Conception d'un modèle de composants logiciels avec ordonnancement de tâches pour les architectures parallèles multi-coeurs, application au code Gysela

Jérôme Richard

Numéro National de Thèse : 2017LYSEN040

THESE de DOCTORAT DE L'UNIVERSITE DE LYON
opérée par
l'Ecole Normale Supérieure de Lyon

Ecole Doctorale N°512
en Informatique et Mathématiques de Lyon

Spécialité de doctorat :
Informatique

Soutenue publiquement le 06/12/2017, par :
Jérôme RICHARD

Conception d'un modèle de composants logiciels avec ordonnancement de tâches pour les architectures parallèles multi-cœurs, application au code Gysela

---

Devant le jury composé de :

Françoise BAUDE	Professeure – Université de Nice	Rapporteure
Marco DANELUTTO	Professeur – Università Di Pisa, Italie	Rapporteur
Marie-Alice FOUJOLS	Ingénieure – Institut Pierre Simon Laplace	Examinatrice
Bruno RAFFIN	Directeur de Recherche – Inria Grenoble	Examinateur
Christian PÉREZ	Directeur de Recherche – Inria ENS Lyon	Directeur
Julien BIGOT	Ingénieur Chercheur – CEA Saclay	Co-encadrant

Remerciements

Je voudrais commencer par remercier les membres du jury : Françoise Baude et Marco Danelutto pour avoir évalué le manuscrit malgré le peu de temps qu'ils aient eu et bien que le manuscrit soit rédigé en français ; Bruno Raffin et Marie-Alice Foujols pour avoir accepté d'examiner ma soutenance.

Je souhaite ensuite remercier mes encadrants de thèse Christian Pérez et Julien Bigot sans qui je n'aurai simplement pas réalisé cette thèse. J'ai pu partager avec vous de bons moments (Luminy, Vienne, Madrid), même si ça n'a pas toujours été facile quand vos avis divergeaient. Julien, j'essaierai à l'avenir de garder l'âne vivant !

J'aimerais aussi remercier ceux qui ont directement contribué à ma thèse : Hélène Coullon pour m'avoir encadré et encouragé dans les moments difficiles ; Thierry Gautier pour m'avoir aidé dans mes travaux et pour m'avoir permis d'échanger longuement sur des sujets passionnants ; Olivier Aumage pour m'avoir conseillé (surtout en début de thèse) et m'avoir invité à des événements.

Un grand merci à ma famille, en particulier à mon frère, Nicolas, qui m'a soutenu dans mes études et m'a supporté durant ma thèse ; Merci aussi à mes amis et ceux qui m'ont soutenus durant ce long périple : Issam (l'expert FORTRAN qui aime les salades sans salades) et Millian (le grand gourou de la Strutsologie) pour toutes nos discussions souvent bénéfiques et parfois profondes ; Leslie pour son aide précieuse à la préparation du pot de thèse.

Je tiens à remercier aussi l'équipe de Gysela, dynamique et soudée, avec laquelle j'ai passé des moments agréables (au CEMRACS comme à Cadarache). Plus particulièrement, je tiens à remercier Guillaume Latu et Virginie Grandgirard qui étaient présents pour répondre aux questions sur le fonctionnement l'application. Merci à Charles pour nous avoir fait une visite guidée d'un tokamak grandeur nature !

Un grand merci à toute l'équipe Avalon avec laquelle j'ai pu travailler pendant mon stage et ma thèse, mais aussi avec laquelle j'ai pu longuement discuter (Issam, Radu, Pedro, Sylvain, Violaine), voire échanger des trolls (Laurent P., Anthony, Arnaud). Merci à la team composants (Vincent & Hélène). Merci à Laurent L. (toujours de bonne humeur), Nicolas et Fabien pour m'avoir permis d'enseigner. Merci à Simon, Matthieu et Jean-Christophe pour m'avoir aidé lorsque je rencontrais des problèmes techniques. Merci aux assistantes et en particulier à Evelyne pour m'avoir aidé pour organiser les événements durant ma thèse.

Enfin, merci à l'équipe de la maison de la simulation avec lesquels j'ai pu partager des moments agréables (Matthieu, Julien D., Yacine, Pascal, Michel, Edouard, Samuel, Pierre, Martial, Maxime, Arnaud, Karen, Ksander et j'en oublie) durant mon passage à la maison de la simulation, surtout lors des pauses et des pizzaminaires !

Table des matières

Chapitre 1

Introduction

1.1 Contexte

Le calcul de haute performance (HPC) alimente depuis des décennies l'amélioration continue des applications scientifiques et permet de relever sans cesse de nouveaux défis sociétaux. Les répercussions sont multiples et affectent de nombreux domaines de recherche et industriels, de l'étude du climat à l'analyse du cerveau, en passant par la simulation de collisions de véhicules et la fusion nucléaire. Ces applications ont pu traiter des volumes de données toujours plus importants, tout en gagnant en précision au fil des années. La clef de cette puissance de calcul titanesque réside majoritairement dans la capacité du matériel à traiter des opérations massivement en parallèle. Un grand nombre d'unités de calcul sont agrégées au sein de nœuds, qui sont à leur tour répliqués pour former des supercalculateurs perçus comme une seule et même machine. Les prochaines générations de supercalculateurs devraient atteindre une puissance de calcul de l'ordre de l'exaflops. Pour cela, ils devraient disposer de nœuds de calcul avec des hiérarchies de mémoires caches profondes et complexes, utilisant des processeurs dotés d'un grand nombre de cœurs (tels que les processeurs Xeon-Phi) ainsi que des accélérateurs dédiés (tels que les processeurs graphiques programmables). Toutefois, l'exploitation de telles architectures est difficile, ce qui rend nécessaire la mise en place de modèles de programmation adaptés.

Les supports exécutifs d'ordonnancement à base de graphes de tâches pour le calcul de haute performance ont été conçus pour utiliser efficacement les architectures matérielles complexes des supercalculateurs et pour aider à obtenir des performances portables (c.-à-d. des performances similaires sur du matériel divers). Dans cette approche, les applications sont décrites comme des graphes de tâches avec des contraintes d'ordonnancement entre elles, appelées dépendances. Les supports exécutifs sont responsables d'ordonnancer les tâches produites par une application sur les ressources disponibles (par exemple, les cœurs de processeurs centraux ou d'accélérateurs de calcul). Cela permet une utilisation efficace des ressources disponibles, en particulier lorsque les applications fournissent une charge inéquitable de travail à effectuer en parallèle (c.-à-d. calculs irréguliers).

Cependant, les supports exécutifs d'ordonnancement HPC ne tiennent pas compte des besoins en génie logiciel des applications tels que la réutilisation, la maintenabilité et l'extensibilité des logiciels HPC. Si aucune attention particulière n'est portée sur ces aspects, les coûts de développement peuvent exploser au fil du temps en raison d'une duplication des efforts, d'une multiplication des préoccupations au sein d'un même logiciel, etc. Par ailleurs, le code peut rapidement devenir obsolète. Dans les pires cas, les conséquences néfastes de l'obsolescence peuvent apparaître, avant même qu'un logiciel ne devienne suffisamment fonctionnel.

La programmation par composition est une branche du génie logiciel qui propose de construire des applications en assemblant des briques logicielles indépendantes (c.-à-d. composants) disposant d'interfaces bien définies. Les composants sont instanciés et leurs interfaces sont connectées pour former un assemblage. Cette approche offre un moyen de réutiliser simplement des composants, fournit des mécanismes pour aider à la séparation des préoccupations et permet de manipuler aisément l'architecture des logiciels (à travers leur assemblage). L'objectif de cette approche est d'améliorer significativement la maintenabilité et l'adaptabilité des applications.

Une composition de codes parallèles compatible avec les approches d'ordonnancement de graphes de tâches pourrait être grandement bénéfique pour les applications HPC. En effet, cela permettrait de concevoir des logiciels de haute performance exploitant efficacement les ressources matérielles à disposition et de simplifier leur maintenance au fil du temps.

En particulier, cette thèse s'intéresse à une sous-partie de l'application HPC de simulation gyrocinétique de plasmas par confinement magnétique pour la fusion Gysela. Les enjeux en ce qui concerne l'application sont multiples. Premièrement, la séparation de certaines parties du code de Gysela, sans impacter significativement les performances par rapport à la mise en œuvre utilisée en production, offrirait un moyen de maintenir plus facilement l'application, qui réalise à ce jour des compromis entre performances obtenues et maintenabilité du code. Deuxièmement, l'adaptation aisée des stratégies d'exécution (ordonnancement) des parties du code, au niveau de l'application entière, permettrait de simplifier grandement le processus de conception et d'optimisation de Gysela sur différentes plateformes. En effet, cela permet de relayer le fardeau de la composition efficace de codes parallèles à grain fin à un ordonnanceur, plutôt qu'à des développeurs qui, à l'heure actuelle, effectuent cette tâche manuellement. Une telle approche permettrait à un support exécutif de prendre des décisions globales tout en considérant le matériel utilisé, ce qui est difficilement réalisable manuellement sans dégrader considérablement la maintenabilité du code de l'application.

1.2 Objectifs

Cette thèse étudie les mécanismes de composition de codes parallèles pour le calcul de haute performance au sein d'une application. Plus précisément, elle vise tout d'abord à étudier la faisabilité d'une méthode permettant de composer efficacement des codes HPC indépendants tirant parti des approches d'ordonnancement de

tâches, puis à proposer et à évaluer une telle approche sur une sous-partie de Gysela. Idéalement, l'approche mise en place doit simplifier la maintenance de la portion de Gysela considérée, sans impacter significativement ses performances. Afin de limiter la complexité du problème, cette thèse se concentre plus particulièrement sur les architectures multi-cœurs à mémoire partagée.

1.3 Contributions

Afin de remplir les objectifs énoncés, cette thèse, à la croisée de plusieurs domaines de recherche, propose une méthode visant à résoudre de bout en bout des problèmes relatifs à la composition de codes HPC : de la conception d'un modèle de composants (c.-à-d. COMET) en passant par la réalisation d'une mise en œuvre (c.-à-d. HALLEY), jusqu'à son emploi sur un cas d'utilisation réaliste (c.-à-d. Gysela). Ces aspects forment les trois contributions de cette thèse.

Comet Conception du modèle de composants COMET visant à composer efficacement des codes parallèles indépendants à base de graphes de tâches et ciblant principalement les architectures multi-cœurs à mémoire partagée. Pour cela, COMET combine simultanément des approches à base de flots de données (en y intégrant le concept de partitionnement) et de services requis/fournis. Le modèle aussi offre un support minimaliste des codes MPI.

Halley Réalisation d'un prototype de mise en œuvre conforme au modèle COMET nommée HALLEY s'appuyant sur l'ingénierie dirigée par les modèles (IDM) ainsi que des approches existantes telles que le modèle de composants minimaliste L^2C et le langage de directives OpenMP. La mise en œuvre adopte une approche modulaire permettant à des experts d'étendre simplement la mise en œuvre.

Évaluation Une évaluation des apports de COMET, et plus spécifiquement de la mise en œuvre de HALLEY, en termes de propriété de génie logiciel et de performance constituée de deux parties distinctes. Une première partie analyse des cas d'utilisation synthétiques. Une seconde partie étudie une sous-partie de l'application de simulation gyrocinétique de plasmas par confinement magnétique pour la fusion Gysela.

1.4 Organisation du manuscrit

Ce manuscrit est divisé en sept chapitres distincts.

Le chapitre 2 présente le contexte et les travaux relatifs existants portant sur le calcul de haute performance. Tout d'abord, il introduit les architectures matérielles utilisées. Puis, il présente les diverses applications qui s'exécutent sur les plateformes HPC. Ce chapitre décrit ensuite les modèles de programmation permettant d'exprimer les traitements effectués par les applications de manière à ce qu'ils

soient exécutés sur le matériel sous-jacent. Après, ce chapitre présente les méthodes permettant de structurer et maintenir les logiciels traitant tout particulièrement de l'usage des modèles de composants. Enfin, le chapitre se clôture par une discussion sur les objectifs de cette thèse.

Le chapitre 3 présente COMET, un modèle de composants visant à composer efficacement des codes parallèles indépendants à base de graphes de tâches.

Le chapitre 4 détaille le fonctionnement d'une mise en œuvre extensible de COMET, nommée HALLEY, et décrit par quels moyens des experts peuvent étendre cette mise en œuvre.

Le chapitre 5 analyse et évalue COMET, et plus précisément HALLEY, sur des cas d'utilisation synthétiques tirés de patrons récurrents issus d'applications HPC existantes telles que Gysela.

Le chapitre 6 effectue une évaluation plus détaillée sur un cas d'utilisation réaliste : une partie de l'application de production Gysela.

Enfin, le chapitre 7 conclut ce manuscrit et présente des perspectives de recherche.

1.5 Publications

Conférences internationales

[10] *Combining Both a Component Model and a Task-based Model for HPC Applications : a Feasibility Study on GYSELA*. Olivier Aumage, Julien Bigot, Hélène Coullon, Christian Pérez and Jérôme Richard [1]. Proceedings of the International Symposium on Cluster, Cloud and Grid Computing, IEEE/ACM, 2017.
https://hal.inria.fr/hal-01518730.
Note : Contribution montrant la faisabilité d'une approche de modèles de composants avec un support de l'ordonnancement de graphes de tâches sur une sous-partie de Gysela.

[21] (minor revision) *Building and Auto-Tuning Computing Kernels : Experimenting with BOAST and StarPU in the GYSELA code*. Julien Bigot, Virginie Grandgirard, Guillaume Latu, Jean-François Mehaut, Luís Felipe Millani, Chantal Passeron, Steven Quinito Masnada, Jérôme Richard, Brice Videau [2]. ESAIM : Proceedings and Surveys, 2017.
Note : Contribution visant à analyser les bénéfices de l'utilisation de supports exécutifs d'ordonnancement de graphes de tâches et de techniques d'adaptation automatique de paramètres sur une sous-partie de Gysela.

1. Liste des auteurs par ordre alphabétique. Contribution principale à l'article.
2. Liste des auteurs par ordre alphabétique. Contribution majeure à l'article.

Workshops internationaux

[76] *Towards Application Variability Handling with Component Models : 3D-FFT Use Case Study*. Vincent Lanore, Christian Pérez and Jérôme Richard [1]. In *Euro-Par 2015 : Parallel Processing Workshops*, page 12, Springer, 2015. https://hal.inria.fr/hal-01192732.
Note : Contribution poursuivant les travaux réalisés en stage de master visant à adapter simplement la structure d'un algorithme de transformée de Fourrier tridimensionnelle en manipulant un assemblage de composants.

Conférences nationales

[100] *Vers un modèle de composants supportant l'ordonnancement de tâches pour le calcul de haute performance*. Jérôme Richard [1]. ComPAS (classé dans la catégorie "meilleurs articles"), 2015. https://hal.inria.fr/hal-01192661.
Note : Travaux préliminaires de thèses qui proposent d'étendre le modèle de composants L^2C avec des ports d'ordonnancement de tâches pour permettre de construire et ordonnancer un graphe de tâches en utilisant uniquement des dépendances de contrôle.

Rapports de recherche

[22] *COMET : An High-Performance Model for Fine-Grain Composition*. Julien Bigot, Thierry Gautier, Christian Pérez and Jérôme Richard [1]. Rapport de recherche Inria, 2017. https://hal.inria.fr/hal-01566288.
Note : Travaux proposant et analysant le modèle COMET décrit dans le chapitre 3, évalué sur des cas d'utilisation synthétiques présentés dans le chapitre 5.

[37] *Feasibility Study of a Runtime Component-based Model Integrating Task Graph Concept on a 1D Advection Case Study*. Christian Pérez, Hélène Coullon, Jérôme Richard [1]. Livrable du projet d'investissement d'avenir ELCI (évalué par des pairs). https://hal.inria.fr/hal-01348204.
Note : Rapport présentant des travaux préliminaires de la publication [10] analysant une sous-partie plus simple de l'application Gysela.

[75] *Evaluating Component Assembly Specialization for 3D FFT*. Vincent Lanore, Christian Pérez and Jérôme Richard [1]. White paper PRACE-2IP (évalué par des pairs). http://www.prace-ri.eu/IMG/pdf/WP182.pdf.
Note : Rapport présentant des travaux préliminaires de la publication [76] (relatifs au stage de master).

Chapitre 2

Contexte et travaux connexes : le calcul de haute performance

Contents

Ce chapitre a pour objectif de donner un aperçu du contexte matériel et logiciel des systèmes dédiés au calcul de haute performance (HPC) requis pour une bonne compréhension de ce manuscrit. Dans une première partie, il donne un aperçu de la diversité du matériel utilisé par les applications de calcul de haute performance. Dans une seconde partie, il présente les applications de calcul de haute performance considérées, leurs enjeux ainsi que leur fonctionnement général. Dans une troisième partie, il détaille les modèles de programmation étudiés et les principes fondamentaux sur lesquels ils reposent. Dans une quatrième partie, il étudie les méthodes et procédures systématiques permettant d'élaborer et maintenir des logiciels de grande taille, en analysant tout particulièrement les modèles de programmation par composition. Enfin, ce chapitre se termine par une analyse du contexte actuel qui introduit les motivations au cœur de nos travaux en faisant ressortir les forces et les limitations des modèles existants. À travers cette analyse, nous exposons ainsi le problème étudié dans ce manuscrit et nous formulons les propriétés attendues pour élaborer un modèle qui répond au problème énoncé.

2.1 Plateformes

Afin de comprendre le fonctionnement des applications HPC, il est important de comprendre comment fonctionnent les architectures matérielles sur lesquelles elles s'exécutent. En effet, deux codes sources produisant des résultats identiques ayant

recours à un même algorithme peuvent s'exécuter à des vitesses différentes, parfois même avec une différence de plusieurs ordres de grandeur (p. ex. adaptation des structures de données utilisées, exploitation d'opérations spécifiques mises à disposition par l'architecture sous-jacente, etc.). Ainsi, il est question dans cette section de présenter succinctement l'évolution des architectures jusqu'à aujourd'hui, puis de décrire les architectures actuellement couramment utilisées et finalement d'entrevoir les architectures dont l'usage pourrait se démocratiser dans un futur proche.

2.1.1 Aperçu

La croissance exponentielle de la capacité de calcul des systèmes informatiques alimente depuis des décennies l'amélioration continue des applications issues du monde de l'industrie et de la recherche scientifique et permet de faire face à de nombreux défis sociétaux. Les conséquences sont multiples, impactant de nombreux champs de recherche, de la modélisation du climat à l'échelle planétaire, à l'analyse des structures les plus complexes de l'univers telles que le cerveau, en passant par la recherche d'énergies alternatives telles que la fusion nucléaire. Les applications ont progressivement pu traiter des volumes de données plus gros et plus précisément au fil du temps. C'est dans cette course effrénée, dès 1965, qu'est établie la première *loi de Moore* [86], un postulat empirique affirmant que la complexité (densité) des semi-conducteurs allait doubler tous les ans. La proposition fut réévaluée dans une seconde loi [87], en 1975, suite à la démocratisation des circuits intégrés face à l'usage des transistors, changeant ainsi la période à un peu moins de deux ans et affirmant plus précisément que le nombre de transistors doublerait durant ce laps de temps.

Avant les années 1960, les supercalculateurs étaient séquentiels et contenaient une seule unité de calcul centrale (CPU) dont le but était d'exécuter les instructions des applications le plus rapidement possible. Durant cette période et longtemps après, la majorité des gains provenait principalement de la *montée en fréquence* et de l'optimisation des circuits électromécaniques comportant une forme de *parallélisme intrinsèque*. En effet, les circuits étaient déjà capables d'effectuer une instruction en parallèle (p. ex. calcul d'une addition en parallèle) tout en exécutant un flot d'instructions séquentiellement. Ce parallélisme était donc totalement transparent aux yeux des programmeurs.

Ces progrès ne suffisaient pas à répondre aux besoins applicatifs en quête de toujours plus de puissance de calcul. La montée en fréquence fut tout d'abord limitée par la complexité des opérations à réaliser à chaque cycle d'horloge. Dans les années 1960, les unités de calcul ont alors été dotées de *pipelines* permettant d'exécuter un flot d'instructions séquentiel en parallèle en les décomposant en plusieurs petites étapes (p. ex. chargement d'une instruction, décodage et exécution). La réduction du nombre d'opérations à effectuer à chaque cycle d'horloge a permis d'augmenter significativement la fréquence des unités de calcul. La montée en fréquence fut ensuite rapidement limitée par la dissipation thermique qui constituait un frein majeur à l'évolution de la puissance de calcul. Les concepteurs ont donc commencé à agréger plusieurs unités de calcul dans une seule machine parallèle.

Dans les années 1970, les machines *vectorielles* ont vu le jour. Ces machines étaient capables de traiter simultanément plusieurs données en une seule instruction permettant une augmentation considérable de la puissance de calcul délivrée. Ce type d'architecture était particulièrement efficace pour accélérer un grand nombre d'applications numériques travaillant sur des tableaux et matrices, ce qui a fait d'elles un succès reconnu.

Les architectures d'aujourd'hui sont complexes et massivement parallèles : elles peuvent tirer parti simultanément de toutes les formes de parallélisme évoquées précédemment (c.-à-d. architectures multi-cœurs vectorielles avec pipelines) et même exploiter de nouvelles formes. Par exemple, la plupart des architectures matérielles actuelles permettent de réorganiser un flot séquentiel d'instructions tout en gardant un ordre correct (exécution *out-of-order*) de telle manière à exécuter les instructions plus efficacement. Certains processeurs sont même capables d'exécuter plusieurs instructions d'un unique flot d'instruction en parallèle (parallélisme d'instructions), on parle alors de processeurs *superscalaires*.

Dans ce document, nous classons les architectures parallèles en deux catégories distinctes : d'une part les *architectures à mémoire partagée* et d'autre part les *architectures à mémoire distribuée* qui agrègent des machines à mémoire partagée.

2.1.2 Architectures à mémoire partagée

Malgré la diversité des *architectures à mémoire partagée*, on peut classer ces architectures à mémoire partagée de ces dernières décennies en plusieurs groupes. Nous classifions tout d'abord ces architectures selon la manière dont les unités de calcul accèdent à la mémoire (c.-à-d. impact des temps d'accès en fonction des unités de calcul qui y accèdent). Deux catégories sont présentées : les *architectures à accès mémoire uniforme* (UMA) et *non uniforme* (*NUMA*). Ensuite, nous détaillons le fonctionnement des constituants de ces architectures, en particulier l'architecture des processeurs et le fonctionnement de la hiérarchie mémoire. Enfin, nous expliquons le fonctionnement des *architectures many-cœurs* et hétérogènes qui seront très probablement au cœur des futurs systèmes HPC.

Architectures UMA

Les *architectures UMA* (*Uniform Memory Access*) apparaissent pour la toute première fois dans les années 1960, suite à l'émergence des systèmes multiprocesseurs. Ces architectures contiennent un ensemble d'unités de calcul et de *bancs mémoires* ainsi que des *contrôleurs d'entrées/sorties* (assurant la communication avec d'autres systèmes, le stockage des données, etc.) interconnectés entre eux par un *réseau d'interconnexion* rapide. Un *banc mémoire* est une région d'une mémoire centrale adressable partagée. Dans ces architectures, le temps d'accès à un banc mémoire est indépendant de l'unité de calcul qui y accède. La figure 2.1 montre deux exemples architecture UMA.

Les *architectures SMP* (*Symetric Multi-Processing*) sont un sous-groupe des architectures UMA. Ces architectures considèrent des unités de calcul toutes iden-

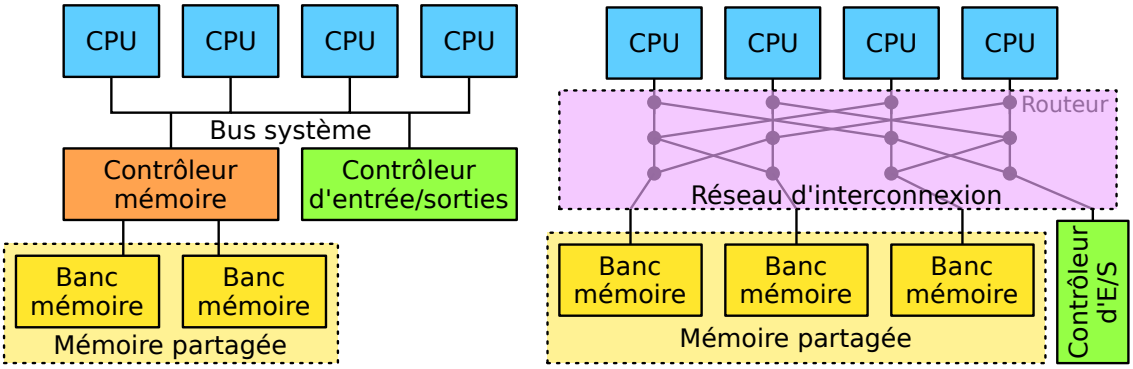

FIGURE 2.1 – Exemple d'architecture SMP (gauche) et d'architecture UMA (droite).

tiques et ont recours à un *bus* (ou un *commutateur crossbar*) comme réseau d'interconnexion. De plus, les unités de calcul accèdent aux bancs mémoire par un unique contrôleur mémoire. Les architectures SMP ont longtemps dominé les systèmes HPC principalement en raison de leur simplicité. La partie gauche de la figure 2.1 présente un exemple architecture SMP.

L'usage d'un bus d'interconnexion et d'un unique contrôleur a constitué plus récemment un goulot d'étranglement lorsqu'un grand nombre d'unités de calcul et de bancs mémoire est présent. En effet, le bus et le contrôleur des architectures SMP ne sont plus capables de supporter le débit de données gigantesque entre les unités de calcul et les mémoires (sérialisation des communications). Ainsi, bien que ces architectures restent toujours utilisées aujourd'hui (p. ex. dans de nombreux CPU) en raison de leur simplicité, elles ont progressivement laissé la place aux architectures NUMA, plus complexes à exploiter, au milieu des années 2000 [1].

Certaines architectures UMA disposent de plusieurs contrôleurs et d'un réseau d'interconnexion plus complexe pour pallier la saturation de l'unique contrôleur mémoire présent dans l'architecture SMP. Pour y parvenir, ces architectures emploient notamment des mécanismes de routage des communications. Néanmoins, la complexité de telles approches peut engendrer un coût prohibitif. La partie droite de la figure 2.1 montre un exemple d'une telle architecture.

Architectures NUMA

Les *architectures NUMA* (*Non-Uniform Memory Access*) contiennent des unités de calcul, des contrôleurs mémoire et des bancs mémoire regroupés ensemble par des *nœuds NUMA* ainsi que des contrôleurs d'entrées/sorties. Les nœuds NUMA et contrôleurs d'entrées/sorties sont interconnectés entre eux par un réseau d'interconnexion point à point. Les unités de calcul peuvent accéder rapidement aux bancs mémoire de leur propre nœud NUMA (bancs locaux) et plus lentement aux bancs mémoire des autres nœuds NUMA (à cause de la proximité et de la structure po-

1. Par exemple suite au remplacement du front side bus, utilisé dans de nombreux systèmes, par un réseau point à point tel que l'HyperTransport ou le QuickPath Interconnect

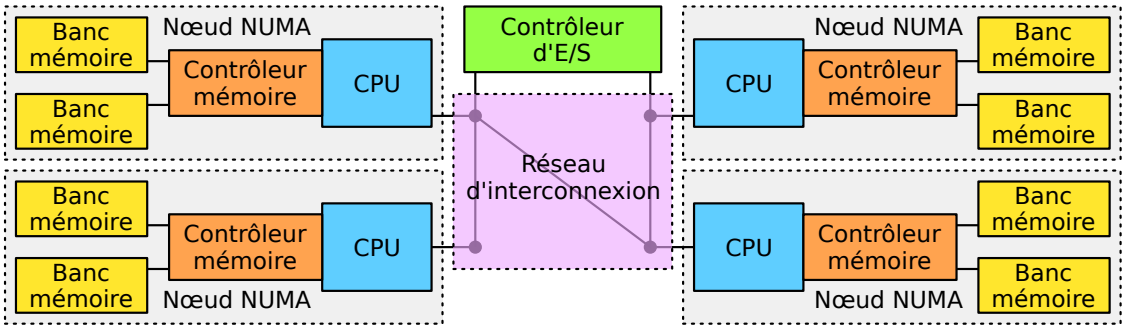

FIGURE 2.2 – Exemple d'architecture NUMA contenant 4 nœuds NUMA interconnectés par un réseau qui détient une structure irrégulière. Les contrôleurs mémoire sont intégrés aux CPU.

tentiellement irrégulière du réseau d'interconnexion). Ainsi, le temps d'accès à un banc mémoire dépend de l'unité de calcul qui y accède, contrairement aux architectures UMA. De même, la proximité du contrôleur entrées/sorties influe sur son temps d'accès. On parle alors d'*affinité* avec les unités de calcul : une unité de calcul peut accéder plus rapidement à sa mémoire que les autres, tout comme certaines unités peuvent communiquer plus rapidement avec le contrôleur d'entrées/sorties que d'autres (étant donné que le réseau d'interconnexion peut avoir une structure irrégulière). Un exemple d'architecture NUMA est présenté sur la figure 2.2.

Processeurs multi-cœurs et hiérarchie mémoire

Les processeurs actuels (p. ex. CPU) contiennent plusieurs cœurs de calcul permettant d'exécuter chacun un flot d'instructions séquentiel. Leur agencement au sein d'un processeur respecte souvent l'architecture SMP.

Dans certains processeurs, les cœurs sont dotés d'un *cache* L1 : une mémoire locale très rapide (car à proximité des cœurs) contenant une copie cohérente d'une partie de la mémoire centrale partagée. Cela permet de réduire les transferts depuis la mémoire centrale partagée, en travaillant sur une copie locale de la mémoire, palliant ainsi l'impact de la latence et du nombre de transferts. En effet, de 1980 à 2004, les performances des processeurs (c.-à-d. FLOPS) se sont améliorées en moyenne de 55 % par an, alors que le débit des mémoires n'a augmenté que de 27 % par an et leur latence de 7 % par an [95]. Ce problème connu sous le nom de *Memory Wall* est aujourd'hui toujours d'actualité et rend l'élaboration de code HPC plus difficile que jamais [84].

