzky, A. Könies, and A. Mishchenko. Electromagnetic gyrokinetic PIC simulation with an adjustable control variates method. *Journal of Computational Physics*, 225(1) :568–590, 2007.

[HLP+11] M. Hall, R. Lethin, K. Pingali, D. Quinlan, V. Sarkar, J. Shalf, R. Lucas, K. Yelick, et al. Ascr programming challenges for exascale computing. 2011.

[HTK+02] R. Hatzky, T. M. Tran, A. Könies, R. Kleiber, and S. J. Allfrey. Energy conservation in a nonlinear gyrokinetic particle-in-cell code for ion-temperature-gradient-driven modes in θ-pinch geometry. *Physics of Plasmas (1994-present)*, 9(3) :898–912, 2002.

[Hun07] J. D. Hunter. Matplotlib : A 2d graphics environment. *Computing In Science & Engineering*, 9(3) :90–95, 2007.

[IIK+08] Y. Idomura, M. Ida, T. Kano, N. Aiba, and S. Tokuda. Conservative global gyrokinetic toroidal full-f five-dimensional Vlasov simulation. *Computer Physics Communications*, 179(6) :391–403, 2008.

[ITK03] Y. Idomura, S. Tokuda, and Y. Kishimoto. Global gyrokinetic simulation of ion temperature gradient driven turbulence in plasmas using a canonical maxwellian distribution. *Nuclear Fusion*, 43(4) :234, 2003.

[JBA+07] S. Jolliet, A. Bottino, P. Angelino, R. Hatzky, T. M. Tran, B. F. McMillan, O. Sauter, K. Appert, Y. Idomura, and L. Villard. A global collisionless pic code in magnetic coordinates. *Computer Physics Communications*, 177(5) :409–425, 2007.

[Kae01] M. Kaempf. Vudo malloc tricks. phrack magazine, 2001. link `http://phrack.org/issues/57/8.html#article`.

[Kna05] A. W. Knapp. Fourier transform in euclidean space. In *Basic Real Analysis*, Cornerstones, pages 373–408. Birkhäuser Boston, 2005.

[KPE+12] S. Kunkel, T. C Potjans, J. M. Eppler, H. E. Plesser, A. Morrison, and M. Diesmann. Meeting the memory challenges of brain-scale network simulation. *Frontiers in neuroinformatics*, 5 :35, 2012.

[Kre12] M. Kreh. Bessel functions. *Lecture Notes, Penn State-Göttingen Summer School on Number Theory*, 2012.

[KS13] P. Kogge and J. Shalf. Exascale computing trends : Adjusting to the. *Computing in Science & Engineering*, 15(6) :16–26, 2013.

[LB00] C. Lever and D. Boreham. Malloc() performance in a multithreaded linux environment. *http ://citi.umich.edu/projects/linux-scalability/reports/malloc.html*, 2000.

[LCM+05] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin : building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Notices*, volume 40, pages 190–200. ACM, 2005.

[Lee87] W. W. Lee. Gyrokinetic particle simulation model. *Journal of Computational Physics*, 72(1) :243–269, 1987.

[LFN02] R. E. Ladner, R. Fortna, and B. Nguyen. A comparison of cache aware and cache oblivious static search trees using program instrumentation. In *Experimental Algorithmics*, pages 78–92. Springer, 2002.

[LG96] D. Lea and W. Gloger. A memory allocator, 1996. link `http://gee.cs.oswego.edu/dl/html/malloc.html`.

[LGCDP11] 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.

[LHB+15] G. Latu, M. Haefele, J. Bigot, V. Grandgirard, T. Cartier-Michaud, and F. Rozar. Evaluating kernels on xeon phi to accelerate GYSELA application. *CoRR*,

abs/1503.04645, 2015. Accepted to ESAIM proceedings for CEMRACS 2014 summer school. Available online at http://arxiv.org/pdf/1503.04645.pdf.

[Li05] J. Li. General explicit difference formulas for numerical differentiation. *Journal of Computational and Applied Mathematics*, 183(1) :29–52, 2005.

[Lit88] R. G. Littlejohn. Phase anholonomy in the classical adiabatic motion of charged particles. *Physical Review A*, 38(12) :6034, 1988.

[LL95] Z. Lin and W. W. Lee. Method for solving the gyrokinetic poisson equation in general geometry. *Physical Review E*, 52(5) :5646, 1995.

[log14a] log-malloc-simple - memory allocation tracking library, 2014. link https://github.com/qgears/log-malloc-simple.

[log14b] log-malloc2 - memory allocation tracking library, 2014. link http://devel.dob.sk/log-malloc2/.

[Max65] J. C. Maxwell. A dynamical theory of the electromagnetic field. *Philosophical Transactions of the Royal Society of London*, pages 459–512, 1865.

[Mil14] W. Milian. Heaptrack - a heap memory profiler for linux. *http ://milianw.de/blog/heaptrack-a-heap-memory-profiler-for-linux*, 2014.

[NS07] N. Nethercote and J. Seward. Valgrind : a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan Notices*, volume 42, pages 89–100. ACM, 2007.

[Nys14] R. Nystrom. *Game programming patterns*. Genever Benning, 2014. link http://gameprogrammingpatterns.com/data-locality.html.

[RLR14] F. Rozar, G. Latu, and J. Roman. Achieving memory scalability in the GYSELA code to fit exascale constraints. In Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Waśniewski, editors, *Parallel Processing and Applied Mathematics*, LNCS 8385, pages 185–195. Springer Berlin Heidelberg, 2014.

[RLRG15] F. Rozar, G. Latu, J. Roman, and V. Grandgirard. Toward memory scalability of GYSELA code for extreme scale computers. *Concurrency and Computation : Practice and Experience*, 27(4) :994–1009, March 2015. Special issue PPAM'13. Available online at http://onlinelibrary.wiley.com/doi/10.1002/cpe.3429/abstract.

[Roz14] F. Rozar. Amélioration de la scalabilité mémoire du code GYSELA. 2014. Available online at https://hal.inria.fr/hal-01111722.

[Roz15] F. Rozar. Passage à l'Échelle mémoire et impact des allocations dynamiques dans le code GYSELA. January 2015. Accepté à TSI (numéro spécial COMPAS'14). Disponible en ligne à https://hal.inria.fr/hal-01176700.