Certains processeurs contiennent, en plus d'un cache L1, des caches (p. ex. L2 ou L3) parfois partagés entre les cœurs afin qu'ils puissent notamment communiquer plus rapidement entre eux que par la mémoire centrale. Il existe ainsi plusieurs niveaux de cache (c.-à-d. L1, L2, L3, etc.) formant une hiérarchie de la mémoire cache, allant de petits caches, très rapides, très proches des cœurs de calcul et coûteux, à de gros caches beaucoup plus lents, plus proches de la mémoire et moins onéreux. La plupart des processeurs possèdent deux ou trois niveaux de cache. La figure 2.3

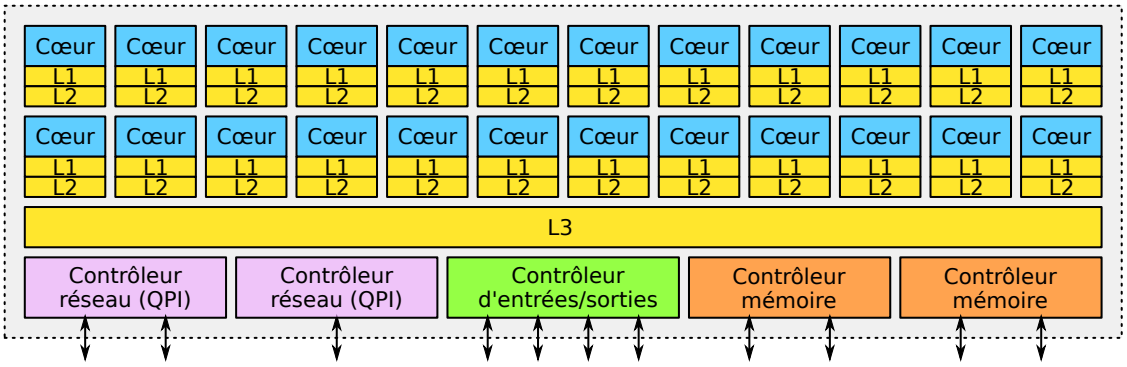

FIGURE 2.3 – Exemple de processeur (Intel Xeon E7-8890V4) contenant 24 cœurs et 3 niveaux de cache (L1, L2 et L3). Le processeur utilise un bus pour communiquer entre les cœurs et les caches (double anneau bidirectionnel).

présente un exemple de processeur central disposant de 24 cœurs et 3 niveaux de cache.

Les mémoires caches travaillent à la granularité d'une *ligne de cache*. Il s'agit de la taille des blocs de données consécutifs chargés ou stockés en mémoire (p. ex. 64 octets). Toute manipulation de données implique donc un multiple de cette taille.

Due à son coût, la taille des caches des processeurs est bien plus petite que celle de la mémoire centrale. Dès lors, plusieurs zones de la mémoire centrale peuvent correspondre à une seule ligne de cache et entrer en conflit. Lorsqu'un conflit survient, il peut causer ou non l'éviction d'une ligne de cache. On parle alors de *défaut de cache* (c.-à-d. *cache miss*). Le choix de l'éviction d'une ligne de cache dépend de la stratégie de remplacement de ligne de cache mise en place par le processeur [66] (p. ex. FIFO, LRU, LFU, etc.).

Pour utiliser efficacement les caches présents sur ces architectures, il est nécessaire de faire attention à la *localité* des accès, autant spatialement (accès à des données contiguës ou proches en mémoire) que temporellement (accès à des mêmes données plusieurs fois).

La conception de processeurs disposant un grand nombre de cœurs introduit de nouveaux défis faisant naître une nouvelle classe de processeurs : les *many-cœurs*. En effet, les architectures SMP possèdent des problèmes inhérents de passage à l'échelle : élaborer des caches partagés par un grand nombre de cœurs et mettre en place des mécanismes de communication efficace entre eux tout en assurant la cohérence des caches et la fluidité des accès mémoire se révèle en pratique particulièrement difficile. Ainsi, les concepteurs de processeurs se tournent aujourd'hui vers des architectures NUMA [2] mais tente de limiter leur effet en se rapprochant d'architecture UMA autre que des SMP [3].

2. AMD annonce en 2010 des processeurs *Opterons* disposant de deux nœuds NUMA. En 2017, AMD lance sa gamme *EPYC* qui contient quatre nœuds NUMA

3. Intel annonce en juin 2017 que ses nouveaux processeurs *Skylake-SP* vont disposer d'une topologie en grille 2D et d'un accès à la mémoire plus uniforme que les générations précédentes.

Systèmes many-cœurs

Les *many-cœurs* sont des processeurs qui disposent d'un grand nombre de cœurs simples (de quelques dizaines à plusieurs centaines). Ces processeurs sont capables de supporter des débits mémoire souvent plus importants au prix d'une latence plus élevée et d'une exécution séquentielle moins efficace. Pour cela, les many-cœurs peuvent disposer de réseaux d'interconnexion complexes, d'une hiérarchie de cache profonde abandonnant parfois la mise en place de caches partagés ou même le mécanisme de cohérence des caches. Ils peuvent aussi embarquer leur propre mémoire généralement beaucoup plus efficace. Ce type de processeur peut introduire des effets NUMA. C'est le cas notamment des *Opteron* d'AMD ou des *Xeon Phi* d'Intel, bien que ce dernier limite leurs effets (NUMA) en permettant de configurer la manière dont les cœurs accèdent à la mémoire ainsi que la façon dont les cœurs s'organisent et communiquent (p. ex. mode de la *MCDRAM* et mode de clustering) dans les processeurs récents de la gamme (p. ex. *Knights Landing*). Ces processeurs sont amenés à se démocratiser dans un futur proche, tout comme les accélérateurs de calcul tels que les GPU.

Accélérateurs de calcul et systèmes hétérogènes

Apparus dans les années 1970, les *accélérateurs graphiques* (GPU) sont des coprocesseurs de calcul spécialisés permettant d'effectuer très rapidement des opérations graphiques (p. ex. rendu d'image 3D et traitements vidéo). Poussés par l'industrie des jeux vidéo, ces accélérateurs capables de délivrer une puissance de calcul gigantesque se sont petit à petit généralisés afin de devenir programmables. On parle alors de *GPGPU* (calcul généraliste sur des GPU). La généralisation de ces coprocesseurs a amorcé une utilisation à large échelle dans des applications pouvant réaliser des opérations massivement vectorielles en parallèle. C'est notamment le cas des applications de calcul de haute performance. D'autres types d'accélérateurs existent tels que les *circuits logiques programmables* (FPGA).

La démocratisation de ces coprocesseurs est freinée par des limitations en grande partie inhérentes aux architectures hétérogènes. En effet, les coprocesseurs possèdent souvent un jeu d'instructions très différent des processeurs généralistes. De plus, ils doivent être contrôlés par les processeurs centraux (p. ex. contrôle d'une mémoire dédiée à l'accélérateur ainsi que des transferts). Enfin, la communication entre les processeurs centraux et les coprocesseurs se fait par un réseau d'interconnexion relativement lent (p. ex. PCI, NVLink, etc.). La figure 2.4 montre un exemple d'architecture matérielle disposant d'un GPU.

Afin de réduire la contention du réseau d'interconnexion, certains processeurs intègrent directement en leur sein le circuit intégré d'un coprocesseur formant un unique processeur disposant d'une architecture interne hétérogène. Par exemple, c'est le cas du processeur *Cell* (utilisé notamment dans la *PlayStation 3* et des anciens supercalculateurs tels que *Roadrunner*) et des processeurs AMD Fusion (p. ex. utilisés entre autres dans des consoles de jeux vidéo telles que la *PlayStation 4*). Néanmoins, ces types de processeurs sont utilisés avec parcimonie dans les systèmes

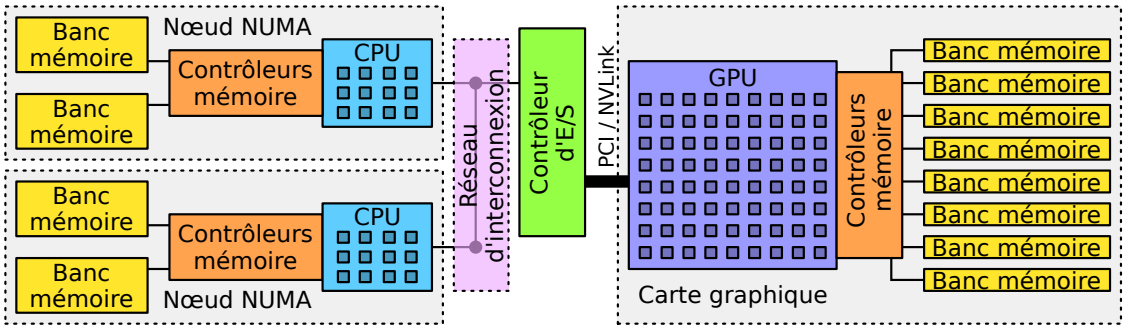

FIGURE 2.4 – Exemple d'architecture NUMA hétérogène contenant deux processeurs et une carte graphique.

HPC à cause de la complexité à les programmer.

2.1.3 Architectures à mémoire distribuée

Les *architectures à mémoire distribuée* des systèmes HPC consistent à connecter un ensemble de nœuds de calcul (c.-à-d. machines à mémoire partagée) entre eux par le biais d'un réseau d'interconnexion. Dans ces architectures, chaque nœud a accès uniquement à sa propre mémoire locale. Les nœuds de calcul peuvent s'échanger explicitement des messages à travers le réseau d'interconnexion pour communiquer entre eux.

Les *grappes de calcul* (c.-à-d. *clusters*) sont une classe d'architectures à mémoire distribuée dans laquelle les nœuds de calcul sont indépendants et forment un ensemble homogène interconnecté par un réseau haut débit.

Les *supercalculateurs* sont une classe d'architectures dédiées au calcul de haute performance construite de manière à n'être qu'une seule et unique machine. Aujourd'hui, les supercalculateurs ont principalement des architectures à mémoire distribuée dans lesquelles les nœuds de calcul (pas nécessairement indépendants) sont très majoritairement homogènes et sont reliés par un réseau d'interconnexion structuré de très haut débit disposant d'une faible latence.

D'autres classes d'architectures à mémoire distribuée existent. On peut notamment citer les *grilles de calcul* qui agrègent un ensemble de grappes ou de supercalculateurs distants géographiquement et souvent hétérogènes. On peut aussi citer les *nuages* qui fournissent une abstraction d'un ensemble de ressources de calcul (souvent issues d'une grappe) sous la forme de services à la demande.

2.1.4 Conclusion

Dans cette section, nous avons tout d'abord évoqué l'évolution des plateformes dédiées au calcul de haute performance à travers le temps, puis nous avons présenté une classification en deux parties des architectures parallèles : d'une part les architectures à mémoire partagée et d'autre part les architectures à mémoire distribuée qui agrègent des machines à mémoire partagée.

Nous avons pu voir que les architectures de haute performance d'aujourd'hui incroyablement complexes et variées fournissent des formes diverses de parallélisme qui opèrent à différents niveaux. En effet, ces architectures sont généralement composées d'un ensemble de nœuds de calcul interconnectés par un réseau singulier. Les nœuds de calcul contiennent eux-mêmes un ensemble de processeurs multi-cœurs superscalaires vectoriels disposant de plusieurs niveaux de caches rapides (partagés ou non) auxquels sont souvent ajoutés des accélérateurs de calcul tels que des GPU. Les processeurs et accélérateurs peuvent communiquer grâce à différents types de réseaux (p. ex. QPI, PCI-Express, Infiniband, etc.) ou grâce à la mémoire avec des temps d'accès potentiellement dépendants de l'unité de calcul qui y accède.

Tout naturellement, nous pouvons nous interroger sur la manière d'exploiter efficacement ces plateformes, en particulier les supercalculateurs sur lesquels ce manuscrit se concentre. Mais avant, il convient de fixer clairement les objectifs que nous voulons atteindre : quelles formes d'applications s'exécutent sur ces plateformes, quelles sont celles considérées et quelles sont les caractéristiques mises en avant par ces applications ?

2.2 Applications

Les applications HPC consomment une très grande quantité de puissance de calcul et nécessitent parfois une quantité de mémoire imposante dans le but de résoudre des problèmes souvent complexes. Elles sont donc exécutées tout naturellement sur des supercalculateurs afin de pouvoir subvenir à leurs besoins.

Cette section présente tout d'abord quels sont les objectifs des applications HPC et quels sont les défis à relever. Puis, elle présente la variabilité des domaines applicatifs et se concentre plus particulièrement sur les applications visant à résoudre des équations différentielles partielles. La section se focalise ensuite sur un exemple d'application étudiée dans ce manuscrit : Gysela, une simulation de physique des plasmas pour la fusion nucléaire. Enfin, une dernière partie conclut cette section.

2.2.1 Conception et analyse d'applications parallèles

Afin d'obtenir des résultats le plus rapidement possible, les applications HPC sont massivement parallèles, c'est-à-dire qu'elles sont capables de fournir une quantité de travail pouvant être réparti (équitablement) sur un grand nombre d'unités de calcul. Cependant, être massivement parallèle n'est pas suffisant pour minimiser le temps de complétion (c.-à-d. temps d'exécution total) : il est aussi nécessaire que les applications HPC *passent à l'échelle*.

Pour évaluer le passage à l'échelle d'une application, on utilise des métriques d'*accélération*. On définit l'accélération parallèle par la relation $\frac{t_{séquentiel}}{t_{parallèle}}$. Lorsque la quantité de données traitées est constante, on parle d'accélération forte. Dans le cas contraire, lorsque la taille des données croît en fonction du nombre d'unités de calcul utilisées, on parle d'accélération faible. Une application passant parfaitement

à l'échelle devrait donc posséder une accélération forte linéaire en nombre d'unités de calcul utilisées.

En pratique, il est difficile d'atteindre une accélération forte linéaire. En effet, la *loi d'Amdahl* [7] stipule que cette accélération est fondamentalement limitée par la proportion du code source exécuté en parallèle par une application (p. ex. une application parallélisée à 95 % ne peut avoir une accélération supérieure à 20). Ainsi l'exécution séquentielle d'une toute petite portion de code source d'une application peut rapidement constituer un goulot d'étranglement considérable. Plusieurs facteurs peuvent en être à l'origine. D'une part, les algorithmes utilisés peuvent fournir un *degré de parallélisme* insuffisant. Pour remédier à ce problème, les concepteurs d'applications peuvent avoir recours à plusieurs formes de parallélisme (*cf.* section 2.3.1). D'autre part, les opérations réalisées peuvent être interdépendantes et introduire des *synchronisations*, *communications* et l'*exclusions mutuelles* de l'usage des ressources (que ce soit au niveau applicatif ou matériel).

Afin d'atténuer ces phénomènes, la *charge de travail* (p. ex. quantité de données traitées) d'une application peut être adaptée en fonction du nombre d'unités de calcul utilisées. La *loi de Gustafson* [59] montre que l'on peut traiter une charge de travail plus grande en temps constant en augmentant le nombre d'unités de calcul afin de pallier les limitations imposées par la loi d'Amdahl.

L'augmentation de la charge de travail peut se faire soit en augmentant le volume des données traitées, soit en effectuant des traitements plus coûteux sur un même volume de données. La première solution suppose qu'il soit possible de prélever ou synthétiser des volumes de données plus imposants, que l'application soit en mesure de les traiter et que cela ait un intérêt. La seconde peut être atteinte en adaptant les algorithmes employés, notamment en utilisant des méthodes de résolution plus précises, bien que cela ne soit pas toujours possible (p. ex. précision limitée par les jeux de données, quantité de mémoire physique limitante).

La conception d'application HPC demande donc de mettre au point des algorithmes et méthodes permettant de résoudre un problème en répartissant (équitablement) une grande quantité de travail sur un grand nombre d'unités de calcul. Pour y parvenir, il est souhaitable d'employer différentes formes de parallélisme, de minimiser les sources de sérialisation telles que les communications et les synchronisations ainsi que d'ajuster la taille des jeux de données traitées lorsque c'est possible.

2.2.2 Domaines applicatifs

Les applications HPC traitent de domaines très divers : climatologie [63], biologie moléculaire et cellulaire (p. ex. neurologie [82], génomique [6], dynamique moléculaire [62]), physique (p. ex. physique des particules [1], fusion nucléaire [58], cosmologie [113]), sciences de l'ingénieur et technologies (p. ex. aérospatiale, énergies renouvelables), etc. La diversité des domaines applicatifs laisse entrevoir des algorithmes, méthodes et structures de données hétéroclites [97]. Par exemple, certaines applications opèrent sur des structures de données régulières telles que des tableaux multidimensionnels, d'autres opèrent sur des arbres ou des graphes ou encore sur

des structures plus spécifiques (liées aux algorithmes utilisés).

Parmi l'ensemble des applications HPC, nous nous intéressons dans ce manuscrit à un sous-ensemble significatif qui effectue des simulations numériques visant à résoudre des *équations aux dérivées partielles*.

2.2.3 Équations aux dérivées partielles

Les *équations aux dérivées partielles* (EDP) sont des équations mathématiques faisant intervenir des relations entre des fonctions inconnues et leurs dérivées par rapport à un ensemble de variables. Elles sont utilisées pour modéliser un système (p. ex. climat planétaire) afin d'en étudier le comportement (p. ex. évolution de la température globale). L'expression 2.1 est un exemple d'équation aux dérivées partielles : c'est l'équation de la chaleur appliquée à un espace de dimension deux. Dans cette équation, une fonction $u(x, y, t)$ est dérivée deux fois respectivement selon les paramètres x et y (p. ex. dimensions d'espace). La somme de ces deux dérivées est égale au produit de la dérivée de u selon la variable t (p. ex. dimension temporelle) par un facteur $\frac{1}{\alpha}$, où α est une constante décrivant la diffusion de la chaleur dans le milieu considéré. Toute la difficulté est de trouver l'expression de u ou au moins d'obtenir une approximation de la fonction. Certaines EDP possèdent une dimension temporelle implicite : la dimension est omise dans l'expression bien qu'elle soit nécessaire à la description du problème visé.

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = \frac{1}{\alpha}\frac{\partial u}{\partial t} \tag{2.1}$$

La résolution analytique des EDP est complexe, en particulier lorsque la relation exprimée n'est pas linéaire. Ainsi un grand nombre d'EDP n'admet pas de solutions analytiques connues à ce jour (p. ex. équations de *Navier-Stokes* utilisées pour décrire le comportement des fluides newtoniens). Dans ce cas, une approximation de la solution peut être trouvée en utilisant des méthodes numériques. Il est alors nécessaire de *discrétiser* le domaine d'application du problème (c.-à-d. espace d'application mathématique des variables considérées).

Le domaine d'une EDP comprend souvent deux parties : spatiale et temporelle. La discrétisation du domaine spatial forme un *maillage* : un graphe sur lequel sont appliquées des données (c.-à-d. valeurs des variables attachées aux nœuds ou aux arcs du graphe). On distingue deux sortes de maillages : les *maillages réguliers* (p. ex. grilles multidimensionnelles) et les *maillages irréguliers* (p. ex. graphes quelconques). La discrétisation du domaine temporel forme un ensemble de pas de temps (p. ex. itérations) représentant l'évolution de la simulation au cours du temps.

Une fois le domaine discrétisé, on applique une méthode de résolution adaptée à l'EDP considérée, on parle alors de *schémas numériques*. Le choix d'un schéma numérique particulier dépend de ses caractéristiques : coût en nombre d'opérations, rapidité de la convergence, précision des résultats en fonction des entrées, degré de parallélisme fourni, etc. Une fois ce choix fait, le schéma numérique doit être implémenté de telle manière à être exécuté efficacement sur des supercalculateurs.

2.2.4 Conception et maintenance

La *conception* d'application HPC visant à résoudre des EDP fait intervenir des acteurs excellant dans des domaines très variés. En effet, la modélisation du problème nécessite des spécialistes du domaine de l'application, la discrétisation du problème fait intervenir des *numériciens*, enfin la parallélisation des schémas numériques fait intervenir des informaticiens. Une bonne interaction entre les acteurs est donc fondamentale à la conception de telles applications, en particulier au niveau de sa mise en œuvre.

La *maintenance* de ces applications peut introduire des modifications parfois profondes à tous les niveaux. Par exemple, durant cette phase de maintenance, la modélisation peut être adaptée afin de considérer de nouveaux facteurs négligés jusque-là. De nouveaux schémas numériques peuvent aussi être ajoutés pour améliorer par exemple la précision des résultats ou les performances de l'application. De même, la méthode de parallélisation peut être adaptée afin de supporter des architectures plus récentes ou encore pour mieux passer à l'échelle. Les applications HPC évoluent donc continuellement au risque de devenir obsolètes autrement.

L'augmentation de la puissance de calcul des supercalculateurs permet d'affiner la précision des modèles ainsi capables de prendre en compte plus de facteurs. Cependant, cette amélioration se fait souvent au prix de codes sources plus complexes, atteignant rapidement plusieurs centaines de milliers, voire millions de lignes de codes. Cette amélioration peut aussi se faire par le *couplage de codes sources* existant permettant ainsi de prendre en compte des phénomènes issus de domaines variés (p. ex. couplage de codes d'applications de biologie et de physique). Il est ainsi nécessaire de mettre en place des méthodes visant à minimiser le coût de la maintenance de l'application tout en facilitant l'interaction entre les codes sources existants.

Gysela [58] est un exemple d'application HPC traitant de multiples EDP. Cette application sert de cas d'étude pour évaluer l'approche proposée dans ce manuscrit. La structure de Gysela et son fonctionnement général sont présentés dans la section suivante. Notez que l'approche proposée dans ce manuscrit (dans les chapitres 3 et 4) n'a pas pour vocation de cibler uniquement ce type d'applications.

2.2.5 Focus : Gysela

Gysela [58, 23] est une application HPC de simulation de physique des plasmas par confinement magnétique pour la fusion nucléaire. Développée au sein de l'Institut de recherche sur la fusion magnétique (IRFM) du Commissariat à l'énergie atomique et aux énergies alternatives (CEA) dans les années 2000. Le code source de l'application est aujourd'hui constitué d'approximativement 60 000 lignes de code dont 95 % en FORTRAN et 5 % en C. Gysela est capable de passer à l'échelle jusqu'à une centaine de milliers de cœurs [57]. Avant de détailler le fonctionnement de Gysela et les enjeux de maintenabilité que cette application pose, il est nécessaire de comprendre le cadre dans lequel l'application se place : la *physique des plasmas*.

Physique des plasmas et fusion nucléaire

Dans un monde où les besoins en énergie ne cessent de croître, la fusion nucléaire constitue une alternative viable aux énergies fossiles promettant de fournir une quantité d'énergie gigantesque à partir de ressources en quantité quasi illimitée et un impact environnemental négligeable comparé aux sources d'énergies alternatives actuelles.

La *fusion nucléaire* est un processus faisant intervenir des noyaux atomiques légers qui fusionnent pour former des noyaux plus lourds et plus stables. Ce phénomène se déroule naturellement au sein des étoiles telles que le Soleil. Il se distingue de la *fission nucléaire* qui effectue le processus inverse.

Pour réaliser des réactions de fusion, une des approches consiste à créer un *plasma* à très haute température qui est *confiné magnétiquement*. Le plasma est un état de la matière constitué d'un mélange d'électrons (c.-à-d. charges électriques négatives) et d'ions (c.-à-d. espèces chimiques chargées) de charges positives circulant librement. Il a notamment la propriété de réagir à des champs magnétiques forts. La température du plasma nécessaire à la fusion est telle qu'aucune matière solide ne pourrait y résister. Ainsi le plasma est confiné par un champ magnétique évitant tous les contacts avec les bords de l'enceinte. L'enceinte de confinement peut être un tore dans lequel le plasma circule. On parle alors de *tokamaks* (p. ex. ITER).

Dans les tokamaks, la différence de température entre le cœur de l'enceinte et les bords est telle que cela cause des instabilités dans le plasma [4] (p. ex. turbulences). L'instabilité du plasma est un frein au contrôle des réactions de fusion. Un des objectifs du code Gysela est donc de simuler les comportements électrostatiques issus des gradients de température des ions dans le plasma dans le but de les analyser pour mieux les comprendre.

Architecture de l'application

L'espace du tokamak dans lequel les particules du plasma se déplacent est décrit par la figure 2.5. C'est un *espace torique en trois dimensions*. Il est décrit par le système de coordonnées (r, θ, ϕ) dans lequel (r, θ) sont des coordonnées polaires d'une section poloïdale du tore et ϕ une coordonnée angulaire dans la direction toroïdale. Les particules se déplacent dans le tore en suivant globalement des lignes de champ magnétique hélicoïdales.

Dans Gysela, l'ensemble des particules qui constituent le plasma du tokamak est modélisé sous la forme d'une fonction de distribution $\bar{f}$ définissant le nombre moyen de particules dans un état donné (position et vitesse) à un instant t. Cette fonction est définie dans un espace en cinq dimensions. Trois dimensions (r, θ, ϕ) correspondent à l'espace physique. Deux dimensions $(v_\parallel, \mu)$ correspondent à l'espace des vitesses. La valeur fonction en tout point évolue dans le temps. L'utilisation d'un modèle *gyrocinétique* permet de ne considérer que deux dimensions en vitesse : $v_\parallel$ aligné sur les lignes de champ et μ. Ce choix permet de réduire significativement

4. La présence d'un gradient de température n'est pas la seule source possible d'instabilité : les gradients de densité ou de champ magnétique du plasma jouent aussi un rôle important.

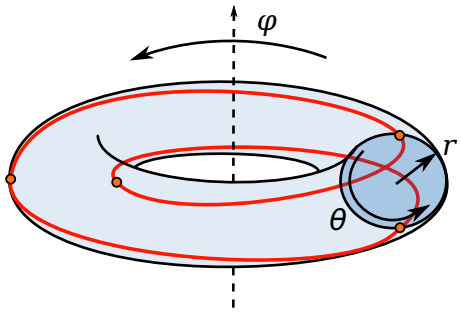

FIGURE 2.5 – Dimensions (r, θ, ϕ) de Gysela représentant un tokamak (bleu) et trajectoire des lignes de champs magnétique (rouge).

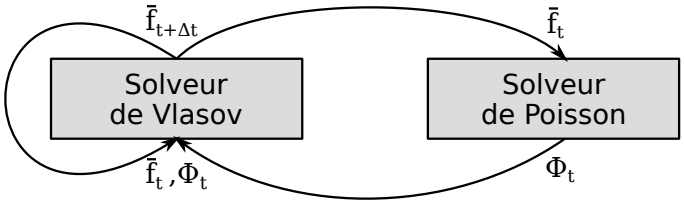

FIGURE 2.6 – Fonctionnement général et simplifié de Gysela.

le temps et la mémoire nécessaire pour la simulation en traitant cinq dimensions plutôt que six. Ainsi, la fonction $\bar{f}(r, \theta, \phi, v_\parallel, \mu, t)$ décrit la densité de particules à la position (r, θ, ϕ) détenant une vitesse $(v_\parallel, \mu)$ à un instant t. Tout comme les particules, le champ de potentiel électrique est modélisé par la fonction $\Phi(r, \theta, \phi, t)$ décrivant la force du champ électrique en tout point de l'espace.

Une exécution de Gysela est composée de trois étapes : une initialisation (lecture et génération des données initiales), des itérations (pas de temps) et une finalisation. Chaque itération de temps est le résultat du couplage de deux *solveurs* principaux (entité logicielle visant à résoudre des EDP) résolvant chacun un type d'équation spécifique : *Vlasov* [116] et *Poisson*. Sommairement, le solveur Vlasov a pour objectif de déplacer les particules du plasma dans le tokamak en fonction du champ de potentiel électrique (*advection*) et le solveur Poisson vise à calculer le champ de potentiel électrique à partir de la nouvelle configuration du plasma. Périodiquement, Gysela effectue des *diagnostics* : l'application extrait des grandeurs physiques issues de l'état du plasma et du champ de potentiel électrique. Cette opération permet de stocker de manière régulière des informations de petites tailles pour analyse par les physiciens. Cette approche est beaucoup plus efficace que s'il fallait stocker entièrement la fonction de distribution $\bar{f}$. La figure 2.6 présente une vue générale et simplifiée du fonctionnement de Gysela.

La résolution des équations de Vlasov est de loin l'opération la plus gourmande en puissance de calcul de toute l'application.

Le solveur Vlasov de Gysela utilise un schéma numérique *semi-lagrangien*. Dans ce type de schéma, on cherche à calculer en tout point du maillage des caractéristiques à un instant t à partir d'une interpolation des caractéristiques déjà calculées à l'instant $t-\Delta t$. Ces caractéristiques peuvent être par exemple la fonction de distribu-

tion $\bar{f}$. Pour cela, deux opérations sont effectuées pour chaque point du maillage : le calcul de *pieds des caractéristiques* et une *interpolation*. En premier lieu, à la position X_{dst} sur le maillage et à l'instant t, on détermine l'origine de la caractéristique (pieds des caractéristiques) à partir des lois du mouvement (p. ex. forces électrostatiques). Le résultat est la position X_{src} d'une caractéristique à l'instant $t - \Delta t$ n'appartenant pas toujours au maillage. Ensuite, on calcule la caractéristique du point du maillage à l'instant t en interpolant les caractéristiques des points du maillage aux environs de la position X_{src}. Plusieurs méthodes d'interpolation peuvent être utilisées telles qu'une interpolation basée sur des *splines* ou un polynôme de *Lagrange*.

Dans Gysela, le solveur de Vlasov est basé sur une décomposition d'opérateurs utilisant la méthode du *splitting de Strang* [109]. L'advection globale est remplacée par une séquence d'advections unidimensionnelles ou bidimensionnelles. Chacune de ces advections peut être exécutée massivement parallèle puisqu'elle est appliquée sur l'ensemble de la fonction de distribution (cinq dimensions). L'importante charge de calcul est distribuée sur les nœuds de calcul. Les advections sont entrelacées par des transpositions afin d'améliorer la localité des calculs.

Le calcul de l'advection constitue la majeure partie du temps de calcul de l'application. Pour cette raison, le cas de l'advection constitue un sujet d'analyse intéressant et est étudié dans ce manuscrit.

Enjeux

La mise en œuvre de Gysela constitue un défi de taille. L'un des plus gros enjeux consiste à mettre au point une mise en œuvre qui soit efficace et maintenable sur le long terme.

En premier lieu, l'application vise à effectuer des calculs complexes sur des volumes de données imposants (p. ex. de l'ordre du téraoctet à l'exaoctet). Ces calculs consomment une quantité de mémoire élevée (p. ex. de l'ordre de l'exaoctet) et un temps allant de quelques minutes à plusieurs semaines, voire plusieurs mois sur des supercalculateurs actuels dotés de centaines de milliers de cœurs [23, 57]. L'ajout d'aspects physiques, actuellement non pris en compte dans la simulation, nécessiterait une puissance de calcul bien supérieure à l'exaflop. Cependant, cette puissance de calcul n'est pas encore à la portée des supercalculateurs actuels, capables de fournir au mieux une centaine de pétaflops [114].

Ainsi, les concepteurs de Gysela portent un intérêt tout particulier à l'optimisation de la mise en œuvre afin que les physiciens puissent obtenir des résultats précis en des temps raisonnables.

De plus, l'application évolue dans le temps afin d'améliorer continuellement la précision des méthodes numériques ainsi que l'efficacité des méthodes employées : des schémas numériques sont alors remplacés par d'autres (p. ex. méthode d'interpolation), la structure du maillage est adaptée et la méthode de parallélisation est remise en question. Cependant, plusieurs problèmes peuvent rendre difficile la maintenance de l'application et donc constituer des freins au développement de l'application. Par exemple, la multiplication des préoccupations entre des experts de domaines différents (physiciens, numériciens et informaticiens) dans un même

code source monolithique rend difficile les modifications du code source car elles nécessitent une expertise dans plusieurs domaines. De plus, les algorithmes sont parfois fortement liés afin de pouvoir les coupler efficacement. Malheureusement, ces liens sont implicites et le remplacement d'un algorithme par un autre nécessite des modifications profondes du code source.

2.2.6 Conclusion

Dans cette section, nous avons tout d'abord entrevu les objectifs et défis des applications HPC qui consistent à appliquer des traitements complexes sur des gros volumes de données, passant à l'échelle sur des architectures massivement parallèles. Nous avons ensuite vu que ces applications sont de natures diverses et nous nous sommes concentré sur les applications numériques visant à résoudre des équations aux dérivés partielles. Nous avons évoqué l'importance du développement et de la maintenabilité d'une application en illustrant les problèmes qui peuvent apparaître dans le cas des applications traitant de la résolution d'équations aux dérivées partielles (bien que ces applications n'aient pas le monopole des problèmes de maintenance). Nous avons ensuite présenté l'application Gysela, un cas d'étude de ce manuscrit : son contexte, son architecture, son fonctionnement ainsi que les enjeux liés à la maintenance et l'évolution de ce code. Plus spécifiquement, nous avons isolé un cas d'utilisation, l'advection, qui est présenté et évalué dans le chapitre 6.

Les enjeux précédemment identifiés nous conduisent tout naturellement à nous interroger sur la manière de programmer efficacement les applications HPC et de les structurer de façon à minimiser les coûts de développement et de maintenance. Ces deux sujets font respectivement l'objet des deux chapitres suivants.

2.3 Paradigmes et modèles de programmation

Cette section présente les paradigmes et modèles de programmation parallèle existants pouvant être utilisés afin de concevoir des applications qui exploitent efficacement les ressources matérielles mises à disposition. Elle s'intéresse donc tout particulièrement aux paradigmes et approches dédiés au calcul de haute performance. Tout d'abord, nous décrivons les deux formes de parallélismes majoritairement utilisées par les applications HPC : le parallélisme de données et le parallélisme de tâches. Ensuite, nous présentons les stratégies d'interactions parallèles issues du matériel telles que la mémoire partagée et le passage de messages. Enfin, nous terminerons cette section par une analyse détaillée des approches existantes pour programmer des applications HPC, ce qui fait leurs forces et leurs faiblesses. Cette analyse se termine par une discussion.

2.3.1 Parallélisme de données et de tâches

Au niveau logiciel, on peut distinguer deux grandes formes de parallélisme présentes dans les applications HPC : le *parallélisme de données* et le *parallélisme de*

tâches.

Parallélisme de données

Le *parallélisme de données* est de loin la forme la plus utilisée dans les applications HPC. Elle consiste à distribuer des données sur des fils d'exécution qui réalisent tous une même opération (sur des morceaux de données différentes). Pour cela, les données d'une application sont partitionnées.

Le *partitionnement* de données consiste à diviser une donnée d'une application en sous-ensembles plus petits appelés *fragments*. Chaque fil d'exécution se voit attribuer un ensemble de fragments. Lorsque le nombre de fragments est plus grand que le nombre de fils d'exécution, on parle de *surdécomposition*. Dans le cas des applications numériques traitant de la résolution d'équations aux dérivées partielles, ce sont les maillages des données qui sont partitionnés. La structure du maillage ainsi que le lien entre le maillage et les données permet de déduire la répartition des données sur les différents fils d'exécution.

Dans beaucoup d'applications HPC, le partitionnement forme des *fragments* contiguës et de taille quasi homogène. En effet, la contiguïté des fragments permet de diminuer la taille des données partagées entre les unités de calcul et ainsi de réduire les échanges nécessaires ou une sérialisation des accès. L'homogénéité de la taille des fragments vise à assurer une répartition équitable de la charge entre les fils d'exécution lorsque le temps de calcul dépend uniquement de la taille des données traitées. On parle alors de *calculs réguliers*.

Dans certains cas les fragments sont disjoints, dans d'autres cas ils se recouvrent (aussi appelés *ghost*). L'utilisation de fragments disjoints évite le partage de données ou de réplication qui peut introduire des problèmes de cohérence des données, des communications supplémentaires entre les fils d'exécution ou encore une sérialisation de l'accès aux données.

Le degré de parallélisme offert par cette méthode dépend principalement de la structure et de la taille des données. Ainsi, il est parfois nécessaire de redistribuer les données entre les fils d'exécution afin d'améliorer l'efficacité de la méthode de parallélisation en vue des calculs à effectuer (afin notamment d'améliorer leur localité). Une *redistribution* des données (c.-à-d. *repartitionnement*) consiste à passer d'un partitionnement à un autre pouvant engendrer des communications entre les fils d'exécution (synchronisation, transfert de données, etc.).

Parallélisme de tâches

Le parallélisme de tâches consiste à diviser un calcul d'une application en un sous-ensemble de calculs (potentiellement différents) appelés tâches.

Les tâches peuvent être dépendantes d'autres tâches. On parle de *dépendance de contrôle* entre deux tâches t_1 et t_2 lorsque t_1 requiert la précédence de l'exécution de t_2. On parle de *dépendance de données* entre t_1 et t_2, lorsque t_1 nécessite des données produites par t_2.

Une tâche peut être exécutée sur les ressources disponibles (p. ex. unité de calcul). Elle consomme ainsi des ressources pendant un temps donné. On appelle ordonnancement (ou placement), le processus visant à affecter chaque tâche à des ressources disponibles tout en respectant les dépendances entre les tâches. Une telle opération vise à utiliser efficacement les ressources disponibles en exécutant un grand nombre de tâches en parallèle de manière à minimiser un critère tel que le temps d'exécution total (p. ex. réduction de l'impact des communications issues des dépendances, exécution consécutive de tâches travaillant sur les mêmes données).

La nature des tâches et des dépendances résulte du modèle de programmation parallèle étudié. On peut ainsi distinguer dans cette forme de parallélisme plusieurs catégories de modèles tels que les modèles de contrôle parallèle, les modèles de flux de données ou encore les modèles de graphes de tâches.

Discussion

Le parallélisme de données et le parallélisme de tâches sont deux formes de parallélisme pouvant être utilisées conjointement dans les applications HPC. Le parallélisme de données est indispensable aux applications HPC car il fournit un degré de parallélisme très large étant donné la taille imposante et la structure des jeux de données traités par les applications du domaine. En plus du parallélisme de données, le parallélisme de tâches permet d'accroître le degré de parallélisme de l'application. En effet, le parallélisme de données introduit des problèmes de passage à l'échelle sur certains types de problèmes (p. ex. factorisation de Cholesky). On s'intéresse donc, dans ce manuscrit, tout particulièrement aux modèles de programmation permettant de décrire les deux formes de parallélisme.

2.3.2 Interaction : mémoire partagée et passage de messages

À partir des classes d'architectures matérielles présentées à la section 2.1, il est possible d'inférer des catégories de modèles de programmation parallèle proches des architectures sous-jacentes afin de les exploiter efficacement. Cependant les modèles proches du matériel ont un faible niveau d'abstraction rendant la programmation de ces architectures relativement complexe. Ainsi, il est possible de programmer les architectures à mémoire partagée avec un modèle dédié, tout comme les machines à mémoire distribuée ou encore les architectures hétérogènes.

Mémoire partagée

En *mémoire partagée*, la programmation à base de verrous est l'une des approches les plus fréquemment utilisées dans les applications. Dans cette approche, un ensemble de fils d'exécution (p. ex. *processus légers* : *threads*) d'une application accèdent à une même mémoire partagée entre eux. Afin d'éviter des opérations concurrentes sur les données en mémoire, les concepteurs d'applications ont recours à des mécanismes d'exclusion d'accès aux données. Ils peuvent, entre autres, utiliser des *verrous* d'exclusion mutuelle (p. ex. *mutex*) afin de mettre en place une *section*

critique accessible par un seul fil d'exécution à la fois. L'utilisation de cette approche peut se révéler complexe car la présence d'erreurs de programmation peut causer des interblocages lors de l'exécution d'une application qui se révèlent être très difficiles à repérer, particulièrement lorsque leur comportement n'est pas déterministe.

D'autres mécanismes de contrôle d'accès aux données existent tels que la *mémoire transactionnelle* [61] permettant à une portion du code source de réaliser un ensemble de lectures et écritures sur des données de manière indivisible à travers des transactions.

Bien que le paradigme de programmation à mémoire partagée soit particulièrement bien adapté aux architectures à mémoire partagée, il est néanmoins possible de l'utiliser sur des architectures à mémoire distribuée. On parle alors de *mémoire partagée distribuée* (c.-à-d. DSM). Cependant, le coût des transferts de données entre les nœuds est non négligeable (p. ex. latence) et doit, par conséquent, être pris en considération par les modèles utilisant cette approche.

Passage de messages

Sur les architectures à mémoire distribuée, le paradigme de programmation issu du matériel le plus naturel est le *passage de messages*. Dans ce paradigme, des fils d'exécution (p. ex. *processus lourds*) communiquent uniquement en envoyant et recevant des messages. De plus, les problèmes de concurrences introduits par le passage de messages sont remplacés par des problèmes de communication.

Bien que cette approche soit particulièrement adaptée aux architectures à mémoire distribuée, elle peut aussi être utilisée efficacement sur des architectures à mémoire partagée. Cependant, cette approche sur les architectures à mémoire partagée peut nécessiter une quantité accrue de mémoire. En effet, étant donné que les fils d'exécution ne peuvent pas partager de données en mémoire, les données sont répliquées en mémoire ou retransmises. Cette dernière opération peut introduire des coûts supplémentaires (lorsque les données sont répliquées).

Hybride

Afin de réduire ces coûts, il est possible de mélanger les approches de passage de messages et de mémoire partagée. On parle alors de programmation hybride. Dans cette approche, plusieurs groupes de fils d'exécution communiquent par passage de messages et les fils d'exécution d'un même groupe communiquent par une mémoire partagée locale au groupe. Cette approche a cependant le défaut d'être plus complexe car elle nécessite de considérer les deux approches.

2.3.3 Modèles de programmation parallèle

Cette section présente une liste non exhaustive de modèles de programmation fréquemment utilisés par les applications HPC. Les modèles de programmation sont regroupés dans cette section par similarité des fonctionnalités qu'ils offrent. Nous

nous intéressons en particulier aux formes de parallélisme et aux stratégies de communications et d'interactions qu'ils fournissent.

Interface de passage de messages

Le modèle de programmation *MPI* [88] (c.-à-d. *Message Passing Interface*) est une norme de passage de messages définissant la syntaxe et la sémantique des opérations de communication entre des fils d'exécutions. MPI supporte les communications point à point par le biais d'opérations d'envois et de réceptions de messages. Ces opérations peuvent être synchrones ou asynchrones et peuvent avoir recours à une mémoire tampon (qui agrège des messages). En MPI, les fils d'exécution d'une application peuvent être regroupés entre eux afin de former un *communicateur*. Un même fil d'exécution peut être inclus dans plusieurs communicateurs. MPI offre un moyen d'effectuer des *communications collectives* : tous les fils d'exécution d'un communicateur sont impliqués dans un ensemble de communications perçu comme une seule communication globale (p. ex. diffusion d'un message, opération de réduction, barrière de synchronisation, etc.). Les versions plus récentes de la norme [89] offrent aussi un moyen d'effectuer des opérations de communication à sens unique (p. ex. put/get) réalisant des accès directs à une zone mémoire d'un fil d'exécution. Depuis les années 1990, l'utilisation de MPI n'a cessé de croître et il est aujourd'hui massivement adopté par les applications HPC.

Espace d'adressage global partitionné

Les modèles *PGAS* (Partitioned Global Address Space) tels que UPC [53], Chapel [33], X10 [34] ou encore XcalableMP [78] considèrent un espace d'adressage global logiquement partitionné entre plusieurs fils d'exécution. Les fils d'exécution peuvent lire ou écrire des données contenues dans tout l'espace d'adressage global, à la manière des architectures à mémoire partagée, à travers des opérations de communication explicites à sens unique (p. ex. put/get) réalisant des accès directs à la mémoire globale. L'espace d'adressage est logiquement partitionné et chaque fil d'exécution peut accéder à un sous-espace local plus rapidement.

Certains modèles tels que *Co-array FORTRAN* [91] manipulent des données à travers une vue locale à chaque fil d'exécution. D'autres modèles comme *HPF* [72] [70] proposent une vue globale des données dans laquelle les données sont distribuées entre les fils d'exécution. Il est possible d'adapter la distribution des données au cours de l'application à l'aide de la redistribution de données. Enfin, des modèles tels que *XcalableMP* proposent les deux formes d'approches.

Fork-Join

Les modèles *fork-join* [36] consistent à diviser, à un instant donné, un fil d'exécution en plusieurs. Les fils d'exécution sont ensuite fusionnés afin d'en reformer un unique. Cette opération peut s'exécuter récursivement dans chacune des branches formées par des divisions successives. L'opération de division des fils d'exécution

fournit intrinsèquement une source de parallélisme. L'opération de fusion quant à elle introduit une synchronisation entre les fils d'exécution produits. En effet, la fusion des branches filles d'une même branche mère requiert la terminaison de toutes les branches filles. Les fils d'exécution peuvent ici être assimilés à des tâches. De nombreuses approches utilisent cette approche telles que la bibliothèque *Threading Building Blocks* [99] qui offre des structures de contrôle parallèle (p. ex. boucles parallèles) ou encore le langage *Cilk* [24] qui propose de décrire le parallélisme en utilisant seulement les deux mots-clefs *spawn* et *sync*.

Modèles de graphes de tâches et supports exécutifs

Les modèles de *graphes de tâches* [9, 52, 31, 73, 120, 105, 16] consistent à construire un ensemble de tâches interdépendantes formant un *graphe acyclique orienté* (c.-à-d. DAG) dans lequel les nœuds sont des tâches et les arcs sont des *dépendances*. Ce graphe est soumis à un ordonnanceur dans le but d'exécuter les tâches sur les ressources disponibles afin de minimiser un ou plusieurs critères (p. ex. temps d'exécution) tout en respectant les dépendances. La nature du lien entre les tâches est déterminée par le type des dépendances. On distingue deux grandes catégories de dépendances : les *dépendances de contrôle* et les *dépendances de données*.

Une *dépendance de contrôle* entre deux tâches t_1 et t_2 implique que t_1 requiert la précédence de l'exécution de t_2. Ainsi deux tâches interconnectées par une dépendance de contrôle ne peuvent jamais s'exécuter simultanément.

Une *dépendance de données* entre t_1 et t_2 implique que t_1 nécessite des données produites par t_2. Dans le cas des dépendances de données, les tâches interdépendantes peuvent mutuellement spécifier le mode d'accès aux données afin de donner des précisions sur les dépendances et permettre des optimisations à l'exécution. Les modes souvent proposés sont *in*, *out* et *inout*. Certains modèles proposent des modes supplémentaires tels que l'accès concurrent en écriture [51] ou encore l'accès exclusif en écriture à une même donnée [9]. Il est possible de déduire de ces modes des contraintes d'exécution qui lient des tâches. Par exemple, lorsqu'une tâche écrit dans des données lues par une autre (c.-à-d. *read after write*), la dépendance est assimilable à une dépendance de contrôle entre les deux tâches. Cependant, lorsqu'une tâche lit des données et qu'une autre doit ensuite écrire dans ces données (c.-à-d. *write after read*), on parle d'*anti-dépendance* car il est possible d'exécuter les tâches simultanément (p. ex. en dupliquant les données et en effectuant un renommage). Le même problème peut aussi se produire lorsque deux tâches accèdent en écriture aux mêmes données.

Les supports exécutifs dynamiques à base de graphes de tâches tels que *StarPU* [9], *Kaapi* [52], *Nanos* [31] ou *Quark* [73] visent à ordonnancer un graphe de tâches sur les ressources disponibles à l'exécution (ordonnancement dit *online*). Le graphe de tâches est construit progressivement par la soumission successive de tâches associées chacune à un ensemble de dépendances. La *soumission de tâches* est une action consistant à fournir à un ordonnanceur les informations définissant une tâche. Ces supports exécutifs divergent sur de nombreux aspects.

La construction du graphe peut se faire explicitement ou implicitement : le support exécutif est libre de manipuler une représentation du graphe telle qu'une forme factorisée comme c'est le cas dans *PaRSEC* [120] (dans le but d'éviter une exploration totale du graphe lors de l'ordonnancement).

Certains modèles supportent les accélérateurs de calcul, c'est le cas notamment de StarPU. Ce modèle met en place une mémoire partagée distribuée afin de gérer la cohérence des données entre les accélérateurs et l'hôte gérant ainsi automatiquement les transferts de données. Il fournit une couche logicielle de gestion des données qui permet de définir des types de données et leur partitionnement. Il permet aussi de décrire des types de tâches nommés *tasklet* qui disposent de plusieurs implémentations (p. ex. CPU, GPU). Le choix de l'implémentation la plus adaptée est ainsi déterminé à l'exécution par l'ordonnanceur. StarPU propose plusieurs stratégies d'ordonnancement (extensible par des experts) sélectionnées à l'exécution.

Ce type d'approche est particulièrement intéressant pour traiter des problèmes irréguliers. Il aide aussi à obtenir des performances portables d'une architecture matérielle à une autre en facilitant la transition entre les architectures sans demander un effort considérable aux concepteurs des applications à chaque fois qu'une nouvelle architecture doit être supportée.

Dans la plupart des supports exécutifs dédiés au calcul de haute performance, l'ordre de soumission des tâches est important car il détermine la structure du graphe de tâches résultant : l'ordre de soumission séquentiel des tâches est supposé être un ordre d'exécution valide. On parle dans ce cas de *cohérence séquentielle* [101]. Par exemple, la soumission d'une tâche A accédant en écriture à une variable D suivie de la création d'une tâche B accédant en lecture à la variable D engendre une dépendance de contrôle entre les tâches A et B : l'exécution de B ne peut se faire qu'après l'exécution de A. Si la tâche B est créée avant A, le sens de la dépendance est inversé : l'exécution de A ne peut se faire qu'après l'exécution de B.

La majorité des supports exécutifs disposent d'une interface de programmation (c.-à-d. API) sous la forme de bibliothèque. Cependant, ils peuvent être aussi utilisés à travers le langage de directive *OpenMP* [94], séparant ainsi la description du parallélisme de l'application du choix du support exécutif le plus adapté. D'autres encore sont utilisables directement à travers un langage dédié, comme c'est le cas dans *Legion* [15] (basé sur le support exécutif *Regent* [105]) dans le but de simplifier la mise en place de tâches dans les applications ou de rendre cette opération transparente.

OpenMP

OpenMP [94] est une interface de programmation visant à exprimer simplement le parallélisme des applications à l'aide de directives (annotations) explicites ajoutées à un code source séquentiel (C, C++ ou FORTRAN). OpenMP cible les systèmes à mémoire partagée (c.-à-d. utilisable sur des architectures à mémoires partagées, ou sur des architectures à mémoires distribuées en utilisant une mémoire partagée distribuée). Les directives proposées par OpenMP permettent des constructions de type fork-join, la création de tâches, la spécification de points de synchronisa-

tion, de constructions dédiées aux accélérateurs de calcul et elles incluent même des constructions de vectorisation de code ou encore des mécanismes de contrôle d'accès aux données (p. ex. verrous d'exclusion mutuelle). Il est important de noter que les directives spécifiées ne sont pas vérifiées, donc la conformité et la validité des annotations est à la charge des développeurs.

Les directives relatives aux tâches permettent de spécifier comment créer un graphe de tâches. En particulier, la directive *task* permet d'encapsuler une portion de code dans des tâches. À chaque fois que la directive est rencontrée, une tâche est créée et peut être exécutée plus tard lors d'un point de synchronisation ainsi que sur un autre fil d'exécution (en fonction des paramètres fournis à la directive). Pour créer un groupe de tâches, il est donc nécessaire de combiner les structures de contrôles séquentielles des langages sous-jacents avec des directives de création de tâches. La directive *task* permet aussi de préciser des dépendances entre des tâches à l'aide de dépendances de contrôle ou de dépendances de données disposant de modes d'accès de type *in*, *out* ou *inout*. Il est important de noter que, dans OpenMP, les dépendances de type *out* agissent comme des dépendances de type *inout* : lorsqu'une tâche A accédant à une donnée D en mode *out* est soumise avant une tâche B effectuant la même chose, OpenMP considère que la tâche B doit s'exécuter après la tâche A. De plus, comme de nombreuses API de supports exécutifs, la construction d'un graphe de tâches en OpenMP respecte la cohérence séquentielle.

Les directives OpenMP sont remplacées par des appels à un support exécutif lors de la compilation du code source. Plusieurs supports exécutifs [55, 65, 29, 9, 73] sont compatibles avec le modèle OpenMP.

Certains modèles de tâches comme OmpSs [31] visent à étendre OpenMP en y intégrant de nouvelles fonctionnalités offertes par un support exécutif sous-jacent (c.-à-d. Nanos, p. ex. support accru des architectures hétérogènes et distribuée).

Modèle de flux de données

Les modèles de *flux de données* (c.-à-d. *dataflow*) consistent à effectuer des enchaînements d'opérations qui se déroulent en parallèle (p. ex. pipeline). Dans ces modèles, un ensemble de tâches interdépendantes s'échangent des flux de données à travers des canaux de transfert. Le tout forme un graphe orienté dans lequel les nœuds sont des tâches et les arcs sont des *canaux de transfert*. Les tâches prennent en entrée des flux de données au travers de *canaux entrants* et produisent des données en sortie au travers des *canaux sortants*. Ces canaux sont responsables du stockage temporaire des données partagées entre des tâches. Ce stockage est nécessaire dans le cas où les tâches peuvent s'exécuter en parallèle, car elles peuvent ne pas être synchrones. Les tâches peuvent traiter simultanément un sous-ensemble des données qu'elles reçoivent par le biais de leurs canaux d'entrées et produire ensuite simultanément un ensemble de données dans leurs canaux de sorties. Ces modèles se distinguent des modèles de graphes de tâches principalement sur deux points : des tâches interdépendantes peuvent s'exécuter simultanément et s'échanger un flux de données au cours du temps, et le graphe de tâches d'une application peut contenir des cycles.

Dans ces modèles, les canaux font généralement transiter des données de manière monotone : c'est-à-dire que la relation d'ordre des données produites par une tâche est conservée lorsqu'elle est transmise à une autre tâche, formant ainsi un flux séquentiel de données. C'est par exemple le cas des *réseaux de Kahn* [68] dans lesquels les canaux sont associés à des files de données non bornées. Dans ce type de réseaux, l'écriture dans les canaux est non bloquante et la lecture bloquante. Lorsque les tâches consomment un nombre fixe (constant à l'exécution) de données dans chaque flux on parle de modèle de *flux de données synchrones* [32] (SDF). Ce type de modèle a la propriété de permettre un ordonnancement efficace des tâches calculable avant leur exécution.

Modèles pour la programmation d'accélérateurs de calcul

Sur les architectures disposant d'accélérateurs tels que des GPU, la manière de programmer diffère souvent du fait de la forte *hétérogénéité* de ces architectures : les accélérateurs peuvent disposer de fonctionnalités très différentes et remplir des objectifs très différents des unités de calcul centrales. Sur de telles architectures, l'un des paradigmes de programmation les plus couramment utilisés consiste à séparer les opérations exécutées sur les différents types d'unités de calcul et de préciser explicitement les *transferts de données* à effectuer lorsque les unités de calcul ne partagent pas un même espace de mémoire partagée. Le modèle *CUDA* [103] par exemple est un modèle permettant de programmer efficacement les GPU NVidia. Certains modèles tels que *OpenCL* [108] offrent un niveau d'abstraction supplémentaire permettant de supporter d'autres classes de GPU, voire d'autres formes d'accélérateurs tels que les FPGA. Des modèles tels qu'*OpenACC* [118] (ou plus récemment OpenMP) offrent un niveau d'abstraction encore plus élevé et permettent de décrire le parallélisme d'un code source à l'aide de directives adaptées aux accélérateurs de calcul (description des transferts de données, des tuiles de calcul, des groupes de fils d'exécution, etc.).

Langages spécifiques à un domaine

Les *langages de programmation spécifiques à un domaine* [107, 98] (c.-à-d. DSL) sont des langages de programmation permettant de résoudre des problèmes d'un domaine particulier tels que par exemple la résolution d'équations aux dérivés partielles [40]. Ils permettent à un spécialiste d'un domaine (p. ex. physicien) d'exprimer un problème simplement et de manière concise afin de générer un programme visant à résoudre ce problème. La spécificité de ces langages permet de mettre en place des méthodes de résolution spécifiques au domaine afin de produire des programmes efficaces. En effet, les méthodes mises en place durant le processus de génération peuvent être facilement adaptées pour prendre en compte les informations relatives au domaine considéré fourni par l'utilisateur du langage.

Cependant, cette approche se fait souvent au détriment de la capacité du langage à être généraliste ou flexible : l'intégration de nouvelles méthodes nécessite d'adapter le processus de génération de programmes qui peut se révéler complexe

car il requiert une expertise dans de nombreux domaines (domaine d'application du langage, méthode de parallélisation, architecture matérielle, etc.).

Squelettes algorithmiques

Les *squelettes algorithmiques* issus de la programmation fonctionnelle sont initialement définis comme des fonctions d'ordre supérieur spécialisées décrivant la structure d'un algorithme de la manière la plus générale possible. Ils offrent notamment une abstraction des *patrons de conception* récurrents de calculs, de communications ou d'interactions parallèles. Les approches basées sur les squelettes algorithmiques permettent de concevoir une application parallèle de manière abstraite en composant des squelettes algorithmiques. La composition se fait en imbriquant successivement des appels de fonctions (pouvant prendre respectivement des fonctions ou des structures de données en entrée). La capacité d'expression d'une application dépend des squelettes mis à disposition et leur capacité à être composés. Malheureusement, les squelettes algorithmiques sont rarement utilisés par les concepteurs des applications HPC, en grande partie parce que ce paradigme de programmation diverge grandement de la programmation impérative omniprésente dans le domaine.

Il existe de nombreuses approches existantes telles que *Muesli* [35], *SkeTo* [112], *SkePU2* [47], *FastFlow* [2] qui proposent entre autres des squelettes algorithmiques parallèles permettant d'appliquer une fonction sur un ensemble d'éléments (c.-à-d. *map*), d'effectuer une séquence d'opérations sur un flux d'éléments (c.-à-d. *pipe*), d'effectuer une opération de réduction sur un ensemble d'éléments (c.-à-d. *reduce*). Une étude des approches de squelettes algorithmiques existantes a été réalisée dans [56].

Approches basées sur les modèles d'acteurs

Certaines approches sont basées sur les modèles d'*acteurs*, où des acteurs (p. ex. processus, objets) communiquent ensemble par le biais de messages point à point. En fonction des messages reçus localement et d'un état interne, chaque entité peut effectuer des calculs, adapter son état, envoyer des messages ou créer d'autres acteurs. Ces acteurs sont regroupés dans des fils d'exécution et peuvent migrer d'un fil d'exécution à un autre.

Tout comme les modèles de graphes de tâches, cette approche a l'avantage de permettre de répartir le travail à réaliser sur les ressources disponibles en surdécomposant un problème en de nombreux acteurs qui opèrent simultanément. Cependant, contrairement aux modèles de graphes de tâches, les opérations à effectuer ne sont connues qu'*a posteriori* (c.-à-d. à partir du moment où elles sont exécutables) au lieu d'être connu avant (p. ex. par des dépendances). La répartition efficace des opérations à effectuer sur les ressources disponibles constitue un défi de taille lorsque les opérations parallèles sont fortement dépendantes.

Parmi les modèles de programmation existants mettant en œuvre cette approche, on peut citer notamment *Charm++* [69] dédié au calcul haute performance ou encore plus généralement les *objets actifs* [14].

2.3.4 Conclusion

Dans cette section, nous avons vu qu'il existe une grande variété de modèles de programmation existants disposant chacun de caractéristiques singulières. Les catégories évoquées dans cette section ne sont pas disjointes : il est courant que les modèles de programmation mettent en œuvre une combinaison de ces approches.

Bien que l'élaboration de modèles de programmation permettant d'exprimer simplement et efficacement le parallélisme des applications HPC constitue un défi de taille, ce n'est pas le seul critère important à considérer lors de la conception d'une application. En effet, la structuration du code source des applications et l'interaction entre les différentes parties d'une même application jouent aussi un rôle crucial dans leur conception.

2.4 Programmation par composition

La complexité des applications s'est grandement accrue depuis les débuts de l'informatique avec la croissance de la complexité des opérations réalisées par les systèmes d'information résultant d'un traitement plus précis, plus complet et capable de supporter des volumes d'information toujours plus grands. Dès les années 1960, la complexité des applications atteint un tel niveau que la qualité des logiciels se dégrade progressivement, laissant la place à des logiciels plus coûteux, peu fiables, délaissant la sécurité, parfois inefficaces et surtout difficiles à maintenir. On parle alors pour la toute première fois de *crise logicielle* [90, 54]. Afin de régler cette situation, des méthodes de conception logicielle systématique sont mises en place [25]. On assiste alors à la naissance du génie logiciel.