[RPH+14] Raghunath Rajachandrasekar, Jonathan Perkins, Khaled Hamidouche, Mark Arnold, and Dhabaleswar K Panda. Understanding the memory-utilization of MPI libraries : Challenges and designs in implementing the mpi_t interface. In *Proceedings of the 21st European MPI Users' Group Meeting*, page 97. ACM, 2014.

[RR97] C. Runciman and N. Röjemo. Two-pass heap profiling : A matter of life and death. In *Implementation of Functional Languages*, pages 222–232. Springer, 1997.

[RSL+15] 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. In *ESAIM : Proceedings*, Luminy, France, 2015. EDP Sciences. Accepted to ESAIM proceedings for CEMRACS 2014 summer school.

[SGF+05] 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.

[SMC+15] 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.

[Son10] E. Sonnendrücker. Approximations numériques des équations Vlasov-Maxwell. *Notes du Cours M*, 2, 2010.

[Str68] G. Strang. On the construction and comparison of difference schemes. *SIAM Journal on Numerical Analysis*, 5(3) :506–517, 1968.

[TaMS+08] C. Terboven, D. an Mey, D. Schmidl, H. Jin, and T. Reichstein. Data and thread affinity in OpenMP programs. In *Proceedings of the 2008 Workshop on Memory Access on Future Processors : A Solved Problem ?*, MAW '08, pages 377–384, New York, NY, USA, 2008. ACM.

[TBG+13] O. Thomine, J. Bigot, V. Grandgirard, G. Latu, C. Passeron, and F. Rozar. An asynchronous writing method for restart files in the GYSELA code in prevision of exascale systems. In *ESAIM : Proceedings*, volume 43, pages 108–116, Luminy, France, 2013. EDP Sciences. Available online at https://hal.inria.fr/hal-01048745/document.

[Tor09] S. Tordeux. Fonctions de plusieurs variables, 2009. link http://stordeux.perso.univ-pau.fr/COURS/ICBE2UV2.pdf.

[TT11] B. M. Tudor and Y. M. Teo. A practical approach for performance analysis of shared-memory programs. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 652–663. IEEE, 2011.

[Val14] S. Valat. *Contribution à l'amélioration des méthodes d'optimisation de la gestion de la mémoire dans le cadre du calcul haute performance.* PhD thesis, Versailles-St Quentin en Yvelines, 2014.

[Vla45] A Vlasov. On the kinetic theory of an assembly of particles with collective interaction. *Russ. Phys. J.*, 9 :25–40, 1945.

[Wei] E. W. Weisstein. Bessel function of the first kind. Visited on 15/05/10. link http://mathworld.wolfram.com/BesselFunctionoftheFirstKind.html.

[WL91] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. *SIGPLAN Not.*, 26(6) :30–44, May 1991.

[WWWG13] S. Widmer, D. Wodniok, N. Weber, and M. Goesele. Fast dynamic memory allocator for massively parallel architectures. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, pages 120–126. ACM, 2013.

[YLPM05] S. Yoon, P. Lindstrom, V. Pascucci, and D. Manocha. Cache-oblivious mesh layouts. In *ACM Transactions on Graphics (TOG)*, volume 24, pages 886–893. ACM, 2005.

[ZH88] B. Zorn and P. Hilfinger. A memory allocation profiler for c and lisp programs. In *Proceedings of the Summer USENIX Conference*, pages 223–237, 1988.

Liste des publications

Revues :

[RLRG15] F. Rozar, G. Latu, J. Roman, and V. Grandgirard. Toward memory scalability of gysela code for extreme scale computers. *Concurrency and Computation : Practice and Experience*, 27(4) :994–1009, March 2015. Special issue PPAM'13. Available online at http://onlinelibrary.wiley.com/doi/10.1002/cpe.3429/abstract.

[Roz15] F. Rozar. Passage à l'Échelle mémoire et impact des allocations dynamiques dans le code gysela. January 2015. Accepté à TSI (numéro spécial COMPAS'14). Disponible en ligne à https://hal.inria.fr/hal-01176700.

[SMC+15] 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.

Proceedings en conférence internationale :

[RLR14] F. Rozar, G. Latu, and J. Roman. Achieving memory scalability in the GYSELA code to fit exascale constraints. In Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Waśniewski, editors, *Parallel Processing and Applied Mathematics*, LNCS 8385, pages 185–195. Springer Berlin Heidelberg, 2014.

Proceedings :

[TBG+13] O. Thomine, J. Bigot, V. Grandgirard, G. Latu, C. Passeron, and F. Rozar. An asynchronous writing method for restart files in the gysela code in prevision of exascale systems. In *ESAIM : Proceedings*, volume 43, pages 108–116, Luminy, France, 2013. EDP Sciences. Available online at https://hal.inria.fr/hal-01048745/document.

[BGL+13] J. Bigot, V. Grandgirard, G. Latu, C. Passeron, F. Rozar, and O. Thomine. Scaling gysela code beyond 32k-cores on BlueGene/Q. In *ESAIM : Proceedings*, volume 43, pages 117–135. EDP Sciences, 2013.

[LHB+15] G. Latu, M. Haefele, J. Bigot, V. Grandgirard, T. Cartier-Michaud, and F. Rozar. Evaluating kernels on xeon phi to accelerate gysela application. *CoRR*, abs/1503.04645, 2015. Accepted to ESAIM proceedings for CEMRACS 2014 summer school. Available online at http://arxiv.org/pdf/1503.04645.pdf.

[RSL+15] 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. In *ESAIM : Proceedings*, Luminy, France, 2015. EDP Sciences. Accepted to ESAIM proceedings for CEMRACS 2014 summer school.

Autres :

[Roz14] F. Rozar. Amélioration de la scalabilité mémoire du code gysela. ComPAS'14. Available online at https://hal.inria.fr/hal-01111722.

[GAB+15] 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.

Annexes

Annexe A

Contraintes de conception de la libMTM

A.1 Utilisable dans différents langages

Lors de la conception de la bibliothèque libMTM, nous nous sommes fixés la contrainte qu'elle soit utilisable en Fortran et en langage C pour pouvoir l'utiliser à terme dans différentes applications. Pour ce faire, il existe classiquement 2 approches. Soit la bibliothèque est portée dans différents langages, soit une implémentation de base est réalisée et une interface par langage cible est créée.