Bien que le génie logiciel s'intéresse à de nombreux aspects gravitant autour de la réalisation d'une application tels que les méthodes de tests logiciels ou les pratiques de pilotage de projet, nous nous intéressons, dans ce manuscrit, à la branche du génie logiciel qui se préoccupe de la *conception logicielle*. Plus particulièrement, nous nous intéressons à la structure des codes sources applicatifs dans le but d'améliorer le développement et la maintenance des applications au fil du temps.

Dans cette section, nous évoquons tout d'abord les méthodes et métriques proposées historiquement afin de structurer les codes sources des applications et qui sont toujours très utilisées actuellement par un grand nombre d'applications. Puis, nous présentons les modèles de composants et leur fonctionnement.Ensuite, nous détaillons les concepts communs à de nombreux modèles de composants existants. Enfin, nous présentons les différentes formes de compositions offertes par les approches existantes.

2.4.1 De la programmation structurée aux modèles de composants

Il existe de multiples paradigmes et méthodes de programmation visant à structurer les codes sources des applications. Dans cette section, nous nous intéressons en

particulier aux paradigmes et méthodes les plus fréquemment utilisés par les applications, c'est-à-dire la programmation structurée et la programmation orientée objet ainsi que la programmation par composition qui fait l'objet de la section suivante.

Programmation structurée

C'est dans les années 1950, alors que la *programmation impérative* était massivement utilisée, qu'est née la *programmation structurée* [39], largement répandue ensuite vingt ans plus tard. Ce paradigme consiste à utiliser des structures de contrôle telles que des *boucles*, des *conditions* ou même des *sous-routines*. Il contraste avec l'usage des instructions *goto* utilisées plus tôt et engendrant un code source peu clair et déstructuré. L'utilisation de structures de contrôle a permis d'améliorer la lisibilité des codes sources dès lors beaucoup plus faciles à maintenir.

Le nombre de routines et sous-routines contenues dans les applications a connu une croissance fulgurante engendrant de nouveaux problèmes : comment décomposer et organiser les routines d'une application efficacement. En effet, les routines d'une application peuvent être plus ou moins interdépendantes entre elles et réaliser chacune des hypothèses parfois implicites. Lorsque le nombre de routines devient imposant, il est difficile de connaître l'impact de la modification d'une routine sur le reste de l'application ou encore de réutiliser une routine seule dans une autre application sans la modifier. La programmation orientée objet permet de regrouper ensemble des routines qui opèrent sur une même structure de données. Cette nouvelle approche, émergeant dans les années 1960 [38], connaît une effervescence dans les années 1980. Elle a rapidement supplanté la programmation structurée et est aujourd'hui très largement utilisée par les applications.

Programmation orientée objet

La *programmation orientée objet* [102] repose sur le concept d'objet. Un objet est un agrégat de données et de routines appelées respectivement attributs et méthodes. Les méthodes issues d'un même objet peuvent accéder ou modifier les données de leur objet. Malgré la diversité des langages orientés objet, dans de nombreux langages, les objets ont un type appelé classe. On appelle instance d'une classe C un objet dont le type est C.

Dans ce paradigme de programmation, une application est un groupe d'objets qui interagissent. Pour assurer une bonne interaction, il est possible de contrôler la visibilité des attributs et des méthodes d'un objet par d'autres (p. ex. interdire ou autoriser l'accès conditionnellement), on parle d'encapsulation. Ce mécanisme permet de contrôler les interactions entre les objets. Il a l'avantage d'améliorer la maintenance des applications en déléguant la cohérence des attributs d'un objet aux méthodes qui le composent, mais aussi en cachant les attributs pouvant être perçus comme détails d'implémentation.

Dans la plupart des langages orientés objet, deux formes de relations existent entre les objets : la composition et l'héritage. La composition d'objets consiste à inclure structurellement un objet en tant qu'attribut d'un autre objet. L'héritage

consiste à définir la base de l'implémentation d'un objet A à partir d'un autre objet B. L'objet A peut alors étendre l'implémentation B et/ou modifier son comportement. Le principe de substitution établi par Liskov [80] précise qu'un objet de type T peut être remplacé par un objet d'un sous-type S de T si toutes les propriétés démontrables d'un objet de type T sont aussi vraies pour un objet de type S.

La forme de composition offerte par la programmation orientée objet n'est pas sans défaut. L'inclusion d'un objet dans un autre en tant qu'attribut ou l'utilisation d'objets dans des routines est masquée au niveau de l'architecture de l'application : les dépendances entre les objets (ou classes) ne peuvent être déterminées que par une analyse souvent profonde et fastidieuse du code source définissant les objets. Or, sur des codes sources de grande taille, cette analyse est coûteuse et la modification du code source d'un objet, sans analyse des dépendances préalable, peut avoir des répercussions dramatiques (p. ex. bogues) sur le reste du code source de l'application. Les approches à base de composants logiciels offrent des méthodes pour résoudre ce type de problème.

Programmation modulaire

La *programmation modulaire* [111], émergeant dans les années 1970 [119], est une méthode de programmation visant à regrouper des ensembles de briques logicielles (routines, objets, etc.) dans des modules de haut niveau. Cette méthode de programmation est indépendante du paradigme de programmation utilisé (p. ex. programmation structurée ou programmation orientée objet). La programmation modulaire s'intéresse donc à la manière de structurer l'architecture d'une application (c.-à-d. organisation du code source) plutôt qu'à la façon de formuler des solutions dans un langage de programmation (c.-à-d. contenu du code source). Dans cette approche, chaque module détient une interface. Dans le cas de la programmation structurée, cette interface est un ensemble de prototypes de routines visibles par d'autres modules et un ensemble de prototypes de routines nécessaires au bon fonctionnement du module défini. Cette méthode a permis de faciliter grandement la conception et la maintenance des applications en permettant de maintenir des modules séparément et de les réutiliser dans plusieurs applications. Cette approche constitue les fondations de la *programmation par composition*.

Avant de présenter les approches par composition plus en détail, la section suivante introduit des notions importantes en génie logiciel et nécessaires pour bien comprendre les bénéfices des approches par composition.

Cohésion, séparation des préoccupations et couplage de codes

En programmation modulaire, la cohésion, la séparation des préoccupations et le couplage de codes jouent un rôle primordial dans la conception d'application [121].

La *cohésion* est une métrique mesurant la capacité d'un élément applicatif à remplir une unique fonction logique (ensemble cohérent de fonctionnalités). On parle de cohésion forte lorsqu'un module ou une routine se concentre sur une seule fonctionnalité et à l'inverse, de cohésion faible lorsque le module fournit des fonctionnalités

variées sans lien logique fort. Une faible cohésion peut engendrer de sévères problèmes de maintenance d'une application. En effet, la multiplication des fonctions au sein d'une même entité a un impact négatif sur la lisibilité de l'ensemble. De plus, l'implémentation de fonctionnalités peut nécessiter une expertise dans un domaine spécifique ce qui fait que la maintenance de l'ensemble requiert, dans ce cas, un haut niveau de compétences dans de nombreux domaines.

Il est possible d'accroître la cohésion de modules applicatifs en *séparant les préoccupations* issues d'une même application : chaque préoccupation (p. ex. intégration de fonctionnalités) de l'application est la raison d'être de modules dédiés. Cette séparation peut se faire verticalement à travers une hiérarchie de couches d'abstraction logicielles (modèles, bibliothèques, frameworks, etc.). Elle peut aussi se faire horizontalement à travers un groupe de modules divisant l'ensemble des fonctionnalités offertes par un logiciel. Lorsque plusieurs personnes sont impliquées dans le processus de conception d'une même application, séparer les préoccupations d'une application est essentiel. En effet, cela permet à chacun de se concentrer sur un sous-ensemble restreint de fonctionnalités isolées des autres évitant ainsi des problèmes de travaux concurrents sur une même portion de code. La séparation des préoccupations est fortement liée à la réutilisation d'un code source. En effet, la séparation des fonctionnalités dans des modules distincts permet de les réutiliser plus facilement dans d'autres applications à condition qu'il soit possible d'extraire facilement les dépendances entre les modules.

Le *couplage* est une métrique mesurant l'interdépendance entre les éléments applicatifs. Un couplage fort entre deux modules signifie que les modules sont fortement liés ensemble. C'est, par exemple, le cas lorsqu'un module dépend du contenu d'un autre (p. ex. implémentation). Un tel couplage présente un risque conséquent, car il peut complexifier la maintenance d'une application, étant donné que les modules ne peuvent être maintenus séparément. *A contrario*, un couplage faible signifie que les modules sont très peu dépendants les uns des autres. Un couplage fort ou une faible cohésion risque de rendre plus difficile la maintenance d'une application ; la modification d'un module est plus encline à impacter les modules interdépendants.

2.4.2 Définitions

La programmation par composition est une méthode de programmation utilisant la composition comme un moyen d'améliorer la séparation des préoccupations et la réutilisation des logiciels. Cette approche vise à assembler des briques logicielles appelées composants afin de former un système logiciel complet (p. ex. applications, bibliothèques, etc.). L'ensemble est appelé assemblage. Cette section commence par définir ce qu'est un composant, puis aborde la manière dont les composants peuvent être assemblés.

Composant

Le concept de *composant logiciel* a été proposé initialement en 1968 par Douglas McIlroy [83]. Il admet aujourd'hui plusieurs définitions, plus ou moins restrictives

ou compatibles entre elles, parmi lesquelles celle de Clemens Szyperski [111] : "*a software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition*". D'autres définitions apportent quelques précisions. La définition donnée par D'Souza et Wills [45], par exemple, définit que la composition de composants ne doit pas nécessiter de modification des composants (excepté la manipulation de leurs attributs faisant partie de leurs interfaces). Celle fournie par la dernière version de l'*Unified Modeling Language* [110] (UML 2.4) de l'*Object Management Group* (OMG) définit qu'un composant n'est pas forcément exécutable : ces composants sont présents lors du processus de développement et peuvent contenir une portion de code source d'un système logiciel (p. ex. *packages*). Cette dernière définition souligne l'importance de l'expression explicite des dépendances des composants, que ce soit lors du développement ou à l'exécution.

Dans ce manuscrit, nous considérons la définition donnée par Szyperski : un composant est une boîte noire dotée d'un ensemble d'*interfaces* bien défini décrivant l'ensemble de ses dépendances et qui visent à être composées par des tierces parties. Le fait qu'un composant soit une boîte noire signifie que le composant n'est défini que par ses interfaces et donc peut être déployé indépendamment des autres. Ainsi, aucune hypothèse, autre que celles faites par les interfaces, ne peut être considérée (p. ex. spécificité de l'état ou du fonctionnement interne à un composant). Un composant peut détenir une ou plusieurs *mises en œuvre* conformes à ces interfaces.

Interface

D'après Heineman et Councill, une interface est "*an abstraction of the behavior of a component that consists of a subset of the interactions of that component together with a set of constraints describing when they may occur. The interface describes the behavior of a component that is obtained by considering only the interactions of that interface and by hiding all other interactions*" [60].

Dans ce manuscrit, nous définissons une interface comme une abstraction décrivant un point d'interaction qui dispose de contraintes d'utilisation. La description d'une interface peut se faire indépendamment des autres interfaces d'un composant. Toutefois, cela ne signifie pas que les interfaces d'un même composant sont indépendantes. En effet, la description d'une interface peut supposer l'existence d'un état interne au composant partagé entre interfaces (et dont le comportement peut agir sur cet état). Les interfaces peuvent être aussi appelées *ports* et sont généralement typées. Une liste non exhaustive de formes d'interfaces existantes dans les approches par composition est présentée en section 2.4.4. Un exemple d'interfaces *use* et *provide* est donné à gauche de la figure 2.7. Les interfaces des composants sont destinées à être composées.

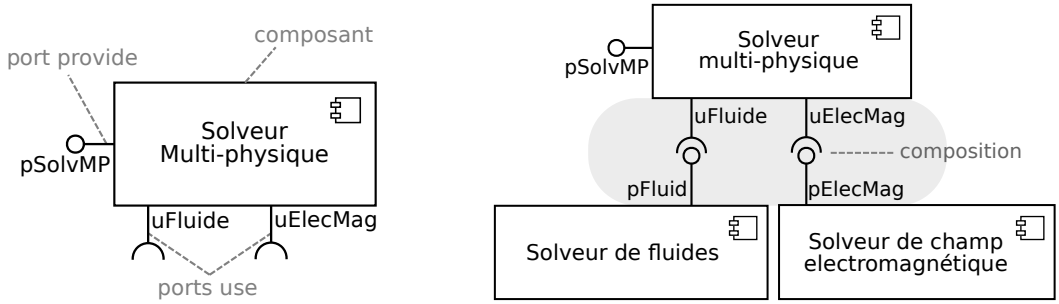

FIGURE 2.7 – Exemple d'un composant avec un port *use* et un port *provide* (gauche) et d'assemblage de deux instances de composants de types différents (droite).

Composition : assemblage, connecteur et connexion

La *composition* est le mécanisme mettant en relation les interfaces des composants. On appelle *assemblage* le résultat de cette composition. La mise en relation des interfaces peut se faire à l'aide de connecteurs.

Un *connecteur* est une brique architecturale permettant de modéliser les interactions entre les composants et les règles qui gouvernent ces interactions. Il se place au même niveau que les composants et joue le rôle de médiateur. Les connecteurs partagent des similarités avec les composants : ils sont dotés d'un ensemble d'interfaces nommées *rôles*. Un connecteur peut imposer des contraintes sur les interfaces qu'il met en relation (p. ex. types des interfaces supportées par chaque rôle, relation d'ordre entre les rôles). Les approches par composition offrent souvent un ensemble de connecteurs prédéfinis. Certaines approches fournissent néanmoins des mécanismes permettant de définir de nouveaux connecteurs (p. ex. par la composition de composants et de connecteurs prédéfinis) manipulables comme des *entités de première classe* [104]. On appelle *connexion* une instance de connecteur.

Il existe un très grand nombre de formes de connecteurs parmi les approches par composition existantes (p. ex. composition à base d'appels de méthodes, de partage de données, de flux de données, d'événements, etc.). Une liste non exhaustive de formes de connecteurs existants est présentée en section 2.4.4. Un exemple d'assemblage de composants disposant de connexions *use-provide* est donné à droite de la figure 2.7.

Modèle de composants

Un *modèle de composants* définit ce que sont les composants et par quels moyens les composer. Bien que l'expression d'une architecture à base de composants puisse se faire à travers des bibliothèques ou des frameworks, de nombreuses approches proposent des langages dédiés. Un langage qui permet de définir comment assembler les composants est appelé un *langage d'assemblage*.

Parmi les modèles existants, certains séparent la phase de conception des composants de la phase d'assemblage durant laquelle ils sont composés. Ce manuscrit s'intéresse tout particulièrement à ces modèles, car ils permettent de rendre visible

l'architecture des applications. Ces modèles distinguent généralement le type d'un composant et d'une instance de composant. Le type d'un composant est défini par l'ensemble de ses interfaces. La description d'un type est indépendante de sa mise en œuvre. Un type est associé à une (ou plusieurs) mise en œuvre. Une instance est une entité présente lors d'une phase d'assemblage et est conforme à un type.

Propriétés

À partir des définitions d'un composant et d'une interface fournies précédemment, il est possible de déduire par construction les propriétés inhérentes des modèles de composants.

Tout d'abord, les composants sont des boîtes noires déployables indépendamment, ce qui signifie qu'il est possible de les développer et de les composer indépendamment des autres. Cette caractéristique favorise ainsi la séparation des préoccupations : chaque composant constitue une brique logicielle (séparation d'un logiciel en morceaux) dédiée à des fonctionnalités précises (préoccupations) qui peuvent être développées par des personnes différentes (dans la mesure où les composants ont pour objectif de posséder une forte cohésion).

De plus, la description explicite de toutes les dépendances assure de pouvoir composer les composants sans causer de dommages collatéraux : elle évite des effets de bord (p. ex. comportement anormal) lors de la composition qui proviennent de la présence de considérations implicites. Cette caractéristique permet de mesurer simplement le couplage des parties logicielles par la visibilité des dépendances entre les composants. Dès lors, il est plus facile d'adapter le couplage des parties logicielles de telle manière à ce qu'il ne nuit pas à la maintenabilité.

Enfin, le fait que la composition puisse être réalisée par des tiers implique que les dépendances entre les composants ne sont pas entièrement définies par les concepteurs de chaque composant : le concepteur d'un composant ne contrôle pas directement avec qui le composant est connecté, il précise uniquement des interfaces (dépendances partielles) ensuite composées par un tiers (dépendances entièrement déterminées). Cette caractéristique est une condition nécessaire à la réutilisation des parties logicielles.

2.4.3 Concepts

Il existe une grande diversité de modèles de composants couvrant une large gamme de concepts divers. Cette section s'intéresse aux concepts qui reviennent fréquemment parmi les modèles de composants. En particulier, elle présente les concepts de hiérarchie et généricité des unités de composition (c.-à-d. composants et les connecteurs).

Hiérarchie

La *hiérarchie* est un mécanisme permettant de décrire des composants dont la mise en œuvre est un assemblage de composants. De tels composants sont appelés

composites. Un composant qui n'est pas un composite est un composant primitif. Les assemblages de la mise en œuvre des composites peuvent contenir des instances de composants qui sont, à leur tour, des composites. Le tout forme une arborescence de composants imbriqués. Un modèle de composants avec cette caractéristique est dit *hiérarchique*. La hiérarchie joue un rôle déterminant dans la modularité des applications. En effet, elle permet de modéliser simultanément différents niveaux d'abstraction de l'architecture des applications. Il est alors possible de raisonner sur chaque niveau d'abstraction indépendamment des autres. Or, il est plus simple de concevoir et maintenir des portions de l'architecture d'une application que l'ensemble. Les modèles de composants Fractal [30], BIP [12], HLCM [18] et FlowVR [79] sont des exemples de modèles de composants hiérarchiques.

Généricité

La *généricité* des composants s'apparente à la généricité dans les langages comme C++ ou Java (c.-à-d. *templates*). Cette propriété peut être utilisée conjointement avec la hiérarchie. La généricité des modèles de composants permet de décrire des structures de composition dans lesquelles les traitements à effectuer ou les types de données utilisées sont des paramètres des unités de composition (inclus composants et connecteurs). Par exemple, dans le cas d'une architecture de type maître-esclave constituée d'une instance de composant maître et de plusieurs instances de composants esclaves, il est pertinent de définir le nombre d'instances à créer en tant que paramètre générique. Il peut être aussi intéressant de fournir le type du composant esclave en tant que paramètre générique afin de spécialiser automatiquement la mise en œuvre du composant générique. HLCM [18] [19] est un exemple de modèle supportant la généricité.

2.4.4 Formes de composition

Il existe de nombreux moyens d'assembler des composants, ou plus précisément leurs interfaces (ou ports). Dans cette section, nous nous intéressons aux formes de composition les plus fréquentes dans les modèles de composants existants ainsi que celles particulièrement utiles aux applications HPC. Cette section traite des méthodes de composition de codes indépendamment, sans considérer particulièrement le cas de la composition efficace de codes parallèles, rencontrée dans les codes HPC. Ce problème est analysé dans la section 2.5.

Use-provide

La composition *use-provide* est l'une des formes de composition les plus récurrentes dans les modèles de composants. Elle permet aux composants d'exposer des interfaces appelées port *provide* fournissant des services et des interfaces nommées ports *use* qui requièrent des services fournis par d'autres composants. Les ports *provide* exportent des fonctions alors que les ports *use* importent ces fonctions. Les ports *use* et *provide* peuvent être typés. Le type d'un port est généralement une

interface objet contenant un ensemble de méthodes auxquelles les modèles ajoutent parfois des informations supplémentaires telles que l'impact des méthodes sur le comportement du composant ou encore des contraintes d'utilisation des méthodes.

Les ports *use* et *provide* peuvent être connectés ensemble par une connexion de type *use-provide*. Lorsqu'un port *use* est connecté à un port *provide*, le port *use* peut appeler des méthodes du port *provide* (dans la mesure où la méthode est conforme au type des ports). Certains modèles proposent des connecteurs fournissant une abstraction d'un ensemble de connexions *use-provide* (voire d'un assemblage partiel) permettant d'exprimer notamment les cas où plusieurs ports *use* sont connectés à un même port *provide*. Ce type de connecteur est appelé *use-multiple*.

Ce type de composition est utilisé dans de très nombreux modèles de composants tels que *Fractal* [30], *CCM* [106, 117], *GridCCM* [96], *GCM* [13], Koala [93], ou encore parmi les approches dédiées au HPC telles que *CCA* [4] et L^2C [20].

Flux de données

Les approches à base de *flux de données* ont été présentées dans la section 2.3.3. Dans les modèles de composants utilisant cette approche, des interfaces de sortie produisent un flux de données et des interfaces d'entrée consomment un flux de données. Les ports d'entrée et de sortie peuvent être typés. Le type des ports inclut généralement un type de structure de données (p. ex. primitive ou composée). Il peut aussi inclure des informations supplémentaires relatives au cycle de vie des données, des contraintes sur l'utilisation des données, etc.

Les ports d'entrée peuvent être connectés aux ports de sortie par le biais de connexions d'entrée-sortie transférant ainsi les données d'un port à un autre. Certains modèles proposent des connecteurs plus complexes permettant de filtrer les données, de les répliquer vers plusieurs ports de sortie, de les stocker temporairement ou encore de les redistribuer.

FlowVR [79], *FastFlow* [2] et *Decaf* [44] sont des exemples de modèles de composants supportant cette approche. On peut noter que dans le cas de FastFlow, l'architecture de l'application n'est pas explicitement séparée des détails d'implémentation (c.-à-d. architecture visible dans les sources C++ de l'application).

Flux de travaux

Les modèles de composants basés sur les *flux de travaux* incluent les approches basées sur les flux de données et les étendent afin d'y intégrer des connecteurs des structures de contrôle telles que des conditions ou des boucles. Ces modèles permettent généralement aux composants d'exposer des interfaces d'entrée ou de sortie. Certains modèles dotent les composants d'interfaces de contrôle permettant de déclencher des traitements sur un autre composant sans transmettre de données.

Les interfaces de contrôle/données peuvent être connectées afin de définir un flot de contrôle entre deux composants (confondu avec le flot de données en cas d'interfaces de données). Ces modèles se distinguent de ceux basés sur les flux de données par la présence des connecteurs permettant de configurer le comportement

du flot de contrôle. Par exemple, certains modèles offrent un moyen d'exprimer des commutateurs dont le but est de sélectionner l'activation d'une interface de contrôle parmi plusieurs en fonction de données reçues depuis un port.

Parmi les approches existantes supportant ce type de composition, on peut citer comme exemple Gwendia [85] ainsi que le modèle de composants décrit dans [77]. STCM [27] est un autre modèle de composants supportant cette approche qui combine la notion de tâches et de composants dans une même unité de composition disposant d'interfaces *use*/*provide* et d'entrée/sortie, permettant d'effectuer des compositions selon deux dimensions différentes (composition *use-provide* et composition à base de flux de travaux décrivant le flux de contrôle d'une application le biais de structures de contrôle telles que des boucles).

Squelettes algorithmiques

La définition des modèles de composants présentée en section 2.4.2 est suffisamment large pour y inclure des approches basées sur les *squelettes algorithmiques* présentés en section 2.3.3. On peut donc considérer les fonctions de premier ordre comme des composants, les types de paramètres de leurs prototypes comme des interfaces et les squelettes comme des connecteurs (ou des composites) permettant de mettre en relation certaines des interfaces des composants (paramètres de sortie des fonctions). Les approches de squelettes algorithmiques se concentrant tout particulièrement sur la définition d'un ensemble de connecteurs (ou composites) spécifiques et prédéfinis afin de pouvoir exprimer une large gamme d'algorithmes.

Parmi les approches existantes de modèles de composants supportant la composition à base de squelettes algorithmiques, on peut citer notamment STKM [3] qui bénéficie de la composition *use-provide* et de la composition à base de flux de travaux.

L'expression de squelettes algorithmiques peut aussi se faire à travers la généricité des modèles de composants hiérarchiques. Par exemple, dans HLCM [18], il est possible de créer un composant maître-esclave générique dont l'implémentation est un ensemble d'instances de composants esclaves connectés à un même composant maître. Le choix des composants maître et esclave peut être déterminé par des paramètres génériques.

2.4.5 Conclusion

Dans cette section, nous avons commencé par évoquer l'importance du génie logiciel dans la conception d'applications. Nous avons ensuite présenté les approches utilisées jusqu'à maintenant afin de faire face à la complexité croissante des applications ainsi que des métriques et concepts permettant d'évaluer et d'agir sur la qualité de l'architecture des applications. Nous nous sommes concentré plus spécifiquement sur les composants logiciels, une approche favorisant la réutilisation de code et la séparation des préoccupations. Pour cela, nous avons fourni une définition des composants et des interfaces, puis nous avons décrit comment ils peuvent être assemblés afin de créer une application complète.

L'utilisation des modèles de composants est aujourd'hui largement démocratisée dans de nombreux domaines de l'informatique, des systèmes embarqués aux services web en passant par la conception de logiciels multimédias (musique, jeux vidéo, etc.). Cependant, l'adoption des approches à base de composants est bien moindre dans le cas des applications HPC. La faible adoption des modèles de composants dans ce domaine peut s'expliquer par la difficulté de combiner efficacement des codes parallèles complexes dans les approches existantes.

2.5 Programmation parallèle par composition

Cette section s'intéresse à l'usage de la programmation par composition dans la conception de codes parallèles pour le calcul de haute performance. En effet, dans la section 2.3, nous avons vu qu'il existe un ensemble varié de modèles de programmation permettant d'exprimer le parallélisme des applications HPC. Nous avons ensuite vu dans la section 2.4, qu'indépendamment du paradigme de programmation utilisé pour exprimer les traitements réalisés par une application, il est important de mettre en place des méthodes de conception afin de pouvoir mettre au point une application maintenable, en particulier lorsque plusieurs personnes sont impliquées dans le processus de conception. Dans cette dernière section, nous nous sommes plus particulièrement concentré sur les modèles de composants et leur capacité à combiner des codes indépendamment de l'expression du parallélisme. Tout naturellement, nous pouvons nous demander si la présence du parallélisme a un impact sur la composition. Plus précisément, est-il possible de composer des codes parallèles comme on compose des codes séquentiels et, dans le cas contraire, comment réaliser une telle opération ?

Cette section reprend les formes de composition étudiées dans la section 2.4.4, analyse leurs capacités à supporter des codes parallèles et présente plus généralement les approches existantes permettant de composer des codes HPC. Elle décrit ensuite le modèle de composants L^2C utilisé dans les chapitres 3 et 4. Puis la section conclut en laissant entrevoir les problématiques abordées dans ce manuscrit.

2.5.1 Forme de composition de codes parallèles

L'approche la plus simple pour composer des codes parallèles dans les modèles de composants consiste à considérer que la présence du parallélisme fait partie intégrante de la mise en œuvre des composants et ne doit donc pas être nécessairement visible au niveau des interfaces des composants. Cette méthode a l'avantage de permettre de remplacer facilement une mise en œuvre parallèle par une autre (disposant d'une nouvelle stratégie de parallélisation ou utilisant un paradigme de programmation différent) de manière transparente. Cependant, en cachant le parallélisme lors de la phase d'assemblage, la composition de codes parallèles indépendants est un véritable défi.

Sans information sur l'usage parallèle des interfaces (p. ex. *use-provide*), il est impossible de déterminer lors de l'assemblage si les interfaces de deux composants

sont compatibles (c.-à-d. les mises en œuvre des deux composants peuvent interagir par le biais de ces interfaces sans causer de problème). Pour simplifier la composition, des contraintes supplémentaires sur les interfaces peuvent être ajoutées (p. ex. paradigme de programmation parallèle utilisé et contraintes d'utilisation parallèle), soit par les concepteurs des composants, soit directement dans le modèle de programmation. Le premier risque est de rendre les interfaces dépendantes d'une mise en œuvre spécifique d'un composant, de réduire la compatibilité des interfaces (p. ex. paradigme de programmation ou approche de parallélisation non compatible) ou encore d'impacter les performances en introduisant des coûts non nécessaires (p. ex. redistribution, synchronisations, etc.). Le second a le défaut de réduire potentiellement la classe des paradigmes de programmation supportés par le modèle de composants et laisse au modèle le choix de définir les interfaces.

Une dernière approche consiste à fournir une description du parallélisme au niveau de l'assemblage des composants. Le parallélisme est soit exprimé à l'aide de la structure de contrôle explicite, soit implicitement déduit de la structure de l'assemblage (p. ex. modèle de flux de données). Cette approche peut être utilisée conjointement avec la précédente.

Cette section analyse les différentes formes d'interfaces existantes dans les modèles de composants qui permettent une composition de codes parallèles.

Use-provide parallèle

Un des moyens d'effectuer une composition *use-provide* dans un contexte parallèle consiste à appliquer une composition *use-provide* d'un contexte séquentiel sur chaque élément responsable du parallélisme (p. ex. fils d'exécution, tâches, etc.) indépendamment des autres éléments. Cependant, bien que cette variante parallèle soit aussi expressive que son homologue séquentiel, elle limite fortement la composition de codes parallèles. En effet, il est fréquent en HPC de composer des codes n'opérant pas avec le même niveau de parallélisme (c.-à-d. problème $M \times N$, p. ex. nombre de fils d'exécution différent, composition de deux groupes de tâches de taille différente, etc.), ou encore que les éléments responsables du parallélisme des codes composés ne soient pas indépendants (p. ex. besoin de redistribution des données avant l'utilisation d'un service, synchronisation partielle ou totale nécessaire, etc.).

Cette approche est utilisée notamment par CCA [4], un modèle de composants dédié au calcul de haute performance. Dans CCA, les instances de composants et leurs ports sont dynamiquement créés à l'exécution dans un contexte pouvant être séquentiel ou parallèle. Dans le cas d'un code parallèle, une exécution typique d'une application CCA consiste à répliquer un assemblage dans plusieurs processus. L'ensemble des instances d'un même composant répliqué sur tous les processus est appelé *cohorte*. Tous les processus d'une cohorte exécutent le même code issu de leur composant et peuvent communiquer entre eux à l'aide de bibliothèques de passage de messages quelconques, telles que MPI. CCA fournit uniquement un moyen de connecter des instances de composants à travers des connexions *use-provide* réalisées localement au sein d'un processus. Lorsque deux instances de composants opèrent sur des données distribuées de manière différente entre les processus, le concepteur

des composants doit lui-même effectuer une redistribution, car CCA n'offre aucun mécanisme pour effectuer une telle opération. Cela est typiquement réalisé en divisant l'ensemble des processus en deux parties qui contiennent des assemblages différents, mais qui partagent un même ensemble d'instances de composants visant à redistribuer des données. Certaines mises en œuvre de CCA étendent le modèle en définissant comment effectuer une composition de composants dont les cohortes ne comportent pas le même nombre de processus. C'est le cas de *SCIRun2* précisant que les cohortes issues de deux composants effectuent des échanges de type *round-robin*. Cependant, la redistribution des données dans SCIRun2 reste à la charge des concepteurs des applications.

GridCCM [96] étend le modèle de composants CCM dans le but de spécifier comment effectuer une composition de codes parallèles basée aussi sur le concept de *use-provide*. Dans GridCCM, les composants peuvent disposer d'une implémentation pouvant être séquentielle ou parallèle. Le modèle permet d'assembler ces composants en étendant les interfaces *use-provide* avec une description parallèle (utilisée uniquement par les composants parallèles) qui spécifie pour chaque méthode d'une interface, le type de distribution de chacun des paramètres (p. ex. distribution en blocs réguliers pour un type de données tel qu'une matrice). Le changement de distribution de données est effectué par un code utilisateur injecté lors d'une phase de compilation de l'assemblage en un assemblage CCM. Cette approche a l'avantage de permettre une composition transparente des codes parallèles, notamment ceux qui ont recours à un nombre de fils d'exécution différent. De plus, elle évite d'engendrer des sursynchronisations à l'exécution.

GCM [13], une extension de Fractal, propose des interfaces dérivées du *use-provide* séquentiel : une interface de diffusion d'appels de méthodes et une interface de rassemblement d'appels de méthodes. Dans cette approche, une mise en œuvre parallèle d'un composite peut disposer d'un ensemble d'instances de composants séquentiels. Un composite peut disposer d'interfaces de diffusion qui ont pour effet de transformer un unique appel de méthode en un ensemble d'appels de méthodes à appliquer sur les ports des instances du composite après avoir traversé la membrane du composite. Tout comme GridCCM, GCM redistribue automatiquement les données de chaque paramètre de la méthode appelée indépendamment. Un composite peut aussi disposer d'interfaces de rassemblement qui ont pour effet de transformer un ensemble d'appels de méthodes effectués en parallèle par les instances du composite en un unique appel de méthodes après avoir traversé la membrane du composite. La composition de deux composites dont la mise en œuvre est parallèle (c.-à-d. contenant des instances séquentielles) peut être réalisée en fusionnant les opérations de rassemblement et de diffusion en une unique étape qui consiste à connecter directement les instances des deux composites en fonction des distributions des paramètres des interfaces de rassemblement et de diffusion. Tout comme pour GridCCM, cette dernière opération permet d'éviter des sursynchronisations lorsque le niveau de parallélisme n'est pas le même entre deux composants qui sont composés ensemble.

Flux de données et flux de travaux

FlowVR [5] est un exemple de modèle de composants (dédié à la visualisation scientifique permettant d'effectuer des traitements *in situ*) à base d'interfaces de flux de données qui offrent des mécanismes pour composer des codes parallèles. Dans cette approche, les composants peuvent communiquer par le biais de ports d'entrée ou de sortie en utilisant trois opérations (de type *push*) : *put* (envoi d'un message sur un port de sortie), *get* (récupération du message le plus ancien sur un port d'entrée) et *wait* (attente de données sur un port d'entrée). Le modèle permet de connecter des ports de données entre eux et de filtrer les messages qui transitent sur la connexion résultante. Chaque connexion est associée à une file qui stocke les messages envoyés par les instances de composants. Les composants sont manuellement placés sur des unités de calcul. Le transfert des messages entre des ressources est réalisé de manière transparente. Deux mises en œuvre parallèles de composants utilisant différents niveaux de parallélisme peuvent donc être simplement composées ensemble. Il en résulte deux codes parallèles qui communiquent par échange de flux de messages. Il est important de noter que le type des messages transmis, la taille des messages ainsi que leur ordre d'envoi doivent être judicieusement choisis pour pouvoir composer efficacement les codes parallèles. La redistribution de structures de données complexes entre deux codes parallèles est laissée à la charge des concepteurs d'une application.

Decaf [44] est un modèle très similaire à FlowVR qui se distingue par une conception ciblant particulièrement les supercalculateurs. Il inclut notamment Bredala [43], un modèle de description et de transmission des données, permettant d'effectuer des redistributions de données entre des traitements indépendants facilitant ainsi la composition des codes HPC.

Partage de données

Les modèles de composants supportant le *partage de données* permettent d'exprimer des interfaces qui fournissent des données partagées et des interfaces qui accèdent à ces données. La connexion de ces deux types d'interfaces permet d'exprimer la dépendance d'un composant à une donnée fournie par un autre. Le partage de données met en évidence les accès aux données entre les composants, ce qui se révèle particulièrement intéressant lorsque les composants sont associés à un modèle de ressources (c.-à-d. placement de composants sur des ressources matérielles ou logicielles). Cette forme de composition se distingue de celle des approches de flux de données où plusieurs données transitent d'une source vers une destination. Au lieu de cela, une seule donnée est accessible par plusieurs composants, à un instant donné, ce qui a pour effet de soulever des problèmes d'accès concurrents à une donnée partagée par plusieurs composants. Ainsi, les interfaces des modèles de composants possèdent des mécanismes permettant de contrôler l'accès aux données entre les composants (p. ex. exclusion mutuelle).

Cette méthode de composition a notamment été proposée comme extension de CCM et CCA [8, 28].

Squelettes algorithmiques

Comme nous avons pu voir dans les sections 2.3 et 2.4, les squelettes algorithmiques peuvent être utilisés comme paradigme de programmation parallèle permettant d'exprimer des traitements parallèles, mais aussi comme méthode de composition de codes en utilisant une stratégie de conception à base de patrons, chacun faisant souvent l'objet d'approches disjointes. Dans le cas de la composition de codes parallèles, les mécanismes offerts dépendent fortement des opérateurs mis à disposition dans les approches existantes.

FastFlow [2] est un exemple de modèle fournissant des squelettes algorithmiques permettant d'écrire et de composer des codes parallèles pour des architectures multicœurs à mémoire partagée avec cohérence de cache. Cette approche est segmentée en trois couches : la première offre des mécanismes de synchronisation sans verrou, la seconde fournit des collectives de communications de type producteur-consommateur, la dernière offre des patrons de codes parallèles principalement à base de flux de données (p. ex. pipeline, maître travailleur, diviser pour régner). Dans FastFlow, un programme parallèle consiste en une agrégation de plusieurs nœuds de traitement séquentiel opérant ensemble en parallèle et communiquant entre eux par des flux de données. Tout comme dans FlowVR, le type des données transmises ainsi que leur taille ou leur ordre d'envoi doivent être judicieusement choisis et la redistribution de structures de données complexes doit être réalisée par les concepteurs d'une application.

Connecteurs MPI

Dans le modèle de composants L^{2}C [20], les composants peuvent détenir des *interfaces MPI*. Ces interfaces permettent de partager un communicateur MPI avec d'autres instances de composants (nécessairement placées sur des processus MPI différents). Un *connecteur MPI* permet le partage d'un unique communicateur MPI par un groupe d'interfaces MPI. Les instances (placées sur des ressources distribuées) peuvent ensuite utiliser le communicateur pour interagir entre elles par le biais de collectives MPI. Une telle interface permet de rendre explicite les moyens de communication utilisant MPI entre les instances au niveau de l'assemblage.

2.5.2 L^{2}C : un modèle de composants minimaliste pour le HPC

Cette section présente L^{2}C [20], un modèle de composants utilisé comme base pour définir Comet présenté dans le chapitre 3 ainsi que pour faire fonctionner sa mise en œuvre Halley décrit dans le chapitre 4.

L^{2}C peut être vu à la fois comme une extension du processus de compilation modulaire et un modèle de composants de bas niveau proche du système. En effet, chaque composant est tout d'abord compilé en une bibliothèque (statique ou dynamique). Puis à l'exécution, les composants sont instanciés et les instances sont connectées en suivant un fichier de description d'assemblage.

L^{2}C permet une composition via l'appel de procédures natives C++, l'appel de procédure à distance via CORBA, et le passage de messages via MPI. Tout comme CCA, les composants L^{2}C fournissent des services par le biais de ports *provide*, en utilisent d'autres par le biais de ports *use*, et la communication entre les instances de composants consiste à appeler des procédures sur les ports. Les services doivent être déclarés en tant que classe abstraite C++ ou interface CORBA. L^{2}C fournit aussi des ports MPI qui permettent aux instances de composants de partager des communicateurs MPI. Les composants peuvent aussi exposer des attributs utilisés pour configurer les instances. En pratique, les composants L^{2}C sont des classes C++ auxquelles sont ajoutées des annotations nécessaires à la déclaration de composants (p. ex. nom, ports et propriétés).

Un assemblage L^{2}C peut être décrit en utilisant un fichier de description d'assemblage en XML (LAD) dont la spécification (schéma XML) est donnée en annexe B.6 (page 174). Ce fichier contient la description de toutes les instances de composants, leur configuration et les connexions entre les instances (*use-provide* et MPI). Chaque instance de composant est attachée à un processus et à chaque processus est associé à un point d'entrée (une interface sollicitée au démarrage de l'application).

2.5.3 Conclusion

Dans cette section, nous avons vu qu'il existe plusieurs méthodes permettant de composer des codes parallèles : la composition *use-provide* parallèle, le partage de données, les flux de données et de travaux, les squelettes algorithmiques ainsi que les connecteurs MPI.

L'usage d'une méthode plutôt qu'une autre dépend des codes à composer : le paradigme de programmation utilisé, les propriétés intrinsèques aux codes composés telles que la présence d'un état persistant nécessaire pour réaliser les traitements ainsi que la simplicité à adapter les codes afin qu'ils puissent interagir ensemble à travers la méthode de composition ciblée.

Il est important de noter que la majeure partie des méthodes de composition existantes présentées dans la section précédente font des hypothèses sur le paradigme de programmation des codes composés : les codes parallèles doivent utiliser soit des fils d'exécution tels que des processus MPI (principalement dans le cas de la composition *use-provide* ou de données partagées), soit des tâches persistantes qui interagissent par le biais de flux de données.

Dans une dernière partie, cette section a décrit L^{2}C, un modèle de composants dédié au calcul de haute performance qui est utilisé dans les chapitres 3 et 4.

2.6 Discussion

Cette section reprend les éléments et principes présentés dans les sections précédentes (c.-à-d. les sections 2.1, 2.2, 2.3, 2.4 et 2.5) afin de motiver et d'énoncer les objectifs de cette thèse.

2.6.1 Analyse

Dans la section 2.1, nous avons vu que les supercalculateurs d'aujourd'hui sont complexes et leur amélioration continue n'a pas cessé d'accroître leur complexité au fils des années, notamment par l'intégration de nouvelles formes de parallélisme, de hiérarchies de caches plus profondes, ou encore de systèmes de prédiction probabiliste. Exploiter efficacement ces architectures matérielles s'avère difficile. Alors que l'usage de systèmes NUMA et d'accélérateurs de calcul tend à se démocratiser [71, 74], programmer des applications numériques (telles que celles visant à résoudre des équations différentielles partielles) sur de ces architectures matérielles est aujourd'hui un véritable défi sans l'aide d'un paradigme de programmation adapté.

Ce fardeau devient quasiment insurmontable dans le cas où les concepteurs tentent de considérer d'autres critères tels que le support de plusieurs types d'architectures (p. ex. code ciblant les CPU et GPU) et la maintenabilité de l'application (problème évoqué dans la section 2.2).

Dans la section 2.3, nous avons vu que l'ordonnancement de graphes de tâches est une approche prometteuse visant à pallier les problèmes d'exploitation des architectures matérielles complexes et la portabilité des performances des codes HPC sur diverses architectures matérielles. Néanmoins, cette approche ne prend pas en compte les aspects liés à la maintenance des applications énoncés dans la section 2.4.

Les modèles de composants (présentés dans les sections 2.4 et 2.5) possèdent des propriétés qui favorisent la conception d'applications maintenables. De ce fait, une utilisation combinée des modèles de composants et de l'ordonnancement de graphes de tâches pourrait donc être manifestement bénéfique pour les applications HPC. Une telle combinaison apporterait un nouveau moyen de composer des codes HPC.

L'avantage majeur de cette forme de composition serait de permettre une exploitation efficace des ressources matérielles utilisées par les codes composés. Idéalement, cette forme de composition devrait être capable de combiner des codes HPC à base de tâches sans occasionner de sérialisation, synchronisation ou communication artificielle.

L'approche de composition existante la plus proche pour effectuer une telle opération consiste à utiliser des flux de données entre des composants. Cependant, les approches existantes imposent généralement une sérialisation du flux. De plus, comme nous l'avons évoqué précédemment, cette forme de composition ne convient pas à tous les usages. En particulier, elle s'avère être peu appropriée à l'échelle de l'ensemble du code Gysela, actuellement divisé en modules impératifs constitués d'un ensemble de fonctions interdépendantes (de granularité variable) partagées entre les modules. Dans ce dernier cas, la composition *use-provide* se révèle être particulièrement bien adaptée.

À notre connaissance, aucun modèle de composants existant n'offre toutes ces propriétés.

2.6.2 Objectifs et approche

Dans ce manuscrit, nous nous intéressons tout particulièrement à la composition de codes parallèles qui visent à produire un graphe de tâches. L'un des objectifs majeurs de cette approche est de pouvoir composer deux codes parallèles, qui seuls, exploitent efficacement les ressources à disposition et dont la composition engendre une exécution utilisant pleinement toutes les ressources nécessaires. Par exemple, en composant un code limité par la mémoire avec un code de calcul intensif, il est possible d'engendrer un code plus efficace que si chaque code est considéré indépendamment, notamment en régulant l'utilisation de la mémoire par l'exécution de tâches de calcul intensif (évitant ainsi la saturation d'une ressource partagée).

Dans les sections 2.2 et 2.3, nous avons vu que les applications HPC exploitent massivement le parallélisme de données afin de pouvoir passer à l'échelle. Pour cela, les applications HPC partitionnent généralement les données traitées. De plus, nous avons pu voir que bon nombre de supports exécutifs d'ordonnancement de graphes de tâches permettent de porter des dépendances entre les tâches par le biais de données. Ainsi, offrir des mécanismes de composition de codes HPC centrés sur des données pouvant être partitionnées semble être une bonne direction de recherche pour aborder ce problème.

Notre objectif est donc d'étudier la faisabilité d'une méthode de composition de codes parallèles à base de tâches préservant les propriétés suivantes :

— deux codes efficaces composés ensemble doivent être au moins aussi efficaces (c.-à-d. utilisation efficace des ressources après composition) ;
— deux codes ayant recours à des distributions de données différentes doivent être facilement composés sans introduire de synchronisations artificielles (c.-à-d. indépendance des codes composés et présence de mécanisme de redistribution des données entre les codes) ;
— une exécution entrelacée des codes parallèles doit être possible avec de faibles surcoûts (c.-à-d. composition de codes à grain fin).

2.7 Conclusion

Ce chapitre a présenté en six sections le contexte et l'état de l'art de la thèse qui se concentrent sur le calcul de haute performance et la composition de code parallèle à travers l'utilisation de modèles de composants.

Pour cela, la première section a étudié les architectures matérielles pour le calcul de haute performance. Cette section a montré en particulier leur complexité et leur variabilité ainsi que leur capacité à pouvoir exécuter des codes massivement parallèles en mettant en œuvre différentes formes de parallélisme.

La seconde section de ce chapitre a analysé les applications HPC s'exécutant tout naturellement sur les architectures matérielles présentées avant. Elle s'est concentrée plus particulièrement sur le cas de l'application de physique des plasmas par confinement magnétique pour la fusion Gysela, dont des sous-parties sont étudiées dans ce manuscrit. Cette section a fait ressortir plus spécifiquement les compromis

réalisés dans Gysela liant les performances du code à sa maintenabilité.

La troisième section a présenté les catégories de paradigmes de programmation les plus utilisés pour programmer des applications HPC. Elle a mis en évidence la variabilité des modèles de programmation existants et leurs singularités. Cette section a présenté plus particulièrement les approches à base de graphe de tâches particulièrement utiles pour utiliser efficacement les ressources de calcul des plateformes HPC ainsi que pour s'abstraire de la variabilité du matériel.

La quatrième section s'est concentrée sur la conception des applications HPC et plus précisément sur la structuration des codes HPC et les interactions qui lient les différentes parties d'une même application. Elle a présenté les approches à base de composants logiciels et a décrit les méthodes pouvant être employées pour composer des codes indépendamment de la présence du parallélisme en leur sein.

La cinquième section a décrit plus précisément les méthodes existantes permettant de composer des codes HPC. Elle a souligné que l'usage d'une méthode de composition plutôt qu'une autre dépend des codes à composer, en particulier du paradigme de programmation utilisé dans les codes composés.

Enfin, la dernière section a repris l'ensemble du contexte et de l'état de l'art exposé dans ce manuscrit afin d'évoquer les motivations de cette thèse ainsi que ces objectifs. En particulier, elle s'est intéressée au cas de la composition de codes parallèles utilisant des approches à base de graphes de tâches qui, à notre connaissance, n'est pas traitée dans les modèles de composants existants.

Le chapitre suivant présente COMET, un modèle de composants pour le calcul de haute performance supportant l'ordonnancement de graphes de tâches et offrant des mécanismes pour la composition de ces graphes de tâches.

Chapitre 3

Le modèle COMET

Contents

Cette section présente une des contributions de la thèse : un modèle à composant avec support de l'ordonnancement de tâches. La section 3.1 donne un aperçu du modèle COMET et de sa structure. La section 3.2 présente les entités fondamentales de COMET. La section 3.3 montre comment construire une application en composant les entités fondamentales du modèle. Enfin, la section 3.5 conclut ce chapitre.

3.1 Aperçu

COMET est un modèle de composants dédié au calcul haute performance et supportant l'ordonnancement de graphes de tâches. Il vise à assembler simplement des codes parallèles indépendants tout en permettant une exécution efficace. Pour cela, COMET se base sur le modèle de composants existant L^2C qui supporte la composition *use-provide* et *MPI*. Il l'étend en supportant une nouvelle forme d'interaction entre les composants à base de flux de données manipulant des données partitionnées ainsi qu'en ajustant les ports *use-provide*. Cette extension permet de construire un graphe de tâches en composant successivement des morceaux de graphe de tâches.

COMET cible en particulier les architectures à mémoire partagée. Néanmoins, il supporte aussi les architectures à mémoire distribuée en se basant sur MPI pour la communication entre les nœuds de calcul.

3.2 Unités de composition

COMET repose sur deux formes d'*unités de composition*, c'est-à-dire des unités logicielles disposant d'interfaces visant à être connectées ensemble. D'une part, les *composants* qui sont des briques logicielles détenant une partie de la mise en œuvre d'une application. D'autre part, les *métatâches* qui sont des unités d'ordonnancement définissant la structure de morceaux de calculs réalisés par une application.

3.2.1 Composants

Un *composant* COMET est un morceau de code doté d'interfaces bien définies. Un *port de composant* peut être un *attribut*, un port *use*, un port *provide*, un port *MPI* ou un *port de données*. Les composants de COMET peuvent être vu comme des composants L^2C (présenté dans la section 2.5.2) étendus avec des ports de données.

Interfaces de composants Un attribut décrit une valeur d'un type bien défini (p. ex. chaîne de caractères, entier), fixée à l'exécution pour configurer un composant. Un port *provide* décrit un service fourni par le composant (typiquement une

```
Component SampleCompo
    provide     Interpol  pInter
    use         LinearAlg uLinAl
    indata      Array1D   input
    outdata     Array1D   output
    inoutdata   Array2D   inoutput
    attribute   String    aName
```

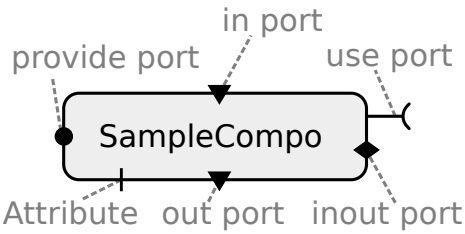

FIGURE 3.1 – Exemple de définition de composant (gauche) et sa représentation graphique (droite).

interface orientée objet avec un ensemble de méthodes). Un port *use* décrit un service requis par le composant (il permet d'exécuter des méthodes sur une interface orientée objet fournie par un autre composant). Un port *MPI* partage un communicateur MPI entre des composants permettant ainsi des communications point à point ou collectives entre des parties d'une application répartie sur différents nœuds de calculs.

Un *port de données* permet à un composant de partager des *buffers de données* d'un type bien défini (p. ex. tableau bidimensionnel). Un *buffer de données* est un conteneur de données (c.-à-d. information quelconque). C'est une instance d'une structure de données qui à la propriété d'être adressable et d'occuper une zone potentiellement non contiguë en mémoire partagée. Les types de buffers de données disponibles ne sont pas définis par le modèle lui-même. C'est la mise en œuvre de COMET qui est responsable de la définition des types disponibles et de moyens pour les étendre. Ainsi, un type buffer de données est un identifiant qui peut prendre la forme d'une chaîne de caractères telle que "Array2D". Il dispose d'un *mode d'accès* qui peut être : *in* pour un accès en lecture seule, *out* pour un accès en écriture-seule et *inout* pour un accès simultanément en lecture et écriture. Un port de données possède un état décrivant l'accessibilité du buffer de données partagé. Les valeurs possibles pour cet état sont *disponible*s et *occupé*es. Un port de données est disponible si aucune autre unité de composition n'utilise le buffer partagé (lecture ou écriture).

Exemple La figure 3.1 montre un exemple de composant et sa représentation graphique. Le langage utilisé pour décrire le composant et plus généralement dans ce chapitre n'est pas celui de la mise en œuvre de COMET. Il est uniquement utilisé à des fins pédagogiques pour présenter de façon concise les informations nécessaires à la spécification d'éléments du modèle ainsi que la manière dont elles sont structurées. Des exemples de définitions sont donnés dans le chapitre 4.

Propriétaire des données Un composant qui partage un buffer de données en est le responsable : il contrôle son cycle de vie. À travers un port de données, un composant transfère temporairement la responsabilité du buffer de données à ceux qui peuvent y accéder (jusqu'à ce que le port soit disponible). Un composant peut sonder l'état d'un port de données ou attendre que le buffer soit disponible sur ce port.

Port use/provide et effets de bords Les ports *use* et *provide* possèdent une propriété *task-safe* permettant d'indiquer qu'ils fournissent ou requièrent un service *task-safe*. Un service *task-safe* est utilisable dans un contexte d'une tâche. Cette propriété est fixée explicitement par les concepteurs d'un composant (c.-à-d. valeur booléenne précisant si le port est *task-safe* ou non). Étant donné que des tâches peuvent s'exécuter en parallèle, cela signifie qu'un service *task-safe* doit être compatible avec une utilisation concurrente (c.-à-d. thread-safe). Un port *use task-safe* indique que le port requiert un service utilisable dans un contexte d'une tâche. Cependant, la mise en œuvre de COMET peut ajouter des contraintes supplémentaires à ce type de service. En effet, elle peut avoir recours à des supports exécutifs d'ordonnancement de tâches qui imposent des contraintes sur l'implémentation de ces services tels que la non-présence de verrous dans le code exécuté ou encore sa réentrance. Le modèle COMET considère que les services *task-safe* sont sans effet de bord, c'est-à-dire que leur utilisation n'a pas d'impact sur le comportement du composant qui les fournit vu de l'extérieur (c.-à-d. pas d'influence sur le comportement de l'ensemble des services proposés par le composant). L'ensemble des services *task-safe* sont un sous-ensemble des services thread-safe et un sur-ensemble des services sans effet de bord.

3.2.2 Métatâches

Aperçu Une *métatâche* est une unité de composition décrivant une partie d'un calcul parallèle à base de tâches. Elle peut être perçue comme une abstraction d'une portion d'un graphe de tâches : une métatâche décrit la structure d'un morceau de graphe de tâches (ensemble des tâches et des dépendances entre les tâches) et possède des interfaces bien définies permettant d'assembler cette portion avec d'autres parties. Cette unité a pour but de soumettre, à l'exécution, un ensemble de tâches exécutant un même code. Un port de métatâche est soit un attribut, soit un port *use* (requis et unique), soit un port de données.

Interfaces de métatâches Un *attribut de métatâche* est identique à celui d'un composant et remplit le même objectif : configurer la métatâche. Plus particulièrement, cette configuration peut contenir des valeurs nécessaires au paramétrage des ports de la métatâche. Une métatâche possède aussi un unique port *use compute* qui requiert un service définissant l'implémentation des tâches soumises à l'exécution. Ce port *use* à la propriété d'être *task-safe*. Une métatâche détient au moins un port de données. Un *port de données de métatâche* diffère de celui d'un composant sur un seul aspect : il opère sur des buffers de données partitionnés.

Exemple simplifié de métatâche La figure 3.2 présente un exemple simplifié de métatâche, opérant sur un unique buffer de données en entrée-sortie. La figure décrit un cas simplifié où le buffer de données traité est partitionné en un seul fragment (des cas plus complexes faisant intervenir plusieurs fragments sont détaillés dans la suite cette section). La figure montre aussi un exemple de graphe de tâches pouvant

FIGURE 3.2 – Exemple simplifié d'une définition de métatâche (gauche), sa représentation graphique (milieu) et un graphe de tâches associé pouvant être produit à l'exécution par la métatâche (droite).

être généré par la métatâche : les rectangles sont des buffers de données et les cercles des tâches. C'est un exemple minimaliste d'une métatâche produisant qu'une seule tâche qui lit et écrit dans le même buffer de données.

Ports de données et partitionnements Un buffer de données partitionné est un buffer divisé logiquement en sous-ensembles disjoints appelés *fragments*. Une description plus formelle est donnée par la définition 1 [1]. Un fragment est accessible par un *index* : une instance de structure de données dont le type est défini par le partitionnement (p. ex. couple d'entiers représentant la position (x, y) d'un bloc dans un tableau bidimensionnel). Il peut aussi détenir des métadonnées (p. ex. taille du bloc, pointeur vers une zone mémoire). Tout comme les définitions de types de données non partitionnés, les types de données partitionnées disponibles et leur extensibilité ne sont pas définis par le modèle lui-même, mais par sa mise en œuvre. Ainsi, un type de données partitionné est un identifiant qui peut prendre la forme d'une chaîne de caractères telle que "Array2D_Bloc". Il est associé à un type de données, afin d'assurer la compatibilité entre les types de données partitionnées et non partitionnées.

Définition 1 *Soit F l'ensemble des fragments. Soit B l'ensemble des buffers de données. On note $\mathcal{P}(E)$ l'ensemble des sous-ensembles d'un ensemble E [2]. Un partitionnement est défini par une fonction part qui à tout buffer de données associe un ensemble de fragments :*

$$\begin{aligned} &part : B \to \mathcal{P}(F) \\ &tel\ que : \\ &\forall b_1 \in B, \forall b_2 \in B : \\ &b_1 \neq b_2 \implies part(b_1) \cap part(b_2) = \varnothing \end{aligned} \tag{3.1}$$

On dénote par $fragof$ la fonction permettant de déterminer le buffer de données

1. Dans COMET, une fonction de partitionnement peut ne pas couvrir l'intégralité de l'espace d'un buffer de données.
2. $X \in \mathcal{P}(E) \iff X \subseteq E$

associé à un fragment :

$$\begin{aligned} &fragof : F \rightarrow B \\ &tel\ que : \\ &\forall b \in B, \forall f \in F : \\ &fragof(f) = b \iff f \in part(b) \end{aligned} \tag{3.2}$$

Expressions et langages de relations Le partitionnement des buffers de données est la source du parallélisme issu des métatâches (parallélisme de données). Il permet à cette unité de composition de soumettre des tâches opérant sur des fragments de buffers de données tout en évitant des situations de compétition au niveau de l'application entière grâce aux modes d'accès des ports de données (c.-à-d. cohérence des accès mutuels) : deux tâches issues d'une même métatâche ou de métatâches différentes ne peuvent pas opérer simultanément sur un même fragment en écriture [3] (ou l'une en lecture et l'autre en écriture). Il est aussi et surtout la source de la composition parallèle efficace des métatâches en permettant une exécution entrelaçant les traitements opérant sur chaque fragment. L'ensemble des tâches à soumettre est décrit par des *relations*. L'objectif des relations est de déterminer le nombre de tâches à créer et de spécifier, pour chaque tâche, l'ensemble des fragments (issus de buffers de données partitionnés) en dépendances. La définition 2 décrit plus formellement ce que sont des relations. Ces relations sont décrites à travers une *expression relationnelle* définie dans un *langage de relations*. Une expression relationnelle peut prendre typiquement la forme d'une expression mathématique (p. ex. expression algébrique) sérialisée sous la forme d'une chaîne de caractères. Des exemples de langages de relations sont détaillés dans le chapitre 4.5. L'élaboration d'un langage de relations permettant de définir le minimum de dépendances nécessaire (c.-à-d. sans approximation), de manière concise, se révèle très complexe dans le cas général. Pour cette raison, le support des langages de relations disponibles est conçu pour être extensible de telle manière à pouvoir utiliser un DSL adapté aux cas traités. Par conséquent, les métatâches contiennent une expression relationnelle, définie sous la forme d'une simple chaîne de caractères, ainsi qu'un langage de relations associé, spécifié via un identifiant. L'utilisation d'un langage de bas-niveau et Turing-complet tel que le langage C pour définir les dépendances n'est pas exclu. Un tel langage permet d'exprimer des graphes de tâches complexes, dans les cas inhabituels, à défaut de cibler uniquement des experts de COMET. Tout comme les types de buffers de données, les langages disponibles sont implémentés dans la mise en œuvre de COMET et décrits dans la section 4.5.