La solution de portage consiste à implémenter dans différents langages l'API de la bibliothèque libMTM. Dans notre cas, cela signifie que pour chaque fonction décrite dans la section 2.2.2,p. 27, nous aurions réalisé une implémentation en Fortran et une implémentation en C. Cette approche permet de profiter des atouts des différents langages. Par exemple en Fortran, les fonctions `size()` et `kind()` permettent respectivement d'interroger un tableau pour récupérer son nombre d'éléments et la consommation mémoire associée à un élément. De façon analogue, le langage C donne accès à des fonctionnalités indisponibles en Fortran comme par exemple l'opérateur `&` qui permet de récupérer la référence d'une variable, ou encore la possibilité d'utiliser le type générique `void *`. En tirant profit des différentes particularités des langages, le fonctionnement interne de la bibliothèque peut être simplifié dans certains langages. Le point faible du portage d'une bibliothèque dans différents langages vient de la difficulté à maintenir l'ensemble des implémentations. Dans le cas où une erreur de conception a été commise, cette dernière doit être corrigée pour chaque implémentation. Ce travail est généralement fastidieux et source de désynchronisation entre les différentes implémentations.

L'approche alternative qui consiste à avoir une implémentation de base et une interface différente par langage cible est meilleure d'un point de vue génie logiciel [22]. Le cœur de métier de la bibliothèque étant implémenté à un seul endroit, la correction de bogue nécessite des modifications localisées dans le code source. De plus, cela simplifie l'ajout de fonctionnalité, ou plus généralement, les évolutions de la bibliothèque. Par contre, il n'est pas possible dans cette solution de profiter de fonctionnalités spécifiques proposées par un langage. L'implémentation de base qui constitue le cœur de la solution est de faite assez simple car elle utilise uniquement des fonctionnalités communes à l'ensemble des langages gérés, dans notre cas Fortran90 et C. C'est l'approche que nous avons choisie pour le développement de la libMTM.

22. Selon `http://www.larousse.fr` : ensemble des méthodes et des procédures mises en œuvre dans les différentes phases de la production d'un logiciel afin d'en améliorer les qualités et la maintenance.

A.2 Utilisable sur différentes machines

Le premier objectif de la LIBMTM était d'être utilisé par GYSELA pour étudier ses aspects consommations mémoires. GYSELA s'exécute sur plusieurs machines qui utilisent différents compilateurs (par exemple `ifort`, `gfortran` ou `bgxlf90_r`). L'utilisation de la LIBMTM est possible sur une machine donnée à condition qu'on puisse compiler et exécuter le code de la bibliothèque. Cet aspect portabilité représente une contrainte forte dans le développement de la bibliothèque.

Premièrement, le fait de changer régulièrement de machine n'encourage pas l'utilisation de bibliothèques externes. Une fonctionnalité offerte par un code de tierce partie peut accélérer le développement d'un code. Néanmoins, l'utilisation de cette fonctionnalité introduit naturellement une dépendance avec la bibliothèque qui la fournit et représente de fait une prise de risque qui dépend de la maturité de la bibliothèque externe. Dans le cas où le code externe est stable, utilisable sur différents types de machine, avec différents compilateurs, on peut avoir confiance en ce code et espérer que l'utilisation qu'on en fait ne va pas mettre en évidence un bogue de ce code externe. Dans le cas contraire, on risque d'être confronté à des problèmes de portabilité dû à ce code externe sur une des machines qu'on souhaite utiliser. Ce cas de figure n'est pas souhaitable car cela rendrait inutilisable la LIBMTM par effet collatéral. C'est pourquoi les dépendances de la LIBMTM par rapport à des codes externes sont réduites au strict minimum.

La seconde difficulté vient du fait que la LIBMTM fournit un module FORTRAN à l'utilisateur. En FORTRAN, l'usage des modules via la commande `use` sont conceptuellement équivalents à l'utilisation de fichiers d'entête en langage C. Ils sont utiles pour la vérification de type au niveau de la compilation. Les appels de fonction dont le type des arguments ne correspond pas au prototype déclaré de la fonction sont signalés à l'utilisateur. Néanmoins la différence fondamentale entre le FORTRAN et le C sur ce point concerne la gestion de ce fichier d'entête. En C, ce fichier fait partie du code standard manipulé par le développeur. En FORTRAN, le fichier de module (par exemple `mtm_alloc.mod`) est généré par le compilateur. Le format d'un fichier `.mod` dépend du compilateur et souvent aussi de la version du compilateur car il n'est pas explicitement spécifié dans la norme FORTRAN. Il n'est pas toujours indépendant de l'architecture. Enfin, ce fichier peut être écrit en ASCII ou en binaire selon le compilateur utilisé [23]. Face à l'ensemble de ces difficultés, l'installation de la bibliothèque LIBMTM sur les machines utilisées par GYSELA est difficile car pour chaque machine, il faudrait autant de dossier d'installation que de compilateurs et de versions de compilateur disponibles sur la machine. Afin d'éviter ces difficultés, les sources de la LIBMTM sont incluses dans le projet GYSELA et elles sont compilées au même moment que l'application GYSELA avec le même compilateur et les mêmes options de compilation.

La troisième difficulté vient des appels de fonctions C à partir du langage FORTRAN. Il est possible de s'appuyer sur les noms de symboles générés par le compilateur pour appeler des fonctions C à partir d'un code FORTRAN. Actuellement, il n'y a pas de standard qui dicte les règles de générations de nom de symboles. Ces noms dépendent donc du compilateur utilisé. En FORTRAN, le module `ISO_C_BINDING` permet de pallier à ce problème. Il propose une procédure standard d'appel de fonction C en FORTRAN et inversement. Néanmoins ce module étant relativement récent, il peut être indisponible sur des machines qui n'ont pas bénéficié de mises à jour récentes du compilateur. Dans l'état actuel de la LIBMTM, le module `ISO_C_BINDING` n'est donc pas utilisé pour assurer une meilleure portabilité.

23. Une discussion plus générale sur la gestion des modules FORTRAN dans LINUX est donnée https://bugs.linuxfoundation.org/show_bug.cgi?id=757

Annexe B

Exemples avancés d'utilisation de la libMTM

Cette section technique met en avant différentes situations où l'instrumentation à mettre en place pour assurer une prédiction de la consommation mémoire juste est délicate à réaliser. Deux exemples qui illustrent ce genre de situation sont présentés dans cette annexe. Pour chaque exemple, un extrait du code source et la trace générée sont fournis. La difficulté qui est mise en avant dans chaque exemple est discutée et commentée.

B.1 Allocations sensibles aux paramètres de fonction

```
1  subroutine alloc_array( a, n )
2    integer, dimension(:), pointer :: a
3    integer :: n
4    integer :: color
5
6    color = 1
7    MTM_ALLOC_1_ZERO( a, 1, n, color )
8
9  end subroutine alloc_array
10
11 program hello
12   ! [ ... ]
13   integer :: Nr, Ntheta
14   integer, dimension(:), &
15       pointer :: tab_r, tab_theta
16
17   Nr     = 10
18   Ntheta = 100
19   MTM_PARAM_SCALAR_RECORD(Nr, MTM_I32)
20   MTM_PARAM_SCALAR_RECORD(Ntheta, MTM_I32)
21
22   MTM_EXPR_RECORD( n, Nr, MTM_I32 )
23   call alloc_array( tab_r, Nr )
24   MTM_EXPR_RECORD( n, Ntheta, MTM_I32 )
25   call alloc_array( tab_theta, Ntheta )
26
27   ! [ ... ]
28 end program hello
```

Listing B.1 – Extrait du code source de l'exemple 1.

```
1  # Processus : 0
2
3  BEGIN hello
4  ARRAY 0 Nr  1 10    avance_1.F90  44
5  ARRAY 0 Ntheta  1 100   avance_1.F90  45
6  EXPR  0 n Nr  avance_1.F90  47
7  ALLOC  a  0x1fd6250 40  (n-1+1) 4 1 avance_1.F90  13
8  EXPR  0 n Ntheta  avance_1.F90  49
9  ALLOC  a  0x1fd6300 400 (n-1+1) 4 1 avance_1.F90  13
10 FREE  0x1fd6250 avance_1.F90  52
11 FREE  0x1fd6300 avance_1.F90  53
12 END hello
```

Listing B.2 – Trace `MTM` associée à l'exemple 1.

Dans cet exemple, l'extrait de programme présenté par le Listing B.1 utilise une *subroutine* `alloc_array()` pour allouer et initialiser à 0 un pointeur `a` passé en argument. Cette *subroutine* réalise une allocation de taille `n` du pointeur `a` ; `n` étant un paramètre d'entrée de cette *subroutine*. Elle est utilisée sur deux pointeurs différents avec une valeur de `n` différente pour chaque appel.

Dans cette situation, pour assurer une prédiction correcte de la consommation mémoire avec le format de trace actuel, il est nécessaire de renseigner des informations concernant le paramètre `n` juste avant l'appel de la *subroutine* `alloc_array()`. Comme on peut le voir sur le Listing B.1, la valeur du paramètre `n` est renseignée juste avant l'appel de la fonction aux lignes 22 et 24. Il n'est pas possible avec l'instrumentation actuelle de connaître les paramètres d'appel de la fonction dans le corps de la *subroutine* `alloc_array()`. Sur le Listing B.2, les informations enregistrées concernant le paramètre `n` se retrouvent aux lignes 6 et 8. Elles sont nécessaires pour la prédiction hors-ligne soit correcte par rapport aux valeurs des paramètres `Nr` et `Ntheta`.

Pour chaque fonction qui alloue des structures donc la taille dépend des paramètres d'entrée, ce type d'instrumentation est à réaliser. Ceci est nécessaire pour pouvoir relier la taille d'une allocation aux variables dont elle dépend. La solution proposée ici qui peut être non triviale à mettre en place, permet de gérer la dépendance par rapport aux paramètres d'entrée. Nous avons eu recours à ce genre de solution car actuellement, les notions de fonction et de portée lexicographique des variables ne sont pas prises en compte dans le format de trace et dans le script de post-traitement.

Les changements à mettre en œuvre pour remédier à cette difficulté sont toujours en cours de discussion à l'heure actuelle.

B.2 Allocations sensibles à l'environnement d'exécution

```
1  #include "mtm_alloc.h"
2
3  program hello
4
5    use mtm_alloc
6    implicit None
7
8    include 'mpif.h'
9
10   integer :: numtasks, rank, pid_root, ierr
11   integer :: nb_proc_r, nb_proc_theta
12   integer, dimension(:), pointer :: tab
13
14   call MPI_INIT(ierr)
15   call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
```

```
16   call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)
17
18   pid_root       = 0
19
20   ! Read from an input file
21   nb_proc_r     = read_nb_proc_r()
22   nb_proc_theta = read_nb_proc_theta()
23
24   MTM_INIT(rank, pid_root)
25
26   MTM_BLOCK_ENTER("hello")
27
28   MTM_PARAM_SCALAR_RECORD(nb_proc_r    , MTM_I32)
29   MTM_PARAM_SCALAR_RECORD(nb_proc_theta, MTM_I32)
30
31   MTM_EXPR_RECORD( numtasks, nb_proc_r * nb_proc_theta, MTM_I32 )
32   MTM_ALLOC_1_ZERO( tab, 1, numtasks, 1 )
33
34   MTM_DEALLOC(tab)
35
36   MTM_BLOCK_EXIT("hello")
37
38   MTM_FINALIZE()
39
40   call MPI_FINALIZE(ierr)
41
42 end program hello
```

Listing B.3 – Extrait du code source de l'exemple 1.

```
1  # Processus : 0
2
3  BEGIN hello
4  ARRAY 0 nb_proc_r     1 2   avance_2.F90  26
5  ARRAY 0 nb_proc_theta 1 2   avance_2.F90  27
6  EXPR  0 numtasks  nb_proc_r*nb_proc_theta avance_2.F90  29
7  ALLOC  tab  0x18713e0 16  (numtasks-1+1)  4 1 avance_2.F90  30
8  FREE  0x18713e0 avance_2.F90  32
9  END hello
10
11 # End
```

Listing B.4 – Trace `MTM` associée à l'exemple 1.