Définition 2 *Une relation est définie par une fonction rel qui à partir d'un ensemble de buffers de données produit un ensemble d'ensembles de fragments représentant, pour chaque sous-ensemble de fragments, les dépendances d'une unique tâche*

3. Certains supports exécutifs supportent un mode d'accès concurrent ou exclusif aux données, cependant ils ne sont pas supportés par la version actuelle de COMET

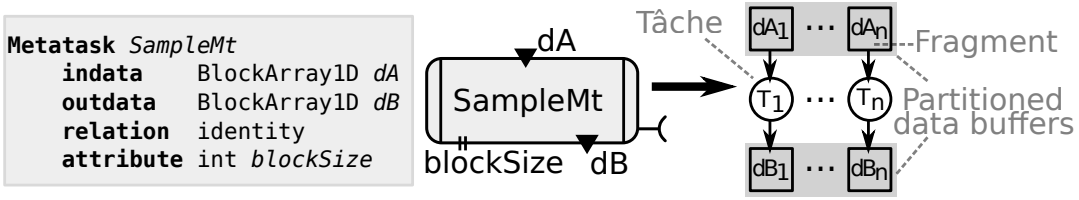

FIGURE 3.3 – Exemple de définition de métatâche avec deux ports de données (gauche), sa représentation graphique (milieu) et un graphe de tâches associé pouvant être produit à l'exécution par la métatâche (droite).

à soumettre :

$$\begin{aligned} &rel : \mathcal{P}(B) \rightarrow \mathcal{P}(\mathcal{P}(F)) \\ &\textit{tel que :} \\ &\forall B' \subseteq B, \forall F' \subseteq F : \\ &F' \in rel(B') \implies (\forall f \in F' : fragof(f) \in B') \end{aligned} \tag{3.3}$$

Exemple de relations Trois langages de relations ont été définis dans HALLEY afin de montrer la faisabilité de l'approche : *identity*, *align* et *transpose*. Ces langages considèrent le cas où toutes les tâches produites par une même métatâche sont indépendantes entre elles. Les exemples de cette section utilisent le langage *identity*. Ce langage ne requiert aucune expression relationnelle. Il suppose que tous les buffers de données de la métatâche possèdent le même partitionnement (nombre de fragments identiques avec les mêmes indexes). Il fait correspondre les fragments qui ont le même index entre eux. Les deux autres langages sont présentés dans le chapitre 4 étant donné qu'ils ne font pas partie intégrante du modèle, COMET mais plutôt de sa mise en œuvre HALLEY.

Exemple de métatâche La figure 3.3 présente une définition complète de métatâche avec deux ports (entrée et sortie) opérant ainsi sur deux buffers de données différents. Le partitionnement des buffers utilisé ici est un partitionnement en bloc de taille fixe. La taille des blocs est fournie par l'attribut `blocksize`. Le lien entre les attributs de métatâche et ceux des buffers de données partitionnés relève de la mise en œuvre et est expliqué plus en détail dans le chapitre 4. Le langage d'alignement *identity* utilisé permet d'exprimer un calcul trivialement parallèle sans interaction entre les tâches (c.-à-d. *embarrassingly parallel*).

3.3 Assemblages

La description d'une application COMET est réalisée par un *assemblage*. Un tel assemblage contient des *instances de composants* (contenant des portions de codes de l'application), un point d'entrée, des connexions ainsi que des *sections dataflow*

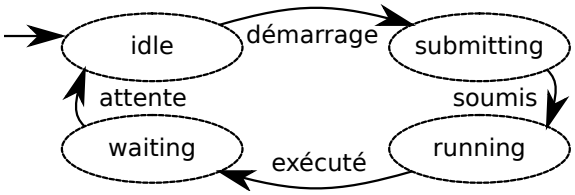

FIGURE 3.4 – Automate d'état fini d'une section dataflow.

(décrivant des calculs sous la forme de flux de données) qui, elles-mêmes, contiennent des *instances de métatâches* (décrivant comment soumettre une portion d'un graphe de tâches).

3.3.1 Instances

Une *instance* de composant ou de métatâche est une entité du modèle qui référence respectivement un type composant ou un type de métatâche et qui possède une *configuration* (un état initial déterminé par l'assemblage). Une instance détient des ports (issus des types référencés) qui sont destinés à être connectés à d'autres ports (issus d'autres instances). Une configuration d'une instance assigne une valeur à chaque attribut spécifié par le type de l'instance.

3.3.2 Points d'entrées

Un *point d'entrée* d'un assemblage COMET référence un port *provide*. Il définit un port *provide* utilisé pour démarrer l'exécution de l'assemblage.

3.3.3 Connexions

Un *assemblage* COMET contient deux sortes de connexions : des connexions *use-provide* et des connexions de données.

Une *connexion use-provide* connecte un port *use* à un port *provide*. L'instance de composant exposant le port *use* peut utiliser un service fourni par le port *provide* (il peut appeler les méthodes fournies par le port *provide*). Les ports *use task-safe* ne peuvent être connectés qu'à des ports *task-safe*.

Une *connexion de données* est orientée. Elle connecte un port de données de sortie (source, p. ex. `out` ou `inout`) à un port de données d'entrée (destination, p. ex. *in* ou *inout*). Elle exprime un flux de données circulant du port source vers le port destination (c.-à-d. flux d'informations). Les connexions autorisées par le modèle sont régies par des règles présentées en détail dans la section 3.3.5.

3.3.4 Sections dataflow

Une *section dataflow* est une entité composée d'un groupe d'instances de métatâches interconnectées. Leur composition vise à générer un graphe de tâches à la

demande effectuant des calculs sur des données mises à disposition par des composants. La section dataflow a pour objectif principal de contrôler la soumission du graphe de tâches. Pour cela, une section expose un port provide permettant de déclencher la soumission du graphe et un autre port provide visant à sonder l'état de la section.

Une section peut être dans quatre états différents :

— `idle` : la section est prête à produire un graphe de tâches à la demande ;
— `submitting` : la soumission du graphe de tâches a été demandée et est en cours de soumission (des tâches peuvent être exécutées durant cet état) ;
— `running` : toutes les tâches nécessaires à l'exécution du graphe de tâches ont été soumises et sont donc exécutables[4] (l'ordonnanceur est responsable de l'exécution des tâches soumises par une section) ;
— `waiting` : toutes les tâches produites par la section ont été exécutées et les composants peuvent maintenant accéder aux données.

Les transitions possibles entre les états du dataflow sont décrites par la figure 3.4. Les transitions de démarrage et d'attente sont activées à travers le port de contrôle de la section. Lorsque la section est dans l'état `idle` ou `waiting`, les composants connectés à la section peuvent partager des données à travers leurs ports de données. Une fois que tous les ports de données interagissant avec la section ont partagé un buffer de données, la section peut entrer dans l'état `submitting`. Ce changement d'état se fait à la demande des composants connectés au port de contrôle. Le concepteur de l'assemblage est responsable de la coordination entre tous les composants interagissant avec la section. Cela implique aussi que les composants coordonnés possèdent des interfaces permettant une telle synchronisation.

Il est important de noter que l'état `submitting` permet une exécution des tâches. En effet, OpenMP et certains supports exécutifs ne permettent pas de dissocier la soumission de l'exécution des tâches. Cela leur permet notamment d'exécuter directement des tâches soumises lorsque la taille du graphe est imposante, réduisant ainsi des coûts d'ordonnancement (amélioration des performances et réduction de l'occupation mémoire). COMET distingue néanmoins les états `submitting` et `running` afin de permettre aux composants d'interagir avec plusieurs sections sans attendre l'exécution des graphes de tâches générés à chaque fois évitant ainsi des synchronisations inutiles.

La figure 3.5 présente un exemple d'assemblage complet avec deux composants et une section contenant une seule métatâche. L'exécution de cet assemblage fonctionne de la manière suivante. Tout d'abord, le composant master configure son port de données `dOut` avec un buffer de données valide. Ensuite, il démarre la section à travers son port *use* `ctrl` (port de contrôle, c.-à-d. un port provide dont l'interface objet est entièrement déterminée par la mise en œuvre de COMET) produisant ainsi un graphe de tâches. Puis, il attend que le graphe de tâches produit par la section soit exécuté par le biais de ce même port de contrôle. Le buffer de données est

4. Plusieurs sections peuvent soumettre des tâches et être ensuite simultanément dans l'état `running`. Des tâches issues de multiples sections, dans l'état `running`, peuvent alors s'exécuter en parallèle.

```
1: Master master
2: TaskImpl tskImpl

3: Section aSection
4:  SampleMt sampleMt(blockSize: 16)
5:  sampleMt.compute -- tskImpl.compute
6:  master.dOut --> sampleMt.dA
7:  sampleMt.dB --> master.dIn

8: master.control -- aSection.control

9: entryPoint:master.go
```

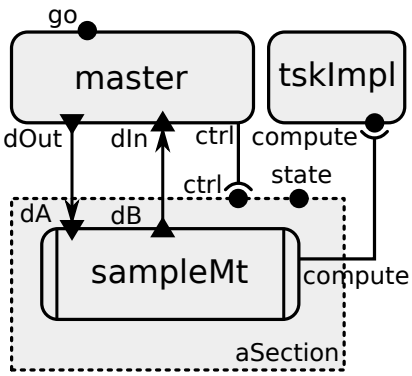

FIGURE 3.5 – Exemple minimaliste d'assemblage COMET. Lignes 1 et 2 : instanciation de deux composants. Lignes 3 à 7 : déclaration d'une section dataflow. Ligne 4 : instanciation d'une métatâche. Lignes 5 et 8 : connexions *use-provide*. Lignes 6 et 7 : connexions de données. Ligne 9 : déclaration du point d'entrée de l'assemblage. Les connexions use-provide sont dénotées par "–" et ne sont pas dirigées (la relation d'ordre peut être déduite du types de ports connectés). Les connexions de données sont dénotées par "–>" et sont dirigées (la relation d'ordre décrit la direction du flot d'informations et n'est pas entièrement déductible des types de ports connectés).

partitionné en fragments. La métatâche `sampleMt` soumet des tâches qui ont comme dépendances des fragments de ce buffer. Dès qu'une tâche de la métatâche `sampleMt` est exécutée, une méthode du service `compute`, fourni par `taskImpl`, est appelée. Une fois que le graphe de tâches est exécuté, l'attente du composant est interrompue lui permettant ainsi de recevoir les données qui ont été traitées à travers son port de données `dIn`. Le port `state` permet de sonder l'état de la section : il fournis aux composants un moyen de déterminer, au moment précis de l'exécution, si la section soumet des tâches, les exécutes ou si toute les tâches résultant de sa dernière exécution ont été exécutées. Ce dernier port n'est pas utilisé en pratique dans cet assemblage car un seul composant interagit avec la section.

Les résultats d'une métatâche peuvent être stockés dans des buffers de données temporaires. Plusieurs métatâches d'une même section peuvent réutiliser un même buffer. De plus, les fragments de buffers de données temporaires peuvent être créés à la demande en fonction de l'ordonnancement des tâches étant donné que les buffers de données sont partitionnés et que les dépendances de fragments sont bien définies. Un ordonnancement en profondeur peut ainsi favoriser une réutilisation des fragments et réduire l'espace mémoire utilisé. Un tel buffer de données doit être déclaré dans une section dataflow. Il peut être connecté à un port *out* à travers une connexion de données temporaires. Le cycle de vie d'un buffer de données temporaire est automatiquement géré par la section dataflow dans laquelle il réside.

La figure 3.6 montre un exemple d'assemblage contenant un buffer de données temporaire. Dans cet exemple, les tâches de `sampleMt1` écrivent dans les fragments temporaires de `buff` et peuvent être directement réutilisés ensuite par les tâches de `sampleMt2` avant d'être finalement détruits ou recyclés par le support exécutif.

```
Master master
TaskImpl tskImpl

Section sampleSection
    SampleMt sampleMt1(blockSize: 16)
    SampleMt sampleMt2(blockSize: 32)
    Buffer buff
    buff --> sampleMt.dB
    sampleMt1.compute -- tskImpl.compute
    sampleMt2.compute -- tskImpl.compute
    master.dOut --> sampleMt1.dA
    sampleMt1.dB --> sampleMt2.dA
    sampleMt2.dB --> master.dIn

master.control -- sampleSection.control
entryPoint:master.go
```

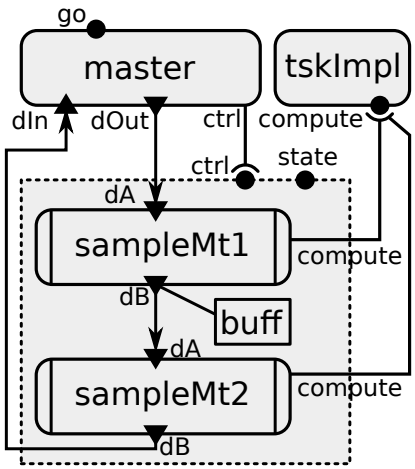

FIGURE 3.6 – Exemple de déclaration de buffer de données temporaire (mot clef : `Buffer`) et de son utilisation.

Lorsque deux métatâches interconnectées m_1 et m_2 opèrent sur un même buffer de données, b elles peuvent avoir recours à des partitionnements différents. Il est alors nécessaire d'effectuer un repartitionnement. Une telle opération consiste à définir des dépendances entre le groupe de tâches généré par m_1 et celui généré par m_2 pour le buffer b. Cette opération est nécessaire pour imposer un ordre d'exécution correct. La définition 3 décrit plus formellement ce qu'est une telle opération. Plusieurs stratégies de repartitionnement sont possibles : d'une barrière globale risquant d'effectuer une synchronisation excessive entre les tâches, à une définition précise des dépendances pouvant introduire un surcoût élevé. Les stratégies de repartitionnement disponibles ne sont pas spécifiées par le modèle, COMET mais plutôt par sa mise en œuvre qui est responsable de choisir la meilleure. Elles sont donc détaillées plus précisément dans le chapitre 4.

Définition 3 *Un repartitionnement est défini par une fonction repart qui pour chaque fragment d'un partitionnement destination donne les fragments qui le chevauchent (dépendances) :*

$$\begin{aligned} &repart : F \rightarrow \mathcal{P}(F) \\ &\textit{tel que :} \\ &\forall b \in B, f \in F, F' \subseteq F, f' \in F' : \\ &(F' = repart(f) \wedge fragof(f) = b) \implies fragof(f') = b \end{aligned} \tag{3.4}$$

Il est important de noter que le choix d'une stratégie de synchronisation locale n'a aucun impact sur la validité de l'exécution d'un assemblage (p. ex. sur-synchronisation engendrant une situation d'interblocage). En effet, une section engendre la soumission d'un graphe de tâches uniquement à la demande d'un composant. Cela signifie que l'ensemble des buffers de données nécessaires à la section ont été fournis précédemment. Or, la section n'a besoin que de buffers pour engendrer

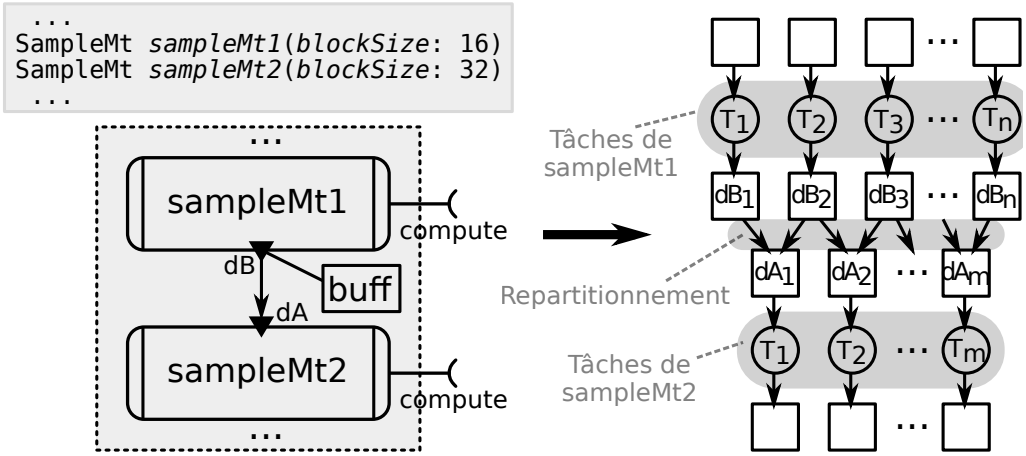

FIGURE 3.7 – Exemple d'un morceau de section dataflow issu de l'assemblage de la figure 3.6 avec un repartitionnement d'un buffer de données entre deux métatâches (gauche) et un graphe de tâches associé pouvant être produit à l'exécution (droite).

un graphe de tâches et les opérations de repartitionnement ne dépendent que d'éléments internes à la section. Ils n'ont pour effet que d'impacter la structure du graphe de tâches produit. De plus, l'ordre de soumission des tâches est un ordre d'exécution valide. Donc le choix d'une stratégie de synchronisation ne peut pas être responsable de situations d'interblocage. Plus généralement, une section ne peut engendrer une situation d'interblocage qu'en présence de cycle au niveau des connexions de données dans l'assemblage, ce qui est interdit par les règles de validation décrite en section 3.3.5.

La figure 3.7 présente un exemple de section dataflow impliquant un repartitionnement entre deux métatâches opérant sur un unique buffer de données dont le partitionnement diffère en fonction la métatâche. La présence d'un repartitionnement provient des valeurs différentes de l'attribut `blockSize`, un attribut interne aux deux métatâches permettant de définir les paramètres de partitionnement des ports `dA` et `dB`, pour chacune des métatâches. La figure montre aussi l'impact du repartitionnement sur le graphe de tâches généré à l'exécution (ajout de dépendances entre les fragments sur lesquels les deux groupes de tâches opèrent). Du point de vue du concepteur de l'assemblage et des unités de compositions, le repartitionnement est une opération transparente et ne nécessite donc aucun effort. À l'exécution, le repartitionnement crée des dépendances entre les tâches de telle manière à ce que les tâches de `sampleMt1` et de `sampleMt2` qui opèrent sur des fragments qui se chevauchent ne puissent pas s'exécuter en même temps.

3.3.5 Règles de validation additionnelles

En plus des règles énoncées dans les sections précédentes, un ensemble de règles supplémentaires permettent de définir la validité d'un assemblage.

La figure 3.1 présente l'ensemble des connexions possibles entre les ports de données de composants et de métatâches. On peut noter que les ports *in* ne peuvent

source \ destination		composant			métatâche		
		in	out	inout	in	out	inout
composant	in						
	out				✓		✓
	inout				✓		✓
métatâche	in						
	out	✓		✓	✓		✓
	inout	✓		✓	✓		✓

TABLE 3.1 – Connexions possibles entre les ports données de composants et de métatâches

être la source d'une connexion de données, que les ports *out* ne peuvent être la destination d'une connexion de données et que les ports de données des composants ne peuvent être directement connectés à d'autres composants. Cette dernière contrainte est la conséquence d'une hypothèse nécessaire au bon fonctionnement des composants : un composant est le possesseur des données qu'il partage.

Définition et notations Soit C l'ensemble des composants. Soit M l'ensemble des métatâches. Soit P_{data} l'ensemble des ports de données. Soit $X_{data} \subset P_{data} \times P_{data}$ l'ensemble des connexions de données. Soit $owner : P_{data} \to (C \cup M)$ la fonction déterminant l'appartenance des ports de données aux unités de compositions. Soit $P_{in} \subseteq P_{data}$, $P_{out} \subseteq P_{data}$ et $P_{inout} \subseteq P_{data}$ respectivement l'ensemble des ports *in*, *out* et *inout* tel que $P_{in} \cup P_{out} \cup P_{inout} = P_{data}$. Soit *path* le prédicat déterminant s'il existe un chemin de taille non nulle de connexions de données décrit par la définition 3.5. Soit $path_m$ une variante du prédicat *path* où un chemin doit passer nécessairement par des ports de données de métatâches décrite par la définition 3.6. Soit S l'ensemble des sections dataflow. Soit $section : M \to S$ la fonction déterminant l'appartenance des métatâches aux sections dataflow.

$$\begin{aligned} path : P_{data} \times P_{data} \to & \mathbb{B} \\ x, y \mapsto & \langle x, y \rangle \in X_{data} \\ & \vee (\exists w \in P_{data} : path(x, w) \wedge path(w, y)) \end{aligned} \tag{3.5}$$

$$\begin{aligned} path_m : P_{data} \times P_{data} \to & \mathbb{B} \\ x, y \mapsto & \langle x, y \rangle \in X_{data} \wedge owner(x) \in M \wedge owner(y) \in M \\ & \vee (\exists w \in P_{data} : path_m(x, w) \wedge path_m(w, y)) \end{aligned} \tag{3.6}$$

Propriété 1 *Tous les ports* in *et* inout *des métatâches doivent être la cible d'une connexion ascendante.*

$$\begin{aligned} & \forall p_d \in P_{data}, \exists p_s \in P_{data} : \\ & (owner(p_d) \in M \wedge (p_d \in P_{in} \vee p_d \in P_{inout})) \implies \langle p_s, p_d \rangle \in X_{data} \end{aligned} \tag{3.7}$$

Propriété 2 *Le graphe formé par connexions entre les ports de données des métatâches est un graphe dirigé acyclique.*

$$\begin{aligned} &\forall p_1 \in P_{data}, \forall p_2 \in P_{data} : \\ &\neg path(p_1, p_2) \vee \neg path(p_2, p_1) \end{aligned} \tag{3.8}$$

Propriété 3 *Les chemins formés par les connexions de données et ayant comme source un port* out *d'un composant ne peuvent pas passer par un port de composant. Il en est de même pour les chemins dont la source est un port inout, excepté que ces chemins peuvent se terminer par le port source formant ainsi un cycle (cas où des buffers de données sont utilisés en écriture).*

$$\begin{aligned} &\forall p_s \in P_{out}, \forall p_d \in P_{data} : \\ &(owner(p_s) \in C \wedge path(p_s, p_d)) \implies owner(p_d) \in M \end{aligned} \tag{3.9}$$

$$\begin{aligned} &\forall p_s \in P_{inout}, \forall p_d \in P_{data} : \\ &(owner(p_s) \in C \wedge path(p_s, p_d)) \implies (p_s = p_d \vee owner(p_d) \in M) \end{aligned} \tag{3.10}$$

Propriété 4 *Une connexion de données ne peut pas connecter deux métatâches appartenant à des sections dataflow différentes.*

$$\begin{aligned} &\forall \langle p_s, p_d \rangle \in X_{data}, \exists \langle m_s, m_d \rangle \in M \times M : \\ &(m_1 = owner(p_s) \wedge m_2 = owner(p_d)) \implies section(m_s) = section(m_d) \end{aligned} \tag{3.11}$$

La propriété 1 assure la présence de tous les buffers de données requis par une métatâche. La propriété 2 assure la terminaison des sections dataflow en empêchant les définitions circulaires. La propriété 3 assure qu'un buffer de données est partagé simultanément par, au plus, un unique composant. La propriété 4 assure que les sections dataflow sont indépendantes. Cette contrainte a pour effet de simplifier grandement le modèle et sa mise en œuvre en traitant les sections dataflow de manière isolée.

3.4 Discussion

3.4.1 Expressivité et limitations

Dynamicité Une description d'assemblage COMET est statique, c'est-à-dire qu'elle ne peut pas changer à l'exécution. Ainsi, la structure abstraite du graphe de tâches produit par une section dataflow ne peut changer au cours de l'exécution d'une application (ce n'est pas le cas de l'ensemble du graphe de tâche produit par l'assemblage entier puisque les sections sont contrôlées par des composants dont le comportement peut changer à l'exécution). Cette limitation issue du modèle est volontaire et peut être levé ultérieurement. En effet, ce manuscrit s'intéresse particulièrement aux applications HPC numériques dont la structure abstraite du graphe de tâche produit

à l'exécution n'est généralement pas dynamique. L'introduction de graphe de tâche dont la structure est dynamique peut se faire à l'aide de plusieurs méthodes, dont la reconfiguration d'assemblage et constitue un axe de recherche orthogonale à cette thèse.

Flux de contrôle Le flux de contrôle d'une application n'est pas visible au niveau d'un assemblage COMET. Il est défini uniquement par l'implémentation des composants qui constituent un assemblage. Par conséquent, le modèle tel que décrit dans ce chapitre ne supporte pas l'expression de structures de contrôle dans les sections dataflow. Il ne fournit pas non plus de mécanismes équivalents. Par exemple, il ne permet pas d'adapter la présence d'une métatâche en fonction d'une expression conditionnelle, ou encore d'exprimer des boucles itératives à l'intérieur d'une section (p. ex. processus itératif avec analyse de convergence).

Synchronisation La présence d'un état interne associé aux sections dataflow (nécessaire au contrôle du cycle de vie des données) empêche de soumettre un graphe de tâches issu du déclenchement de multiples itérations d'une même section sans introduire de synchronisation artificielle. En effet, le déclenchement d'une section requiert la fin de l'exécution du dernier graphe de tâches produit. De plus, les composants ne peuvent partager que des buffers non partitionnés aux sections. Ainsi, lorsqu'un buffer de données est traité par une section, puis par une seconde (ou la même section utilisée plus tard), les informations de partitionnement ne sont pas transmises d'une section à une autre, car le cycle de vie des fragments issus du partitionnement n'est valable que dans une section (il fait partie de l'état interne de la section). Cela signifie que tout transfert de données entre plusieurs sections ou plusieurs exécutions d'une même section introduit une synchronisation artificielle locale aux sections considérées.

Expressivité des métatâches L'expressivité d'une métatâche est intrinsèquement liée à celle d'expressivité d'une expression relationnelle. La définition formelle 2 permet d'exprimer des calculs totalement indépendants (c.-à-d. embarassingly parallel), mais aussi des cas de stencils ou de transpositions sur des tableaux multidimensionnels. Ces cas sont supportés par des langages mis au point expérimentalement, présenté dans chapitre 4 et évalué dans le chapitre 5. D'autres cas, tels que les réductions sont supportées par la définition formelle 2 sous l'hypothèse que tous les buffers de données nécessaires à la réduction (nombre fixe de buffers temporaires partitionnés intermédiaires) sont fournis explicitement à la métatâche. La conception d'un langage supportant efficacement les cas de réductions reste un problème à étudier, tout comme le support d'autres patrons de calcul parallèle (p. ex. pipeline, scan ou encore des traitements parallèles spécifiques "sur-mesures").

3.4.2 Compatibilité et validité

La propriété *task-safe* est nécessaire au concepteur d'un assemblage COMET. En effet, elle lui permet d'assurer la compatibilité des services requis avec les services fournis (et donc la cohérence de l'assemblage) sur un aspect spécifique : le support du parallélisme. Cela a pour effet de contraindre ainsi les mises en œuvres possibles des services considérés. Mais cet aspect n'est pas le seul pouvant rompre la compatibilité des services. D'autres, tels que la non-présence d'entrée-sortie ou encore la garantie d'une borne maximale sur le temps d'exécution, peuvent apparaître dans les applications. Ces problèmes sont orthogonaux à l'intégration du concept de tâches dans les modèles de composants. C'est pourquoi le modèle proposé dans ce chapitre ne les considère pas. A contrario, la propriété *task-safe* est considérée dans le modèle, car COMET introduit des mécanismes fortement liés au parallélisme exhibant ainsi des problèmes de composition de codes parallèle à l'échelle du modèle.

3.4.3 Choix et ajustement automatique de paramètres

Dans un assemblage COMET, certaines propriétés telles que le type des buffers de données, le type de partitionnement, les paramètres de partitionnement, l'expression des dépendances ou la présence et la réutilisation de données temporaires doivent être choisies explicitement et manuellement. Déterminer les valeurs optimales peut être difficile et fastidieux sur de gros assemblages, car l'optimalité des propriétés dépend généralement de l'ensemble de l'assemblage ainsi que de l'architecture matérielle sous-jacente.

Le choix des valeurs optimales pourrait être automatisé. En effet, le partitionnement optimal des données peut être déduit de l'analyse des dépendances issue de l'expression du langage de relations dans de nombreux cas simples. Cette expression des relations peut elle-même être déduite de l'analyse automatique d'un code séquentiel (parallélisation automatique d'un code séquentiel). Ce réglage automatique des propriétés peut être réalisé par un modèle de plus haut niveau. L'objectif du modèle COMET est donc de permettre d'exprimer ces choix avant tout. La sélection automatique des choix, peut être réalisée dans un second temps.

3.5 Conclusion

Dans ce chapitre, nous avons décrit le modèle COMET. Tout d'abord, nous avons défini les unités de composition qui constituent le modèle tels que les composants et métatâches. Puis, nous avons spécifié comment les composer correctement pour former un assemblage d'une application.

La mise au point du modèle COMET montre la faisabilité de l'approche. Néanmoins, nous avons pu voir qu'il existe plusieurs points du modèle à améliorer, tels que le support de la dynamicité, l'expressivité du modèle, la sélection automatique de choix de bas niveau.

Le chapitre suivant présente une mise en œuvre du modèle COMET appelée HALLEY. Il décrit le fonctionnement de HALLEY, fournit des informations complémentaires à COMET telles que des exemples de type de données, de partitionnements, de repartitionnement et apporte des précisions sur les langages de relations mis en place.

Chapitre 4

Halley : une mise en œuvre de Comet

Contents

Ce chapitre décrit les méthodes permettant de réaliser une mise en œuvre efficace et extensible conforme au modèle Comet présenté dans le chapitre 3. Plus précisément, il présente Halley, une mise en œuvre de Comet reposant simultanément sur le modèle de composants L^{2}C et le modèle de directives OpenMP.

Il est important de noter que Halley est un prototype. Son rôle est de montrer qu'il est possible de fournir une mise en œuvre extensible, efficace et conforme au modèle Comet. Ce prototype vise à être suffisamment complet pour pouvoir mettre en œuvre une partie de l'application Gysela (afin de disposer d'un maximum de qualités issues de Comet et analyser leur coût), facilement maintenable (afin de résister à l'évolution de Comet) tout en réduisant le temps de développement nécessaire à son élaboration (prototype de recherche).

Ce chapitre présente tout d'abord la façon dont fonctionne Halley : ses objectifs et son fonctionnement basé sur une compilation source à source et un système de greffon extensible. Puis, il décrit le fonctionnement de la compilation source à source exploitant des méthodes de transformation de modèles de l'ingénierie dirigée par les modèles. Il décrit ensuite comment Halley permet d'implémenter des types de données. Puis, le chapitre détaille comment il est possible définir des types de données partitionnées et comment effectuer des opérations de repartitionnement efficace sur-mesure. Enfin, il présente comment fonctionnent des langages de relations et comment en concevoir de nouveaux avant de conclure.

4.1 Fonctionnement

Cette section détaille la manière dont Halley fonctionne et présente son architecture. Elle donne tout d'abord les objectifs remplis par Halley et les méthodes utilisées pour atteindre ces objectifs. Puis, la section présente l'ingénierie dirigée par les modèles, une méthode de conception logicielle centrée sur la notion de modèle et de transformation de modèles utilisée afin de concevoir Halley. Enfin, la section détaille comment cette approche est utilisée dans Halley.