Le second cas de figure présenté ici met en avant une difficulté lié aux allocations qui dépendent de l'environnement d'exécution. Dans l'extrait de code source fournit par le Listing B.3, à la ligne 32 on voit une allocation du pointeur `tab` qui dépend du paramètre `numtasks`. La valeur de ce paramètre dépend du contexte d'exécution car, comme on peut le voir à la ligne 16, il est égal au nombre de processus MPI utilisés par le programme.

Dans le cas présenté ici, nous avons imaginé que le nombre de processus utilisés est égale au produit du nombre de processus dans la direction r et du nombre de processus dans la direction θ, soit $nb\_proc\_r \times nb\_proc\_theta$. Dans ce cas, il est possible de renseigner ces valeurs et l'expression du paramètre `numtasks` (lignes 28, 29 et 31) afin d'assurer une prédiction de la consommation mémoire correcte. C'est typiquement le genre de situation où le développeur doit intervenir pour instrumenter correctement l'application, car il est le seul à connaître la relation, si elle existe, entre le nombre de processus utilisés et les variables du programme. Les informations enregistrées dans la trace doivent refléter le comportement du programme pour que la prédiction

de la consommation mémoire hors-ligne soit correcte. L'ensemble des allocations sensibles à l'environnement d'exécution doit être soigneusement instrumenté.

Détecter ce type de situation peut être difficile. La mise en place de tests qui suivent le protocole donné dans la section 2.3.3, p. 37 permet de faciliter la détection de ce type de situation.

Annexe C

Expression du polynôme d'Hermite en $1D$ et $2D$

C.1 Expression du polynôme d'Hermite en $1D$

On cherche le polynôme $P_i(\alpha) = a_3\alpha^3 + a_2\alpha^2 + a_1\alpha + a_0$ qui interpole la fonction g sur l'intervalle $[0, 1]$. Il respecte les contraintes :

$$\begin{aligned} P_i(0) &= g(0), & P_i(1) &= g(1), \\ P_i'(0) &= g'(0), & P_i'(1) &= g'(1). \end{aligned}$$

De ces contraintes, on déduit :

$$\begin{aligned} g(0) &= a_0, \\ g(1) &= a_3 + a_2 + a_1 + a_0, \\ g'(0) &= a_1, \\ g'(1) &= 3a_3 + 2a_2 + a_1. \end{aligned}$$

D'où :

$$\begin{aligned} a_0 &= g(0), \\ a_1 &= g'(0), \\ a_2 &= -2g'(0) - g'(1) - 3g(0) + 3g(1), \\ a_3 &= g'(0) + g'(1) + g(0) - 2g(1). \end{aligned}$$

On peut maintenant exprimer le polynôme P_i sous la forme suivante :

$$\begin{aligned} P_i(\alpha) = (2\alpha^3 - 3\alpha^2 + 1)g(0) + (-2\alpha^3 + 3\alpha^2)g(1) + \\ (\alpha^3 - 2\alpha^2 + \alpha)g'(0) + (\alpha^3 - \alpha^2)g'(1). \end{aligned}$$

On souhaite exprimer P_i en fonction de f. La variable x étant liée à α par (4.11), on étudie la dérivée de g par rapport à α, $g'(\alpha)$.

Soit :

$$g'(\alpha) = \frac{\mathrm{d}}{\mathrm{d}\alpha}\left(f(x)\right) = \frac{\mathrm{d}x}{\mathrm{d}\alpha}\frac{\mathrm{d}}{\mathrm{d}x}f(x) = \Delta x f'(x).$$

D'où :

$$\begin{aligned} g'(0) =& \Delta x f'(x_i), \\ g'(1) =& \Delta x f'(x_{i+1}). \end{aligned}$$

Donc l'expression de P_i à l'aide de fonctions de base s'écrit :

$$\begin{aligned} P_i(\alpha) = & \varphi_1(\alpha) f(x_i) + \varphi_2(\alpha) f(x_{i+1}) + \\ & \Delta x(\varphi_3(\alpha) f'(x_i) + \varphi_4(\alpha) f'(x_{i+1})). \end{aligned} \tag{C.1}$$

$$\text{avec} \begin{cases} \varphi_1(\alpha) = & 2\alpha^3 - 3\alpha^2 + 1 \\ \varphi_2(\alpha) = & -2\alpha^3 + 3\alpha^2 \\ \varphi_3(\alpha) = & \alpha^3 - 2\alpha^2 + \alpha \\ \varphi_4(\alpha) = & \alpha^3 - \alpha^2 \end{cases}$$

C.2 Expression du polynôme d'Hermite en $2D$

Afin de présenter le détail des calculs du polynôme $P_{i,j}$, considérons la fonction auxiliaire g définie sur le carré unitaire $[0,1] \times [0,1] \subset \mathbb{R}^2$ telle que :

$$g(\alpha, \beta) = f(r, \theta).$$

et

$$r = r_i + \alpha \Delta r, \qquad r \in [r_i, r_{i+1}] \tag{C.2}$$

$$\theta = \theta_j + \beta \Delta \theta, \qquad \theta \in [\theta_j, \theta_{j+1}] \tag{C.3}$$

On cherche le polynôme $P_{i,j}(\alpha, \beta)$ qui interpole la fonction g sur le carré $[0,1] \times [0,1]$. On peut le calculer comme étant interpolation suivant la direction β de 4 fonctions de bases calculées à partir de g :

$$\begin{aligned} P_{i,j}(\alpha, \beta) = & \varphi_1(\beta) g(\alpha, 0) + \varphi_2(\beta) g(\alpha, 1) + \\ & \varphi_3(\beta) \frac{\partial}{\partial \beta} g(\alpha, 0) + \varphi_4(\beta) \frac{\partial}{\partial \beta} g(\alpha, 1). \end{aligned}$$

Ici, la valeur des fonctions de base $g(\alpha, 0)$, $g(\alpha, 1)$, $\frac{\partial}{\partial \beta} g(\alpha, 0)$ et $\frac{\partial}{\partial \beta} g(\alpha, 1)$ résultent d'une interpolation unidimensionnelle suivant la direction α de la fonction g et de sa dérivée $\frac{\partial}{\partial \beta} g$ sur

l'arête inférieure et l'arête supérieure de la cellule considérée :

$$\begin{aligned}
g(\alpha,0) =& \varphi_1(\alpha)g(0,0) + \varphi_2(\alpha)g(1,0) + \\
& \varphi_3(\alpha)\frac{\partial}{\partial\alpha}g(0,0) + \varphi_4(\alpha)\frac{\partial}{\partial\alpha}g(1,0).\\
g(\alpha,1) =& \varphi_1(\alpha)g(0,1) + \varphi_2(\alpha)g(1,1) + \\
& \varphi_3(\alpha)\frac{\partial}{\partial\alpha}g(0,1) + \varphi_4(\alpha)\frac{\partial}{\partial\alpha}g(1,1).\\
\frac{\partial}{\partial\beta}g(\alpha,0) =& \varphi_1(\alpha)\frac{\partial}{\partial\beta}g(0,0) + \varphi_2(\alpha)\frac{\partial}{\partial\beta}g(1,0) + \\
& \varphi_3(\alpha)\frac{\partial^2}{\partial\alpha\partial\beta}g(0,0) + \varphi_4(\alpha)\frac{\partial^2}{\partial\alpha\partial\beta}g(1,0).\\
\frac{\partial}{\partial\beta}g(\alpha,1) =& \varphi_1(\alpha)\frac{\partial}{\partial\beta}g(0,1) + \varphi_2(\alpha)\frac{\partial}{\partial\beta}g(1,1) + \\
& \varphi_3(\alpha)\frac{\partial^2}{\partial\alpha\partial\beta}g(0,1) + \varphi_4(\alpha)\frac{\partial^2}{\partial\alpha\partial\beta}g(1,1).
\end{aligned}$$

Ces relations nous montre que le calcul de $P_{i,j}(\alpha,\beta)$ se résume à une combinaison des fonctions g, $\frac{\partial}{\partial\alpha}g$, $\frac{\partial}{\partial\beta}g$ et $\frac{\partial^2}{\partial\alpha\partial\beta}g$ évaluées aux 4 coins de la cellule $C_{i,j}$.

On souhaite maintenant exprimer $P_{i,j}$ en fonction de f. Les variables r et θ étant liées à α et β par (C.2) et (C.3), on étudie les dérivées $\frac{\partial}{\partial\alpha}g$, $\frac{\partial}{\partial\beta}g$ et $\frac{\partial^2}{\partial\alpha\partial\beta}g$.

Soit :

$$\begin{aligned}
\frac{\partial}{\partial\alpha}g(\alpha,\beta) &= \frac{\partial}{\partial\alpha}(f(r,\theta)),\\
&= \frac{\partial r}{\partial\alpha}\cdot\frac{\partial}{\partial r}f(r,\theta) + \frac{\partial\theta}{\partial\alpha}\cdot\frac{\partial}{\partial\theta}f(r,\theta),\\
&= \Delta r\cdot\frac{\partial}{\partial r}f(r,\theta).
\end{aligned}$$

Et :

$$\frac{\partial}{\partial\beta}g(\alpha,\beta) = \Delta\theta\cdot\frac{\partial}{\partial\theta}f(r,\theta).$$

Enfin :

$$\begin{aligned}
\frac{\partial^2}{\partial\alpha\partial\beta}g(\alpha,\beta) &= \frac{\partial^2}{\partial\alpha\partial\beta}(f(r,\theta))\\
&= \frac{\partial}{\partial\alpha}(\frac{\partial}{\partial\beta}(f(r,\theta)))\\
&= \frac{\partial}{\partial\alpha}(\Delta\theta\cdot\frac{\partial}{\partial\theta}f(r,\theta))\\
&= \Delta\theta\cdot\frac{\partial}{\partial\theta}(\frac{\partial}{\partial\alpha}(f(r,\theta)))\\
&= \Delta\theta\cdot\frac{\partial}{\partial\theta}(\Delta r\cdot\frac{\partial}{\partial r}f(r,\theta))\\
&= \Delta r\Delta\theta\cdot\frac{\partial^2}{\partial r\partial\theta}f(r,\theta).
\end{aligned}$$

C.3 Représentation des fonctions de bases d'Hermite en $2D$

Dans la section 4.3.2, les contraintes sur les polynômes d'Hermite $(\varphi_i)_{1\leqslant i\leqslant 4}$ qui composent une base des polynômes de degré 3 sont données. Le Tableau C.1 représente ces contrainte sous forme compact. On considère ici que les polynômes (φ_i) varient en fonction de la variable $\alpha \in [0,1]$.

	φ_1	φ_3	φ_2	φ_4
$\cdot\,(0)$	1	0	0	0
$\cdot\,(1)$	0	1	0	0
$\frac{\mathrm{d}}{\mathrm{d}\,\alpha}\cdot\,(0)$	0	0	1	0
$\frac{\mathrm{d}}{\mathrm{d}\,\alpha}\cdot\,(1)$	0	0	0	1

TABLE C.1 – Contraintes des polynômes de la base d'Hermite en $1D$.

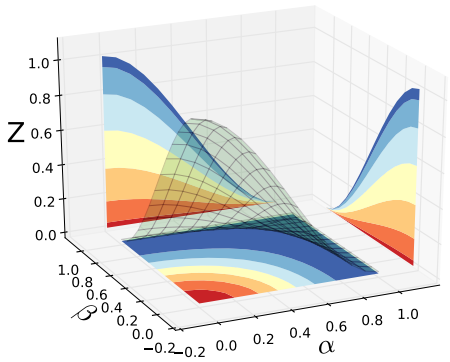
A partir de ce tableau et des expressions des polynômes d'Hermite en $2D$ dont la construction est décrite dans l'Annexe C.2, On peut déduire les contraintes que respectent les polynômes $(P_{i,j})_{1\leqslant i,j\leqslant 4}$. Ces contraintes sont présentées dans le Tableau C.2. On considère ici que les polynômes $(P_{i,j})$ varient en fonction des variables $(\alpha,\beta) \in [0,1]^2$.