4.1.1 Objectifs

Un des objectifs de Halley est de spécifier comment décrire des composants, des métatâches, des assemblages conformes au modèle Comet. Cela signifie qu'il détaille l'ensemble des informations requises pour décrire une application, mais aussi la manière d'exprimer ces informations (c.-à-d. expression des informations à travers un langage).

De plus, Halley vise aussi à fournir tous les éléments nécessaires à l'exécution d'une application dans le but de former un package déployable à l'exécution. Cela inclut par exemple un support exécutif minimaliste utile pour paramétrer le support exécutif de graphes de tâche sous-jacent (OpenMP).

Enfin, Halley a aussi pour objectif de spécifier comment décrire des greffons. Un greffon est une extension du modèle Comet permettant de définir un type de données, un type de données partitionnées, une opération de repartitionnement ou encore un langage de relations. En effet, rappelons que Comet suppose l'existence de greffons prédéfinis référencés par un identifiant dans le modèle, mais qu'il n'a pas pour objectif de spécifier comment les décrire. Cette charge est donc déléguée à la mise en œuvre. L'ajout de greffons supplémentaires dans Halley est une tâche dédiée à des utilisateurs experts, car elle requiert des connaissances avancées et diverses : architecture des greffons, parallélisme, utilisation des tâches avec OpenMP et une connaissance profonde de l'architecture de Halley pour l'implémentation de nouveaux langages de relations.

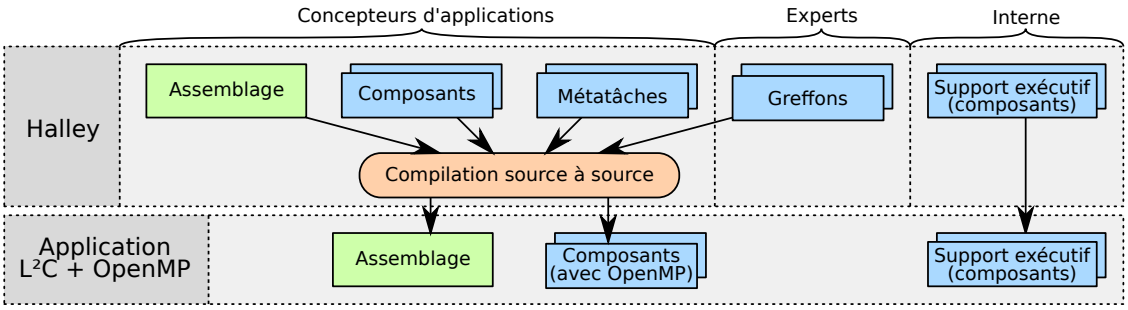

FIGURE 4.1 – Principe de fonctionnement de HALLEY visant à produire une application L^{2}C à partir de sources conformes au modèle COMET, de greffons ainsi que d'un support exécutif minimaliste

4.1.2 Fonctionnement général

À partir d'une description d'un assemblage, d'un ensemble de composants, de métatâches fournies par les concepteurs d'une application, HALLEY effectue une étape de compilation source à source qui a pour effet de générer un assemblage L^{2}C ainsi que des composants L^{2}C dont certains utilisent des directives OpenMP. Par exemple, les sections de l'assemblage et les types de métatâches sont responsables de la génération de composants L^{2}C utilisant des directives OpenMP. Durant cette étape de transformation, HALLEY peut utiliser des greffons définis par des experts, généralement indépendants de l'application considérée. Les greffons sont donc en pratique regroupés dans des dépôts séparés du code d'une application. L'assemblage et les composants L^{2}C résultant de cette étape de compilation sont combinés à des composants du support exécutif de HALLEY pour former une application L^{2}C complète et exécutable indépendamment. La figure 4.1 illustre comment fonctionne ce procédé de manière générale.

Dans l'implémentation actuelle de HALLEY, on distingue deux catégories de greffons : d'une part ceux décrivant un type de données (partitionnées ou non) ou un repartitionnement et d'autre part ceux spécifiant un langage de relations.

Les greffons de la première catégorie sont des composants spécifiques qui doivent posséder des interfaces particulières en lien avec les rôles qu'ils remplissent (p. ex. les greffons définissant des types de données partitionnées doivent fournir des interfaces fournissant des services permettant de partitionner des données). Une présentation plus détaillée de ces greffons est donnée dans les sections 4.3 et 4.4.

Ces greffons définissant un langage de relations sont particuliers, car ils ont un impact sur la génération des composants. Ils interviennent donc directement durant le processus de compilation source à source. Les raisons de ce choix et la nature de ces greffons sont décrites en section 4.5. La section suivante présente plus en détail le processus de compilation.

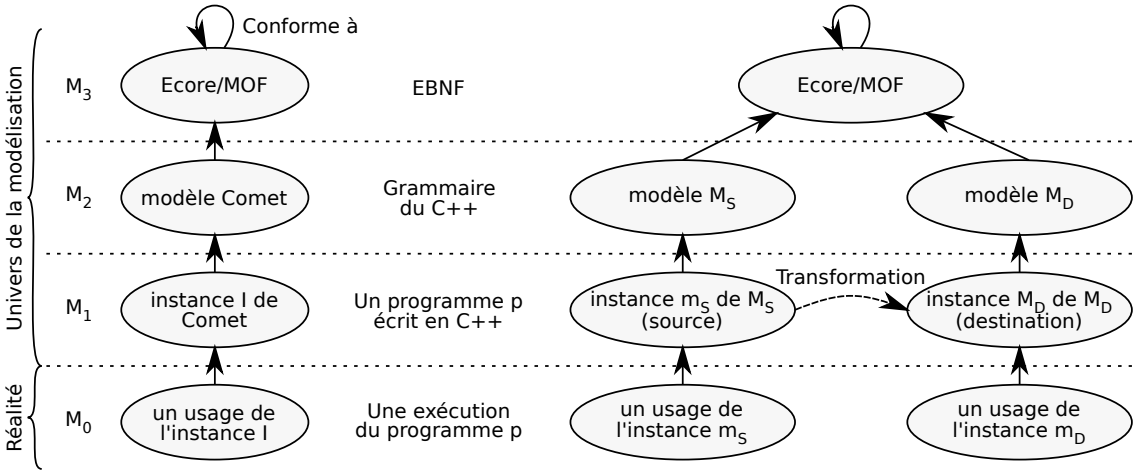

FIGURE 4.2 – Hiérarchie des modèles à gauche, celle des langages au milieu et illustration d'une transformation de modèle à droite.

4.2 Compilation source à source

Afin de décrire le processus de compilation de HALLEY, cette section introduit tout d'abord le fonctionnement de l'ingénierie dirigée par les modèles sur laquelle cette mise en œuvre repose. Elle décrit ensuite en détail les éléments nécessaires à la compilation et résultants de ce processus ainsi que les méthodes de transformation utilisées.

4.2.1 Ingénierie dirigée par les modèles

L'ingénierie dirigée par les modèles [67, 17, 49] (IDM) est une forme d'ingénierie basée sur la génération et la transformation de modèles dans laquelle tout ou une partie d'une application est engendré par des modèles. L'IDM se distingue ainsi des autres formes de programmation en se concentrant principalement sur les abstractions permettant de décrire les fonctionnements d'une application. Un modèle est une abstraction (ou représentation) d'un système destiné à un objectif et permettant de répondre à des problématiques relatives au système considéré. Plusieurs modèles peuvent être utilisés pour décrire un même système afin de répondre à des questions différentes. L'IDM sollicite l'utilisation de plusieurs langages spécifiques interdépendants "sur-mesure" visant à modéliser chacun une partie d'une application.

Cette approche s'intéresse plus particulièrement à la conception et la manipulation des modèles. Un modèle $\mathcal{M}_1$ peut être décrit par un langage $\mathcal{L}_1$ (c.-à-d. sérialisation textuelle ou graphique). Un modèle peut être conforme à un modèle $\mathcal{M}_2$ plus abstrait : son métamodèle. On dit alors que le modèle $\mathcal{M}_1$ est conforme au métamodèle $\mathcal{M}_2$. Il est aussi possible de dire que $\mathcal{M}_1$ est une instance de $\mathcal{M}_2$.

Tout comme un modèle, un métamodèle peut être conforme à un autre métamodèle plus abstrait (parfois appelé métamétamodèle). Par récurrence, il est possible de construire une hiérarchie de modèles opérant à différents niveaux d'abstraction.

Certains métamodèles possèdent un niveau d'abstraction suffisamment élevé pour permettre de se définir eux-mêmes. C'est par exemple le cas du métamodèle MOF défini par l'OMG ou encore de la variante Ecore de la fondation Eclipse. La manipulation de modèles appartenant à un même niveau d'abstraction se fait par une étape de transformation de modèle pouvant être elle-même décrite par un modèle.

La conception d'un logiciel avec cette approche consiste à mettre au point des métamodèles adaptés, puis à les utiliser afin d'établir des modèles de l'application conformes aux métamodèles. Des langages tels que Emfatic issue du métamodèle Ecore peuvent être utilisés comme outils pour décrire un modèle/métamodèle (que celui-ci soit spécifique à une application ou non).

La figure 4.2 illustre une hiérarchie de modèle appliqué au modèle COMET, un exemple de hiérarchie de langages associés (indépendant de COMET) et une illustration du fonctionnement de la transformation entre deux modèles.

4.2.2 Métamodèle source et métamodèle de destination

L'étape de compilation source à source de HALLEY est assimilable à une transformation de modèle : des instances d'un métamodèle source (c.-à-d. modèle source décrivant une application) sont transformées en instance d'un métamodèle de destination. Avant de présenter le fonctionnement de la transformation, il est nécessaire de bien comprendre les métamodèles d'entrée et de sortie de la transformation (c.-à-d. source et destination).

Métamodèle source

La spécification du modèle source (métamodèle) de HALLEY est fournie en annexes B.1 (page 161). Le langage utilisé pour décrire le modèle est Emfatic. Le document décrit uniquement les entités et les relations du modèle sans spécifier de règle de validité des éléments ou de contrainte sur les relations (p. ex. validité des connexions). Néanmoins, les règles de validité ou contraintes peuvent être déduites de celle de COMET, définies en section 3.3.5.

Il est important de noter que la description Emfatic n'est pas restreinte uniquement à HALLEY et peut servir de base pour réaliser d'autres mises en œuvre. En effet, la description du modèle source est indépendante du modèle destination, c'est-à-dire du modèle de composants ainsi que du support exécutif de graphes de tâche sous-jacents utilisés, ainsi que du fonctionnement de la transformation de modèle.

Afin de pouvoir décrire l'architecture d'une application à travers HALLEY, les instances du métamodèle source (constituées d'assemblages, de composants, etc.) doivent être sérialisées en utilisant un langage. Pour cela, nous avons choisi d'utiliser XML, un langage de balisage permettant de formater des données textuellement (p. ex. instance d'un métamodèle). Les documents XML peuvent être manipulés simplement et leur structure peut être spécifiée à l'aide d'un schéma (XSD), aussi utilisable pour valider un document. Les schémas des documents peuvent, dans notre cas, être facilement déduits du métamodèle source. Cette dernière fonctionnalité est

particulièrement utile pour vérifier syntaxiquement les documents XML en entrée de la compilation source à source.

La description d'une application est constituée des éléments suivants :

— un fichier XML décrivant un assemblage (schéma XML en annexe B.5 à la page 172) ;
— des fichiers XML définissant des métatâches (schéma XML en annexe B.3 à la page 170) ;
— des fichiers XML décrivant des informations relatives aux greffons, telles que la liste des greffons disponibles spécifiant leur nom, leur type et une référence vers leurs implémentations (schéma XML en annexe B.4 à la page 171).

Métamodèle de destination

Le métamodèle de destination de HALLEY est tout simplement celui de la mise en œuvre en C++ de L^{2}C fournie en annexe B.2 (page 166), étendu pour considérer le contenu du code source des composants pouvant faire usage du modèle de directive OpenMP, en particulier les constructions à base de tâches. La définition de schéma XML décrivant la structure des assemblages est fournie en annexe B.6 (page 174). La structure des assemblages et celles du code source des composants sont déterminées par la transformation de modèle.

4.2.3 Transformation de modèles

Fonctionnement général

La transformation de modèles, c'est-à-dire la transformation des modèles conformes au métamodèle source en modèles conforme au métamodèle destination, est une opération réalisée par un compilateur source à source écrit en Python. Elle est décomposée en quatre phases distinctes :

1. décodage des fichiers d'entrées dans le but de construire une représentation du modèle source ;
2. analyse sémantique visant à construire un modèle intermédiaire qui attache à chaque information du modèle source des métadonnées (p. ex. propagation de types et de métadonnées), utiles entre autres pour valider l'assemblage et effectuer une analyse globale des sections de l'assemblage analysé (p. ex. pour déterminer un ordonnancement statique des métatâches à déclancher) ;
3. transformation visant construire une représentation du modèle de destination à partir du modèle intermédiaire ;
4. encodage de la représentation du modèle de destination en fichiers XML/C++.

Les phases 1 et 4 consistent respectivement à lire un fichier XML de manière à reconstruire une instance du modèle source en mémoire et écrire des fichiers XML à partir du modèle de destination ainsi que des codes C++ à partir de fichiers de templates du compilateur. Les phases triviales (1 et 4) ne sont donc pas détaillés dans

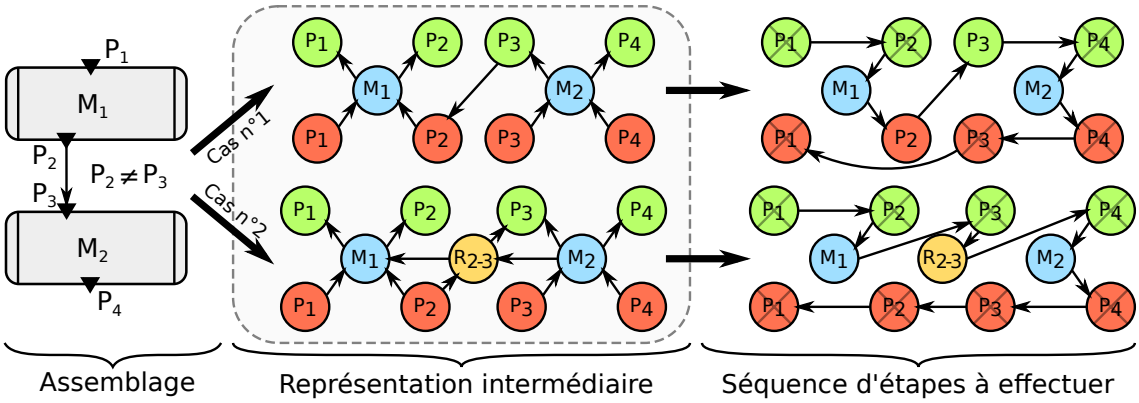

FIGURE 4.3 – Fonctionnement de l'ordonnancement des opérations à effectuer pour une section, sans greffon de repartitionnement dédié (cas n°1, en haut) et avec greffon de repartitionnement dédié (cas n°2, en bas) : une section de l'assemblage d'entrée (à gauche) avec deux métatâches M_1 et M_2 disposant de ports dont les types de données partitionnées sont notés P_i, est transformée en un graphe de dépendances (centre) afin de produire une séquence en utilisant un tri topologique du graphe inverse (à droite). Les nœuds verts, jaunes et rouges représentent respectivement des opérations de partitionnement, repartitionnement et départitionnement. Les nœuds bleus représentent les soumissions de tâches des métatâches. Les cercles barrés décrivent les opérations de partitionnement ou départitionnement qui ne créent pas de dépendance à l'exécution.

la suite de cette section, contrairement aux phases 2 et 3. La phase 3 se décompose en deux parties : la génération d'un assemblage et celle des composants.

Analyse sémantique

L'analyse sémantique est une phase de compilation permettant de réaliser des traitements non locaux sur l'assemblage source (p. ex. traitements globaux aux sections) dans le but d'extraire des informations non triviales (qui ne sont pas directement exprimées dans l'assemblage source) afin de simplifier la phase suivante. Elle effectue les traitements suivants : propagation d'informations dans l'assemblage, production d'une séquence d'opérations à réaliser pour chaque section, énumération des greffons à utiliser pour chaque section et vérification globale de l'assemblage. Le traitement de chaque section est indépendant.

Le premier traitement consiste à propager des informations associées aux buffers de données et au partitionnement à travers les connexions de données. Il permet par exemple de déterminer la présence de buffers de données temporaires ou permanentes ou encore les attributs des types de buffers de données partitionnées ou non partitionnées sur chaque port de l'assemblage source.

Le second traitement a pour objectif de produire une séquence (ordre total) d'opérations qui doivent être effectuées à l'exécution. Chaque opération consiste à soumettre un groupe de tâches à l'exécution. Il existe quatre types d'opérations :

partitionnement, départitionnement, repartitionnement et soumission des tâches de métatâches (le fonctionnement de ces opérations est décrit en détail dans les sections 4.4 et 4.5). Ce traitement effectué par le compilateur source à source peut être assimilé à un ordonnancement statique des métatâches d'une section. Pour réaliser ce traitement, l'analyse sémantique calcule pour chaque métatâche d'une section $\mathcal{S}$, l'ensemble des opérations de partitionnement, de départitionnement et de repartitionnement qui doivent être réalisées avant la soumission des tâches d'une métatâche. L'analyse sémantique calcule ensuite un graphe de dépendances (à gros grain) de toutes les opérations à réaliser à partir de ces informations et en prenant en compte les connexions entre les métatâches de la section $\mathcal{S}$. Ensuite, un tri topologique est appliqué sur l'inverse du graphe de dépendances produisant ainsi un ordre total des opérations à effectuer pour la section $\mathcal{S}$. De plus, ce traitement détermine pour chaque opération de partitionnement ou départitionnement s'il est nécessaire de créer les dépendances à l'exécution entre les données partitionnées et non partitionnées (c.-à-d. dépendance entre un buffer de données et ses fragments). Cette dernière opération est décrite plus en détail en section 4.4. La figure 4.3 illustre le fonctionnement de ce traitement sur un exemple simple avec et sans l'utilisation de greffons de repartitionnement.

Le troisième traitement consiste à choisir l'ensemble des greffons à utiliser dans l'assemblage de destination à générer. Ce traitement est expliqué dans la section suivante.

Enfin, le dernier traitement consiste à vérifier la sémantique de l'assemblage (p. ex. compatibilité des connexions de données). Ce traitement analyse la conformité de l'assemblage d'entrée au modèle COMET et plus précisément la validité des règles énoncées en section 3.3.5. En pratique, l'implémentation actuelle de HALLEY ne réalise qu'une vérification partielle de l'assemblage afin de limiter les erreurs les plus fréquentes et les plus dangereuses.

Production de l'assemblage de destination

Fonctionnement général La transformation de l'assemblage source en un assemblage de destination consiste fondamentalement à transformer chacune des sections du modèle source en un assemblage de composants. Les composants issus de l'assemblage de destination peuvent être classés en quatre catégories :

- les *composants utilisateur*, quasiment inchangés par rapport à l'assemblage source à l'exception des ports de données (transformés en ports *use-provide*) ;
- les *composants glus*, générés à partir des sections et métatâches de l'assemblage source ;
- les *composants experts*, issus des greffons (type de données, type de données partitionnées et repartitionnement) ;
- les *composants nécessaires pour le support exécutif*, visant principalement à configurer OpenMP.

Structure de l'assemblage La figure 4.4 présente la structure générique d'une portion d'assemblage correspondant à la transformation d'une unique section $\mathcal{S}$ ap-

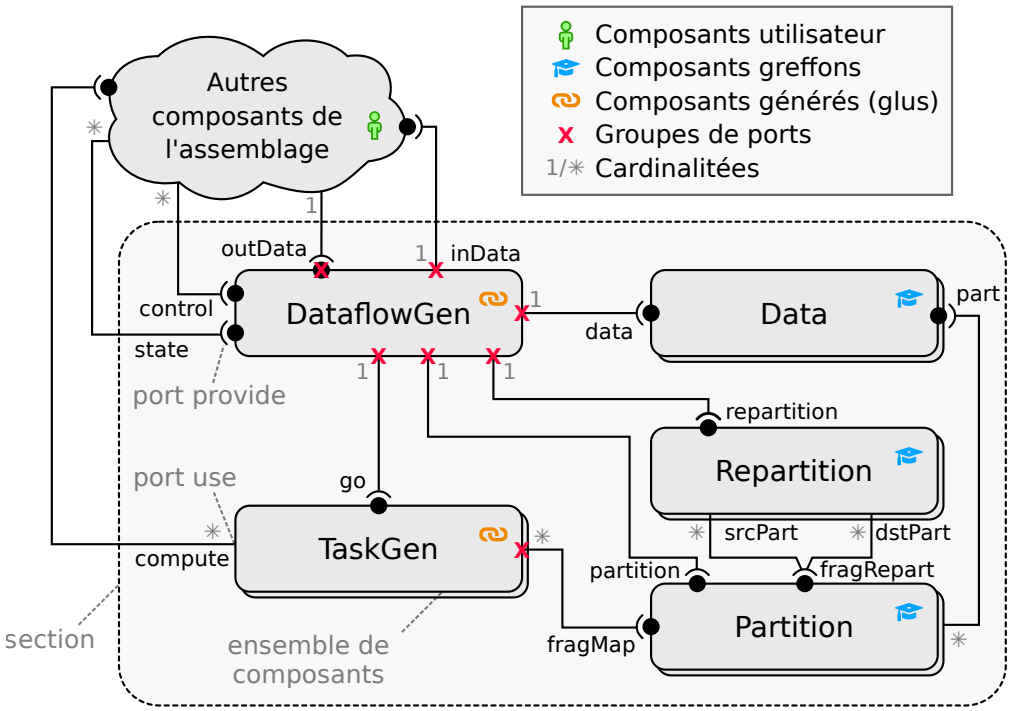

FIGURE 4.4 – Structure générique de l'assemblage de destination généré par la transformation de modèles.

partenant au modèle source. On peut observer que $\mathcal{S}$ est transformée en un assemblage partiel constitué de quatre groupes d'instance de composants.

Types d'instances de composants `DataflowGen` référence une unique instance alors que `TaskGen`, `Data`, `Partition`, `Repartition` font chacun référence à des groupes d'instances de composants de types multiples. Analysons plus précisément le rôle de chacun des groupes.

`DataflowGen` désigne une instance d'un type de composant généré à partir de la section $\mathcal{S}$. Le rôle de cette instance est de contrôler l'ensemble des instances issues de la section $\mathcal{S}$: elle est responsable de l'interaction entre les groupes d'instances `Data` par le biais des ports `data`, `Partition` à travers les ports `partition`, `Repartition` par le biais des ports `repartition`, `TaskGen` à travers les ports `go`, ainsi que les instances de composants utilisateurs par le biais des ports `control`, `state`, `inData` et `outData`.

`TaskGen` désigne des instances de composants dont les types sont générés à partir des métatâches de $\mathcal{S}$. Ces instances ont pour objectif de produire un groupe de tâches à la demande du port `go`. Les tâches exécutent toutes une méthode d'une interface de `compute` à partir d'un ensemble fragments de données partitionnées reçues depuis `fragMap`. Le fonctionnement de ces instances est présenté plus précisément en section 4.5.

`Data` désigne des instances de composants issues des données de $\mathcal{S}$. Les types de

ces instances sont définis par des greffons responsables de la gestion de données. Ils exposent un port `data` permettant de configurer un buffer de données et gérer son cycle de vie, ainsi qu'un port `part` utilisé pour contrôler la cohérence des dépendances lors de partitionnement ou repartitionnement. Ces greffons sont présentés plus précisément en section 4.3.

`Partition` désigne des instances de composants provenant de types de données partitionnées présents sur les ports des métatâches de $\mathcal{S}$. Les types de ces instances sont définis par des greffons responsables de la gestion de données partitionnées. Ils exposent un port `partition` qui permet de partitionner ou départitionner un buffer de données, c'est-à-dire de gérer un type de données partitionnées. Pour cela, ils interagissent avec les greffons de gestion de données par le port `part`. Le partitionnement génère à l'exécution un ensemble de fragments accessibles par les ports `fragMap` et `fragRepart`. Ces greffons sont présentés plus précisément en section 4.4.

`Repartition` désigne des instances de composants issues de connexion de données de $\mathcal{S}$ dont la sémantique engendre une opération de repartitionnement (c.-à-d. connexion de ports détenant des types de données partitionnées différents). Les types de ces instances sont définis par des greffons responsables du repartitionnement de données partitionnées. Ils exposent un port `repartition` qui permet de passer d'un buffer de données partitionnées à un autre. Pour cela, ces types de composants interagissent avec les greffons gérant des types de données partitionnées à travers les ports `srcPart` et `dstPart`. Ces greffons sont présentés plus précisément en section 4.4.

Connexions et cardinalités Les groupes de ports de la figure 4.4 sont des ensembles de ports appartenant à un même service générique. Ils sont générés en fonction de la section $\mathcal{S}$ (p. ex. leur nombre dépend du nombre de données, de repartitionnements nécessaires, du nombre des connexions de données traversant la section, etc.). Les cardinalités indiquent, pour chaque connexion générique, le nombre de ports *use* connectés à un même port *provide* d'un unique composant : 1 signifie qu'il y a un unique port *use* et $*$ signifie qu'il y a un nombre quelconque de ports *use* (0 inclus). Lorsqu'une connexion générique comporte un groupe de port, la cardinalité de la connexion s'applique à chaque port du groupe. Par exemple, dans le cas de la connexion entre `DataflowGen` et `Data`, `DataflowGen` possède autant de ports dédiés à la gestion des données qu'il y a d'instances dans le groupe `Data` et chaque port est connecté à une unique instance du groupe `Data`. Dans des cas plus complexes, tels que la connexion entre les groupes `TaskGen` et `Partition`, il est plus difficile de déterminer quelles sont les connexions issues de la connexion générique : chaque instance du groupe `Partition` peut être connectée à plusieurs instances du groupe `TaskGen` et chaque instance du groupe `TaskGen` possède un nombre variable de ports chacun connecté à des instances différentes du groupe `Partition`.

Les instances de chaque groupe et leurs connexions sont fixées par le processus de transformation à partir des résultats de l'analyse sémantique issus de la section $\mathcal{S}$. Dans le cas du groupe `Data`, le nombre d'instances est le nombre de buffers de données (temporaires ou non) présents dans la section $\mathcal{S}$.

Pour chaque instance du groupe `Data`, le type du composant (c.-à-d. le choix du

greffon) est déterminé à partir du type de données présent sur les ports. Le lien entre les deux se fait à l'aide des fichiers d'informations relatives aux greffons : à chaque type de données est associé un ensemble d'implémentation (greffons) supportant le type de données. Lorsque plusieurs implémentations sont disponibles pour un même type (c.-à-d. conflit), elles peuvent être discriminées selon la présence des attributs détenus par les buffers de données de la section $\mathcal{S}$. Cependant, afin de simplifier la mise en œuvre, HALLEY fait actuellement un choix arbitraire : il sélectionne la première implémentation compatible définie dans les fichiers des greffons sans analyser la compatibilité des attributs.

Le choix du nombre d'instances du groupe `Partition` et la sélection de leurs types fonctionnent de la même manière que pour les instances du groupe `Data` excepté qu'il y a autant d'instances que de types de données partitionnés présents sur les ports de la section $\mathcal{S}$.

Le même fonctionnement s'applique aussi au groupe `Repartition`. Cependant, le nombre d'instances est fixé par le nombre de connexions de ports détenant des types de données partitionnées différents. De plus, la sélection de types de composants se fait en recherchant un greffon capable d'effectuer un repartitionnement entre deux types. Lorsqu'il n'y a pas de greffons disponibles, les types de données partitionnées sont départitionnés par un greffon et partitionnés par un autre.

Enfin, dans le cas du groupe `TaskGen`, il y a autant d'instances que de métatâches dans la section $\mathcal{S}$. Chaque type de métatâche de la section $\mathcal{S}$ engendre un type de composant associé. Le type de chaque instance du groupe `TaskGen` est fixé par le type de l'instance de métatâche correspondant dans la section $\mathcal{S}$.

Configuration des instances Il est important de noter que toutes les instances de composants, à l'exception de `DataflowGen`, possèdent des attributs (des valeurs constantes de types primitifs issues de l'assemblage source), bien que la figure 4.4 les omette. La présence de ces attributs est fortement dépendante de l'assemblage source et des greffons considérés, mais n'a que très peu d'impact sur la compréhension de l'architecture de l'assemblage ou sur sa transformation.

Génération de composants glus

Dans cette section, nous nous intéressons particulièrement à la manière dont sont générés les composants et plus particulièrement au composant `DataflowGen` de la section $\mathcal{S}$. Le fonctionnement des composants `TaskGen` est traité dans la section 4.5 dédiée.

Tous les composants glus sont générés à partir du moteur de template *Mustache*[1]. Ce moteur de template est exempt de flot de contrôle (boucles, condition). Il permet de générer le code d'un composant à partir d'un fichier de template et d'un modèle de données correspondant.

L'implémentation du composant `DataflowGen` est séparée en trois préoccupations majeures : l'interaction avec les composants utilisateurs, le contrôle/récupération

1. https://mustache.github.io

```
interface DataPort<Datatype>
{
    void setBuffer(Datatype* data);
    void wait();
}
```

FIGURE 4.5 – Définition générique d'une interface de données.

```
interface ControlPort
{
    void start();
    void run();
    void synchronize();
}
```

```
interface StatePort
{
    // Possible values:
    // IDLE | RUNNING | WAITING
    State state();
}
```

FIGURE 4.6 – Définition des interfaces de contrôle (gauche) et d'état (droite).

de l'état de la section ainsi que l'exécution de cette section.

L'interaction avec les composants consiste à échanger des descripteurs de buffer de données avec des composants utilisateurs. Pour cela, la section détient un ensemble de ports de données. Les interfaces de données respectent le type générique décrit par la figure 4.5. La méthode `setBuffer` permet de transmettre un descripteur de buffer de données d'un composant utilisateur vers la section. L'implémentation de la section stocke temporairement ce descripteur dans le but de le réutiliser lors de l'exécution de la section. Après cet appel, le buffer de données appartient à la section et ne doit pas être manipulé par le composant. La méthode `wait` permet au composant utilisateur d'attendre que cette donnée soit à nouveau disponible et donc manipulable par le composant. En pratique, l'implémentation rend disponible les données que lorsque toutes les tâches d'une section se sont terminées.