	$P_{1,1}$	$P_{1,2}$	$P_{2,1}$	$P_{2,2}$	$P_{3,1}$	$P_{3,2}$	$P_{4,1}$	$P_{4,2}$	$P_{1,3}$	$P_{1,4}$	$P_{2,3}$	$P_{2,4}$	$P_{3,3}$	$P_{3,4}$	$P_{4,3}$	$P_{4,4}$
$\cdot\,(0,0)$	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$\cdot\,(0,1)$	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$\cdot\,(1,0)$	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
$\cdot\,(1,1)$	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
$\frac{\partial}{\partial\alpha}\cdot\,(0,0)$	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
$\frac{\partial}{\partial\alpha}\cdot\,(0,1)$	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
$\frac{\partial}{\partial\alpha}\cdot\,(1,0)$	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
$\frac{\partial}{\partial\alpha}\cdot\,(1,1)$	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
$\frac{\partial}{\partial\beta}\cdot\,(0,0)$	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
$\frac{\partial}{\partial\beta}\cdot\,(0,1)$	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
$\frac{\partial}{\partial\beta}\cdot\,(1,0)$	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
$\frac{\partial}{\partial\beta}\cdot\,(1,1)$	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
$\frac{\partial^2}{\partial\alpha\partial\beta}\cdot\,(0,0)$	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
$\frac{\partial^2}{\partial\alpha\partial\beta}\cdot\,(0,1)$	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
$\frac{\partial^2}{\partial\alpha\partial\beta}\cdot\,(1,0)$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
$\frac{\partial^2}{\partial\alpha\partial\beta}\cdot\,(1,1)$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

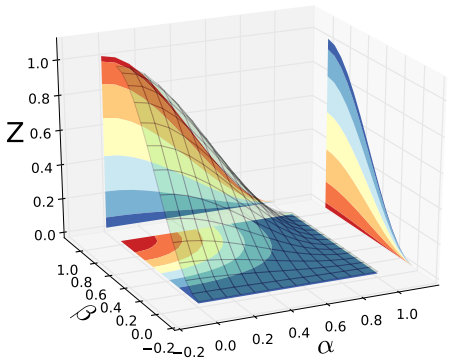
TABLE C.2 – Contraintes des polynômes de la base d'Hermite en $2D$.

Les Figures C.1, C.2 et C.3 donnent l'allure des polynômes $(P_{i,j})$.

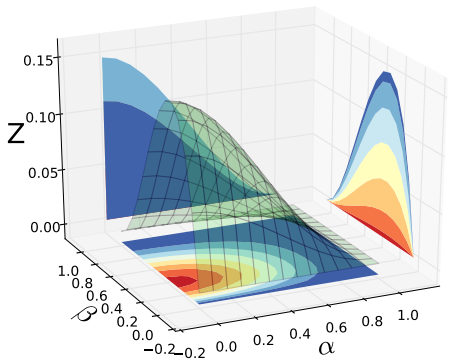
Remarque : La méthode de construction des polynômes $2D$ $(P_{i,j})$ présentée en Annexe C.2 consiste à effectuer des interpolations $1D$ successives. Il est possible détendre ce concept pour construire une interpolation d'Hermite à N dimension.



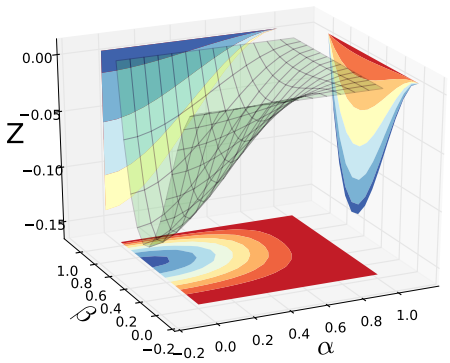
(a) $P_{1,1}(\alpha,\beta) = \varphi_1(\alpha)\varphi_1(\beta)$



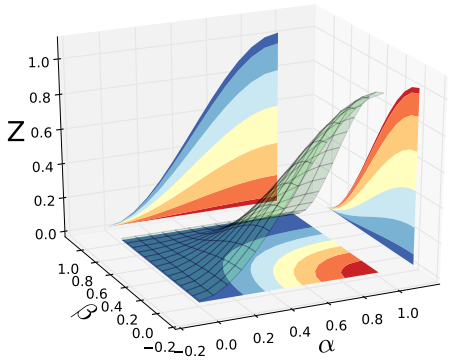
(b) $P_{1,2}(\alpha,\beta) = \varphi_1(\alpha)\varphi_2(\beta)$



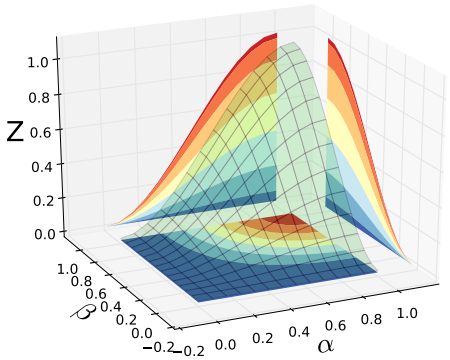
(c) $P_{1,3}(\alpha,\beta) = \varphi_1(\alpha)\varphi_3(\beta)$



(d) $P_{1,4}(\alpha,\beta) = \varphi_1(\alpha)\varphi_4(\beta)$

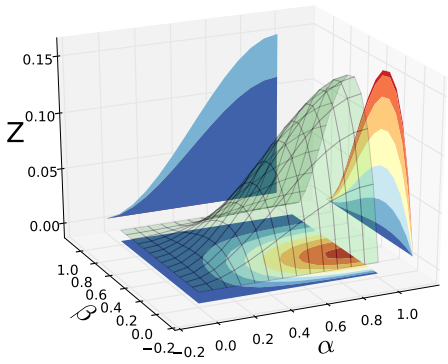


(e) $P_{2,1}(\alpha,\beta) = \varphi_2(\alpha)\varphi_1(\beta)$

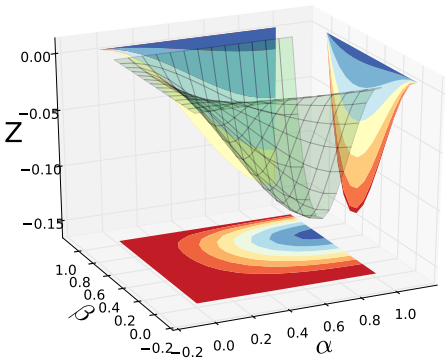


(f) $P_{2,2}(\alpha,\beta) = \varphi_2(\alpha)\varphi_2(\beta)$

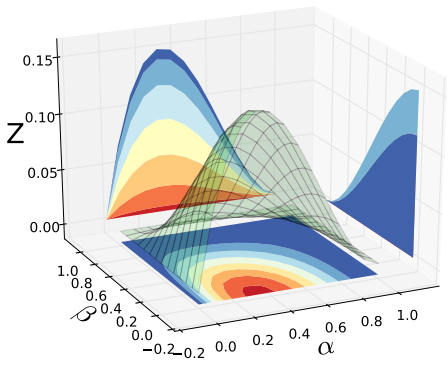
FIGURE C.1 – Polynômes de la base d'Hermite en $2D$ (de $P_{1,1}$ à $P_{2,2}$)



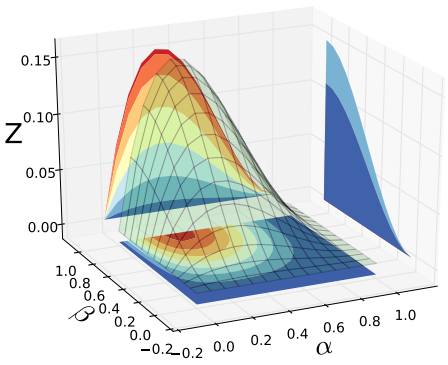
(a) $P_{2,3}(\alpha,\beta) = \varphi_2(\alpha)\varphi_3(\beta)$



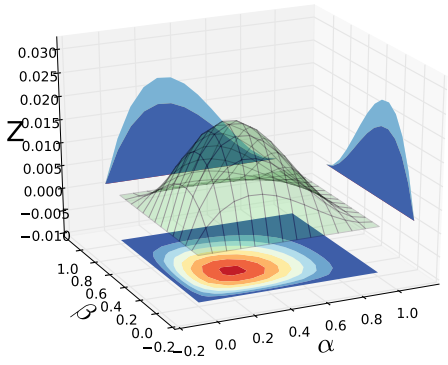
(b) $P_{2,4}(\alpha,\beta) = \varphi_2(\alpha)\varphi_4(\beta)$



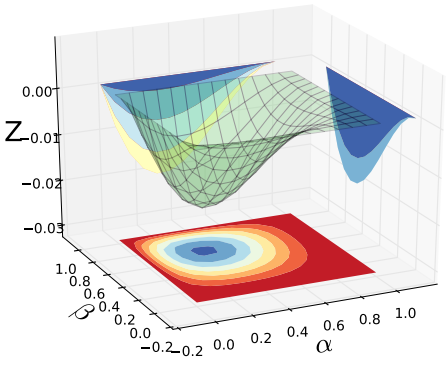
(c) $P_{3,1}(\alpha,\beta) = \varphi_3(\alpha)\varphi_1(\beta)$



(d) $P_{3,2}(\alpha,\beta) = \varphi_3(\alpha)\varphi_2(\beta)$

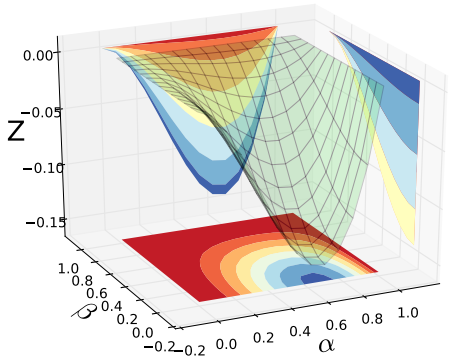


(e) $P_{3,3}(\alpha,\beta) = \varphi_3(\alpha)\varphi_3(\beta)$

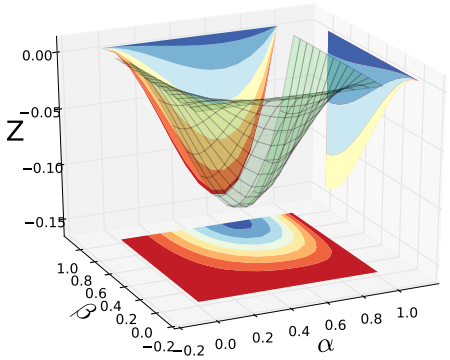


(f) $P_{3,4}(\alpha,\beta) = \varphi_3(\alpha)\varphi_4(\beta)$

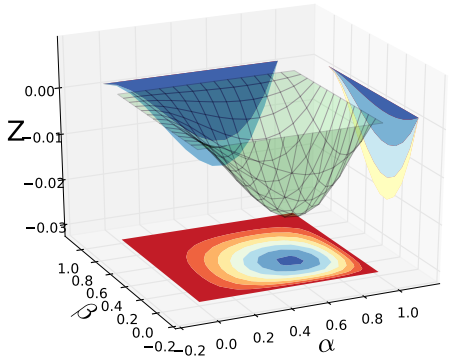
FIGURE C.2 – Polynômes de la base d'Hermite en $2D$ (de $P_{2,3}$ à $P_{3,4}$)



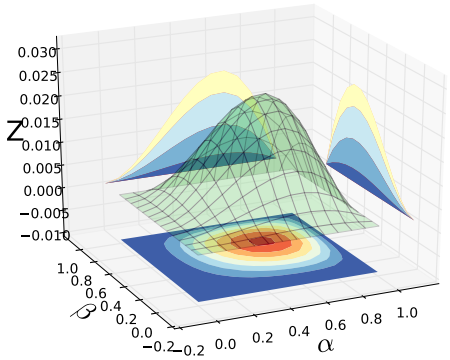
(a) $P_{4,1}(\alpha, \beta) = \varphi_4(\alpha)\varphi_1(\beta)$



(b) $P_{4,2}(\alpha, \beta) = \varphi_4(\alpha)\varphi_2(\beta)$



(c) $P_{4,3}(\alpha, \beta) = \varphi_4(\alpha)\varphi_3(\beta)$



(d) $P_{4,4}(\alpha, \beta) = \varphi_4(\alpha)\varphi_4(\beta)$

FIGURE C.3 – Polynômes de la base d'Hermite en $2D$ (de $P_{4,1}$ à $P_{4,4}$)