Les interfaces de contrôle et d'état d'une section respectent les types décrits par la figure 4.6. La méthode `start` déclenche la soumission des tâches issues d'une section sans attendre leur terminaison (c.-à-d. comportement asynchrone). Elle doit être suivie par un appel à la méthode `synchronize` qui attend que toutes les tâches issues de la section soient terminées (débloquant ainsi tout appel à la méthode `wait` dans l'implémentation actuelle). Ces deux méthodes modifient l'état interne d'une section. Le comportement de l'état interne est décrit par la figure 3.4 présentée au chapitre 3. Toutefois, l'implémentation actuelle considère l'état `submitting` comme l'état `running` (OpenMP 4.5 ne permet pas de forcer la soumission d'une tâche sans permettre son exécution). La méthode `run` déclenche la soumission des tâches issues d'une section et attend leur terminaison (c.-à-d. comportement synchrone). Il est important de noter que l'implémentation actuelle ne permet pas d'exécuter simultanément des méthodes du port de contrôle dans des contextes parallèles différents (c.-à-d. méthodes du port de contrôle non thread-safe). L'état interne peut être sondé par d'autres composants par le biais du port d'état des sections.

```
1  pour chaque port use p issu d'un buffer de données b de 𝒮 faire
2      si b est un buffer de données temporaire alors
3          Générer: p.allocate();
4      sinon
5          Générer: p.acquire(dataDescriptor(i));
6      fin
7  fin
8  pour chaque port use p issu d'un (re)partitionnement de 𝒮 faire
9      Générer: p.prepare();
10 fin
11 Générer: #pragma omp taskgroup
12 début
13     pour chaque opération o de l'ordonnancement des opérations à
         effectuer à partir de 𝒮 faire
14         si o est un partitionnement alors
15             port ← port use connecté à l'instance du greffon responsable de o
                 ;
16             carryDependencies ← dependencyInfos(o);
17             Générer: port.partition(carryDependencies) ;
18         sinon si o est un départitionnement alors
19             port ← port use connecté à l'instance du greffon responsable de o
                 ;
20             carryDependencies ← dependencyInfos(o);
21             Générer: port.unpartition(carryDependencies) ;
22         sinon si o est un repartitionnement alors
23             port ← port use connecté à l'instance du greffon responsable de o
                 ;
24             Générer: port.repartition();
25         sinon si o est une métatâche m à exécuter alors
26             port ← port use connecté à l'instance issue de m ;
27             Générer: port.go();
28     fin
29 fin
   // Ici : synchronisation implicite due à l'annotation taskgroup
30 pour chaque port use port issu d'un (re)partitionnement de 𝒮 faire
31     Générer: port.clean();
32 fin
33 pour chaque port use port issu de buffer de données b de 𝒮 faire
34     si b est un buffer de données temporaire alors
35         Générer: port.free();
36     sinon
37         Générer: port.release();
38     fin
39 fin
```

Algorithme 1 : Pseudo-code du template des composants de section générés. Les opérations d'ordonnancement sont celles issues de l'analyse sémantique illustrée par la figure 4.3.

Le code déclenchant la soumission des tâches (réalisée par les méthodes `start` et `run`) est généré à partir d'informations issues de l'analyse sémantique de la section $\mathcal{S}$ en suivant l'algorithme 1 (l'algorithme présente uniquement le cas synchrone mis en place par la méthode `run` ; dans le cas asynchrone, l'ensemble du code jusqu'à la synchronisation implicite est encapsulé dans une tâche OpenMP). L'algorithme se décompose en cinq grandes parties. Il commence par récupérer des données permanentes depuis les ports de données d'entrées/sorties et allouer des données temporaires. Ensuite, il déclenche la préparation des partitionnements et repartitionnements. Après, il exécute la séquence d'opérations calculée lors de l'analyse sémantique (dont le fonctionnement est décrit par la figure 4.3) en alternant des opérations de partitionnement, de soumission de tâches de métatâche, de départitionnement et de repartitionnement. L'exécution de ces opérations fait appel à des services fournis par les greffons présentés dans les sections 4.3 et 4.4 ainsi qu'à des services fournis par des composants générés depuis des métatâches dont le fonctionnement est décrit en section 4.5. Enfin, l'algorithme demande le nettoyage des structures de données issues des opérations relatives aux partitionnements avant de déclencher la libération des données.

Transformation des composants utilisateurs

Les sources des composants utilisateurs sont composées de deux parties distinctes : le code contenant l'implémentation du composant qui prend la forme de classes C++ et la définition des interfaces d'un composants, exprimé en utilisant des macros C++. La transformation de composants Halley en composants L^{2}C se fait automatiquement lors de la compilation : les macros de Halley génèrent automatiquement à la compilation les macros L^{2}C responsables de la description des interfaces d'un composant. Cela a pour effet de transformer les ports de données en ports *use* dont le type est décrit par la figure 4.5. La transformation des composants utilisateur n'a donc aucun impact sur l'implémentation même d'un composant et s'effectue de manière totalement transparente.

4.3 Support des types de données

Cette section présente comment des experts peuvent définir des types de données dans Halley en utilisant des greffons dédiés. Elle définit les interfaces que les greffons doivent posséder ainsi que les liens entre les greffons et les codes utilisateurs.

La définition d'un greffon se fait en deux parties distinctes : la spécification d'un composant et son enregistrement. Un greffon est associé à un composant visant à gérer des buffers de données, disposant de ports bien définis. Chaque instance de ce composant est responsable d'un unique buffer de données manipulable à travers les interfaces du composant. L'enregistrement consiste à définir ce composant en tant qu'implémentation d'un type de données bien défini, utilisable ensuite dans les applications par des utilisateurs quelconques.

```
interface ExtDataPort<DataType>
{
    void acquire(DataType* ptr);
    void release();

    void allocate();
    void free();

    void wait();
};
```

FIGURE 4.7 – Type de port `data` des greffons de données (interface publique).

Étant donné que HALLEY est un prototype, il est important de noter que les interfaces des greffons ne couvrent pas le cas général, mais ces interfaces sont suffisamment généralistes pour traiter les cas d'utilisations traités dans les chapitres 5 et 6. Des travaux supplémentaires sont nécessaires afin d'améliorer la généricité des interfaces ainsi que pour évaluer leur applicabilité sur d'autres cas d'utilisations plus variés. Cette remarque s'applique aussi aux autres sortes de greffons présentés dans les sections 4.4 et 4.5.

4.3.1 Interfaces des greffons

Les greffons de données doivent disposer d'au moins deux ports *provide* : un port de gestion des données `data` et un port `part` relatif au partitionnement du buffer de données comme cela a été précédemment illustré sur la figure 4.4. Détaillons le contenu de ces deux interfaces.

Port d'accès et de manipulation des données Le port `data` d'un greffon de données vise à permettre la manipulation d'un buffer de données par la section. Le type de cette interface est décrit par la figure 4.7.

Les méthodes `acquire` et `release` sont utilisées pour gérer des buffers de données permanents (c.-à-d. obtenus depuis des composants utilisateurs). La méthode `acquire` permet au composant de récupérer le descripteur d'un buffer de données (p. ex. pointeur). Cette méthode rend le greffon responsable du buffer de données. Les métadonnées relatives au buffer de données (p. ex. taille du buffer) ne sont pas passées en paramètres et sont définies par des attributs du composant. La méthode `release` relâche la responsabilité du buffer de données.

Les méthodes `allocate` et `free` sont utilisées pour gérer des buffers de données temporaires (c.-à-d. visibles uniquement dans les sections et n'existant que lors de leurs exécutions). Dans le cas des buffers de données temporaires, les greffons de données sont toujours responsables de leur buffer. La méthode `allocate` crée un buffer de données temporaire non initialisé à partir des métadonnées fournies par les attributs du composant. La méthode `free` détruit le buffer temporaire en mémoire.

```
interface ExtDataPartPort<DataType, MetaInformations>
{
    void* getDependency();
    DataType* data();
    MetaInformations infos();

    // Autres méthodes supplémentaires requises
    // par les greffons de partitionnement
}
```

FIGURE 4.8 – Type de port `part` des greffons de données (interface privée).

La méthode `wait` est une fonction bloquant le fils d'exécution de l'appelant jusqu'à ce que toutes les tâches qui opèrent sur le buffer de données soient terminées.

Port de partitionnement Le port `part` d'un greffon de données vise à fournir des informations aux greffons de partitionnement. Les greffons de données sont libres de fixer leurs propres types d'interface pour ce port, en accord avec les experts fournissant des types de données partitionnées associés. En effet, le type de l'interface `part` ne dépend que des greffons de données et des greffons de partitionnement associés et non des composants glus de HALLEY. HALLEY propose néanmoins un patron de type d'interface qui permet aux greffons de partitionnement de fonctionner (indépendamment du type de données traité). Ce type est décrit par la figure 4.8.

La méthode `getDependency` fournit un pointeur opaque utilisé pour définir des dépendances entre un buffer de données et les fragments issus de son partitionnement, assurant ainsi la cohérence des calculs effectués par le graphe de tâche généré à l'exécution, notamment en cas de repartitionnement. La méthode `data` fournit un descripteur référençant le buffer de données géré par le greffon. La méthode `infos` fournit les métadonnées du buffer de données, gérées par le greffon, qui sont nécessaires au partitionnement (p. ex. taille de la donnée). Plus d'informations sur l'usage de cette interface par les greffons de partitionnement sont fournies en section 4.4.

Spécification des interfaces La figure 4.9 montre un exemple de définition de type de données (tableau bidimensionnel) utilisé pour déduire les interfaces du greffon associé. `elemSize` est utilisé pour définir la taille des éléments du tableau (p. ex. *float*, *double*, *int*, etc.). Il est important de noter que le type `DataType` est aussi utilisé pour décrire le type des ports de données, permettant à HALLEY de déduire automatiquement les ports *use*/*provide* issue des ports de données.

4.3.2 Enregistrement de greffons et paramétrage

Afin qu'une application puisse accéder à un greffon, celui-ci doit être associé à un type de données et répertorié dans des fichiers de description de greffons fournis au compilateur HALLEY. La figure 4.10 montre un exemple de fichier (défini

```
namespace Array2D
{
    struct DataType
    {
        void* ptr;
    }

    struct MetaInformations
    {
        int32_t elemSize; // sizeof
        int64_t width, height;
    }
}
```

FIGURE 4.9 – Exemple de définition d'un type de données pour les tableaux bidimensionnels.

```
<dataType id="Array2D">
    <parameter id="width" type="int64"/>
    <parameter id="height" type="int64"/>
    <parameter id="elemSize" type="int32"/>
</dataType>

<dataImpl id="Plugin_Array2D" type="Array2D"/>
```

FIGURE 4.10 – Exemple de définition d'un type de données `Array2D` pour les tableaux bidimensionnels et d'un greffon `Plugin_Array2D` supportant ce type.

par le schéma XML en annexe B.4 à la page 171) décrivant un type de tableaux bidimensionnels avec trois attributs `width`, `height` et `elemSize` correspondant aux métadonnées contenues dans la structure `MetaInformations` de la figure 4.9, ainsi que le nom d'un greffon supportant ce type de données. Le nom du greffon correspond au nom du composant responsable de l'implémentation du type. Plusieurs greffons peuvent implémenter un même type de données.

4.4 Support des données partitionnées et repartitionnement

Cette section présente comment des experts peuvent définir des types de données partitionnés et des opérations de repartitionnement de données partitionnées dans HALLEY en utilisant respectivement deux sortes de greffons dédiés. Elle définit dans un premier temps les interfaces que les greffons doivent posséder, ainsi que les liens entre les greffons et les codes utilisateurs. Elle décrit ensuite comment HALLEY gère

```
interface ExtPartitionPort
{
    void prepare();
    void partition(bool carryDepencencies);
    void unpartition(bool carryDepencencies);
    void clean();
};
```

FIGURE 4.11 – Type de port `partition` permettant de contrôler le partitionnement des greffons de données partitionnées.

les dépendances entre les types de données partitionnées et non partitionnées de telle manière à assurer une exécution cohérente des tâches issues des métatâches (c.-à-d. conforme à l'assemblage des métatâches).

4.4.1 Interfaces des greffons

Greffons de données partitionnées

Les greffons de données partitionnées visent à diviser (virtuellement) un buffer de données en un ensemble de fragments. Un fragment est une instance d'un type de données *POD*[2] opaque défini par le greffon.

Les greffons de données partitionnées doivent disposer d'au moins trois ports, illustrés sur la figure 4.4 : un port *provide* `partition` visant à contrôler le partitionnement d'un buffer de données, un port *use* `part` permettant de récupérer des informations nécessaires au partitionnement et au départitionnement d'un buffer de données ainsi que des ports *provide* `fragMap` et `fragRepart` dont l'objectif est de mettre à disposition les fragments du buffer de données partitionnées dans le but de soumettre des tâches opérant sur les fragments ou encore d'effectuer une opération de repartitionnement entre des données partitionnées. Détaillons le contenu de ces trois interfaces.

Port de contrôle du partitionnement La figure 4.11 décrit le type de l'interface `partition`. La méthode `prepare` prépare le partitionnement du buffer de données géré par le composant. Cette méthode sert en pratique à initialiser ou allouer des données nécessaires au partitionnement ou encore à récupérer toutes les informations requises au partitionnement lui-même, notamment par le biais du port `part`. La méthode `partition` déclenche une opération de partitionnement. Le buffer de données partitionné est celui qui est accessible par le port `part`. Le partitionnement d'un buffer de données engendre des fragments de données (accessibles par le port `fragInfo`. Lorsque le paramètre `carryDepencencies` est `vrai`, la méthode `partition` génère des dépendances entre le buffer de données global et chacun des

2. Les types *Plain Old Data* (POD) sont spécifiés par la norme C++. Ces types n'utilisent pas de fonctionnalités orientées objet (p. ex. constructeurs, destructeurs, membres virtuels).

```
interface ExtFragMapPort<Fragment, FragmentId>
{
    size_t fragmentCount();
    Fragment getFragment(size_t fragId);
    void* getDependency(size_t fragId);
}
```

FIGURE 4.12 – Type de port `fragMap` minimaliste nécessaire au greffon définissant la relation d'identité.

fragments. Dans le cas contraire, aucune dépendance n'est produite. Plus de détails sont fournis en section 4.4.2. La méthode `unpartition` déclenche une opération de départitionnement. Après cette opération, les fragments ne sont plus accessibles. La méthode `clean` libère les données éventuellement allouées par la méthode `prepare`.

Port d'accès aux fragments Le type de l'interface `fragMap` n'est pas déterminé par HALLEY, mais par les greffons décrivant des langages de relations. En effet, ces greffons exploitent l'interface `fragMap` dans le but de produire des tâches respectant une expression décrite dans les métatâches. Afin d'avoir une idée de l'utilité de ce port, un exemple de type d'interface minimaliste nécessaire au greffon de relations `identity` (définissant la relation d'identité) est présenté sur la figure 4.12. Plus généralement, ce type peut dépendre d'un type de fragment défini par le greffon de données partitionnées et d'un type définissant un identifiant de fragment (permettant de retrouver un fragment dans la partition). Dans le cas de la relation identité, ce dernier n'est pas nécessaire. Une explication plus détaillée est fournie dans la section 4.5. La méthode `fragmentCount` fournit le nombre de fragments produits par l'opération de partitionnement. La méthode `getFragment` permet de récupérer un fragment à partir du numéro du fragment. Il est important de noter que la relation identité suppose que l'ensemble des fragments est ordonné selon des critères décrits par chaque greffon de données partitionnées. La méthode `getDependency` permet de récupérer un pointeur opaque à partir du numéro du fragment. Ce pointeur permet de définir des dépendances entre des fragments et des tâches ou même entre des fragments afin de pouvoir réaliser des opérations de repartitionnement.

Port de repartitionnement Tout comme le port `fragMap`, le type de l'interface `fragRepart` n'est pas contraint par HALLEY, mais par les greffons de partitionnement et de repartitionnement. En effet, l'interface `fragRepart` fournir l'ensemble des informations nécessaires aux greffons de repartitionnement afin qu'ils puissent effectuer des opérations de repartitionnement. Les experts de ces deux sortes de greffons doivent donc s'accorder sur la définition d'une interface adaptée. HALLEY propose une base d'interface minimaliste utile pour concevoir une interface dédiée en l'absence d'un accord précédemment établi. Cette interface est décrite par la figure 4.13. Les méthodes `fragmentCount`, `getFragment` et `getDependency` sont

```
interface ExtFragRepartPort<Fragment, FragmentId, MetaInformations>
{
    size_t fragmentCount();
    Fragment getFragment(FragmentId fragId);
    void* getDependency(FragmentId fragId);
    MetaInformations infos();
}
```

FIGURE 4.13 – Type de port `fragRepart` minimaliste nécessaire au greffon définissant la relation d'identité.

identiques à l'exemple d'interface `fragMap` excepté qu'elles opèrent sur des identifiants de fragments. L'interface proposée ajoute une méthode `infos` permettant de récupérer des attributs du greffon de données partitionnées (p. ex. taille du buffer, caractéristiques générales des fragments, etc.).

Spécification des interfaces La figure 4.14 montre un exemple de structures de données visant à définir un buffer de données de type `Array2D` partitionné en blocs. Ces structures de données sont exposées par le greffon de données et permettent aussi de déduire le type des ports du greffon.

Greffons de repartitionnement

Les greffons de repartitionnement doivent disposer d'au moins trois ports, illustrés sur la figure 4.4 : un port *provide* `repartition` visant à contrôler le repartitionnement d'un buffer de données partitionnées et deux ports *use* `srcPart` et `dstPart` permettant de récupérer des informations des partitionnements source et destination pour effectuer des opérations de repartitionnement.

La figure 4.15 décrit le type de l'interface `repartition`. Les méthodes `prepare` et `clean` sont identiques à celles de l'interface 4.11. La méthode `repartition` a pour effet de réaliser une opération de repartitionnement. Une telle opération vise à définir des dépendances entre les fragments issus du partitionnement référencé par le port `srcPart` et ceux qui sont issus du partitionnement référencé par le port `dstPart`. Le fonctionnement des opérations de partitionnement et de repartitionnement est décrit dans la section 4.4.2.

Le type des ports `srcPart` et `dstPart` est identique aux ports `fragMap` des greffons de données partitionnées. Néanmoins, le type du port `srcPart` peut être différent du type de `dstPart` étant donné qu'un greffon peut effectuer un repartitionnement entre des types de données partitionnées différents (les deux doivent cependant appartenir au même type de buffer de données).

```
namespace BlockedArray2D
{
    struct Fragment
    {
        void* ptr; // pointer to the top left origin of the block
        size_t width, height; // size of the block
        size_t ld; // leading dimension (offset between two lines)
    }

    struct FragmentId
    {
        size_t x, y; // position of the block in the block grid
    }

    struct MetaInformations
    {
        size_t blockWidth, blockHeight;
    }
}
```

FIGURE 4.14 – Structures de données permettant de définir un buffer de données de type `Array2D` partitionné en blocs.

```
interface ExtRepartitionPort
{
    void prepare();
    void repartition();
    void clean();
};
```

FIGURE 4.15 – Type de port `repartition` permettant de contrôler les greffons repartitionnement.

4.4.2 Mécanismes de cohérence

Cette section décrit le fonctionnement de l'implémentation des greffons de données partitionnées et de repartitionnement (c.-à-d. comment utilisent-il OpenMP 4.5 afin de garantir la cohérence de l'exécution des tâches issues des métatâches et quelles sont les méthodes que les greffons mettent en place pour qu'ils puissent fonctionner ensemble).

Partitionnement

Le partitionnement consiste à créer un groupe de fragments à partir d'un buffer de données et soumettre des tâches de telle manière à créer des dépendances entre le buffer complet et chacun des fragments. Pour que cela fonctionne, un greffon de données doit fournir un unique pointeur (dépendance) b, référençant le buffer de données complet, par le biais de l'interface `part`. Ce pointeur doit être différent de tous les pointeurs f_i (dépendances), référençant chacun des fragments accessibles par le biais des ports `fragMap` et `fragRepart` des greffons de données partitionnées. De même, chaque fragment d'une partition doit être associé à un unique pointeur dans la partition. Dans le cas contraire, les stratégies de partitionnement proposées dans cette section engendrent une exécution séquentielle des tâches de partitionnement et ainsi créent une synchronisation artificielle.

La méthode la plus simple pour effectuer une opération de partitionnement consiste à soumettre une unique tâche de partitionnement avec b en lecture et tous les f_i en écriture[3]. Similairement, une opération de départitionnement consiste à soumettre une unique tâche de partitionnement avec tous les f_i en lecture-écriture et b en écriture. Ainsi, les tâches opérant sur les fragments issus du nouveau partitionnement ne peuvent s'exécuter qu'après la tâche de partitionnement. Cette tâche de partitionnement ne s'exécutera qu'après toutes les tâches de départitionnement réalisées sur le buffer précédemment. À leur tour, les tâches de départitionnement ne peuvent s'exécuter qu'après toutes les tâches précédemment soumises opérant sur des fragments du buffer avant son départitionnement.

Cette méthode à l'avantage d'être simple, mais ne peut pas être utilisée en pratique. En effet, OpenMP 4.5 ne permet pas de fournir un nombre variable de dépendances à l'exécution (étant donné que la liste des dépendances doit être fournie à la compilation à travers des annotations). De plus, la soumission de tâches avec un grand nombre de dépendances peut avoir un effet néfaste à l'exécution étant donné qu'OpenMP n'empêche pas les supports exécutifs d'utiliser des algorithmes d'ordonnancement avec une complexité non linéaire (p. ex. quadratique) en nombre de dépendances par tâche (ou simplement en nombre de tâches). Par conséquent, il est important d'adapter la méthode de manière à soumettre un nombre raisonnable de tâches de partitionnement et de dépendances par tâche (cela s'applique aussi aux tâches issues des métatâches). Le choix judicieux de la quantité de tâche et de dépendances par tâche dépend du support exécutif.

3. Le mode écriture est équivalent au mode lecture-écriture en OpenMP 4.5.

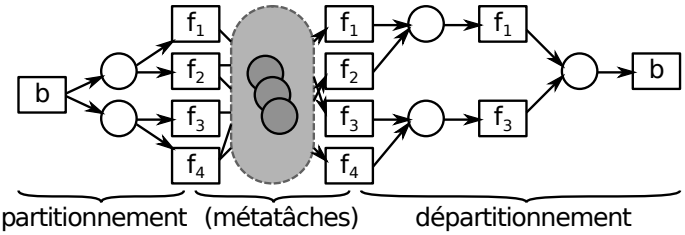

FIGURE 4.16 – Principe de fonctionnement du partitionnement et du départitionnement. Les cercles représentent des tâches et les rectangles des données (c.-à-d. buffers ou fragments). Un maximum de deux dépendances par tâches de partitionnement est utilisé pour la réduction dans cet exemple.

La méthode actuellement utilisée pour effectuer une opération de partitionnement consiste à soumettre $\lceil \frac{n}{k} \rceil$ tâches avec comme dépendances b en lecture et f_i en écriture, où n est le nombre de fragments de la partition et k une constante arbitraire représentant le nombre de dépendances maximal par tâche à soumettre (en pratique fixée à $n = 8$ dans la mise en œuvre). La méthode utilisée pour le départitionnement est similaire, mais elle utilise plutôt une réduction pour éviter l'exécution séquentielle des tâches.

La figure 4.16 fournit un exemple d'opérations de partitionnement et de départitionnement actuellement utilisées par les greffons de données partitionnées.

Repartitionnement

Un repartitionnement est une opération visant à créer des dépendances entre deux groupes de fragments provenant respectivement de partitionnements source et de destination issus d'un même buffer de données. Cela suppose donc qu'un même buffer de données est précédemment partitionné selon simultanément deux partitionnements différents (avec un type de partitionnement éventuellement différent). De plus, cela présume aussi que chacun des fragments (provenant des greffons de données partitionnées) doit être associé à un unique pointeur (c.-à-d. les pointeurs issus de tous les fragments de tous les partitionnements possibles d'un même buffer de données doivent être différents).

Pour effectuer une opération de repartitionnement, un greffon soumet des tâches qui lisent les fragments du partitionnement source et écrivent dans les fragments du partitionnement de destination. Dans les cas les plus simples, le greffon calcule, pour chaque fragment du partitionnement de destination, les fragments du partitionnement source dont il dépend.

Dans les cas plus complexes où les fragments de destination partagent des dépendances vers un même groupe de fragments sources, cette méthode peut être inefficace. Il est alors plus efficace de soumettre des tâches écrivant dans plusieurs fragments de destinations (groupe de fragment disjoints). C'est par exemple le cas pour le repartitionnement du buffer de données de type `Array2D` partitionné en blocs réguliers (p. ex. adaptation de la taille des blocs de 64x32 en 32x64).

Dans les cas les plus complexes, un greffon peut avoir recours à des dépendances

```
<partitionType id="BlockedArray2D" datatype="Array2D">
    <parameter id="blockWidth" type="int64"/>
    <parameter id="blockHeight" type="int64"/>
</partitionType>

<partitionImpl id="Plugin_BlockedArray2D" type="BlockedArray2D"/>
<repartitionImpl id="Plugin_RepartBlockedArray2D"
        srcType="BlockedArray2D" dstType="BlockedArray2D"/>
```

FIGURE 4.17 – Exemple de définition d'un type de données partitionnées `BlockArray2D` pour les buffers de type `Array2D` partitionnés en blocs réguliers et d'un greffon `Plugin_BlockedArray2D` supportant ce type de données ainsi qu'un greffon de repartitionnement pour ce type.

temporaires (internes) dans le but de réduire les coûts associés au nombre de dépendances à l'exécution au risque d'accroître le nombre de tâches, de manière analogue aux opérations de départitionnement. Cette méthode est particulièrement utile pour des partitionnements non réguliers.

4.4.3 Enregistrement des greffons et paramétrage

Tout comme les greffons de données, les greffons de données partitionnées peuvent être associés à des types de données, eux-mêmes associés à des types de données afin que ces greffons soient rendus disponibles aux utilisateurs. Il en est de même pour les greffons de repartitionnement qui sont quant à eux associés à deux types de données partitionnées (source et destination). La figure 4.17 montre un exemple de fichier (défini par le schéma XML en annexe B.4 à la page 171) décrivant un type de buffer `Array2D` (décrit précédemment sur la figure 4.9) partitionné en blocs réguliers avec deux attributs `width`, `height` correspondant aux métadonnées contenues dans la structure `MetaInformations` de la figure 4.14. Elle montre aussi le nom d'un greffon supportant ce type de données ainsi que le nom d'un greffon de repartitionnement associé. Le nom du greffon correspond au nom du composant responsable de l'implémentation du type ou du repartitionnement. Plusieurs greffons peuvent implémenter un même type de données ou une même sorte d'opération de repartitionnement.

4.5 Langages de relations

Cette section s'intéresse aux greffons permettant de décrire des langages de relations et à la transformation des métatâches en composants glus. Contrairement aux autres greffons, ceux qui définissent les relations sont des extensions du compilateur de HALLEY. Ils sont partiellement responsables de la génération des composants glus issus des métatâches. Leur mise en œuvre s'adresse donc à des personnes ex-

périmentées ayant une bonne compréhension de l'architecture du compilateur ainsi que des connaissances poussées du fonctionnement des supports exécutifs (usage d'OpenMP). Cependant, à terme, il est envisageable de relayer la définition des greffons de relations à des composants externes au compilateur, tout comme les greffons de partitionnement par exemple.

L'objectif de cette section est de montrer comment les greffons de relations fonctionnent et comment ils s'intègrent dans le processus de compilation d'Halley afin de générer des composants glus issus des métatâches (indépendamment des détails de mise en œuvre).

4.5.1 Fonctionnement général et intégration

Halley ne fixe que peu de contraintes sur les greffons de relations qui peuvent être de nature très diverse. Pour bien comprendre le fonctionnement des greffons de relations et leurs liens avec d'autres sortes de greffons, cette section analyse un exemple de prototype de greffon `align` décrivant des *stencils* (c.-à-d. traitement appliquant un calcul dépendant du voisinage pour chaque point d'un maillage) simples.

Un greffon de relations a pour objectif principal de définir un langage de relations. C'est-à-dire qu'il spécifie la syntaxe et la sémantique d'un langage dont le rôle est de décrire comment regrouper des fragments issus de buffers de données partitionnées ensemble afin de soumettre des tâches qui opèrent sur ces fragments. Ces greffons sont majoritairement responsables de la génération des composants glus issus des métatâches lors de la compilation.

Un greffon de relations est constitué d'une spécification en deux parties et d'une mise en œuvre. Une première partie de la spécification, dédiée aux concepteurs d'applications, comporte : un langage de relations lié à un identifiant et un type d'interface pour les ports `compute` des métatâches utilisant ce langage. Une seconde partie de la spécification, dédiée aux experts définissant les greffons de données partitionnées, contient un type d'interface pour les ports `fragMap`. La mise en œuvre est constituée conjointement d'une classe C++ et d'un fichier de template intégrés au compilateur.

Étant donné que le fonctionnement des greffons dépend des types de métatâche traités, cette section utilisera l'exemple de métatâche décrit par la figure 4.18 comme fils conducteur. Cette métatâche permet d'effectuer un stencil cinq points. Elle dispose d'un port d'entrée `in` et d'un port de sortie `out`, tous les deux de type `BlockedArray2D` et configurés pour traiter des buffers de tailles carrés (paramétrés par `dSize`) avec des blocs de tailles carrés (paramétrés par `bSize`). Le contenu de la configuration de la métatâche est un code Python, exécuté à la compilation lors de la phase d'analyse sémantique, définissant les valeurs des attributs des greffons de données (partitionnées ou non) associés à chaque port à partir des attributs de métatâche (accessible par la variable `mt`). Les attributs XML `dataPartitionSetter` et `dataBufferSetter` font référence à des fonctions de la configuration.

```
<metatask id="StencilMt">
    <parameter id="dSize" type="uint64"/>
    <parameter id="bSize" type="uint64"/>
    <inPort id="in" type="BlockedArray2D" dataBufferSetter="fData"
                                          dataPartitionSetter="fPart"/>
    <outPort id="out" type="BlockedArray2D" dataBufferSetter="fData"
                                            dataPartitionSetter="fPart"/>
    <relation language="align">
        out(x,y) = in(x-1,y), in(x,y), in(x+1,y), in(x,y-1), in(x,y+1)
    </relation>
    <setting language="python3">
        def fData(mt):
            return dict(width=mt.dSize, height=mt.dSize, elemSize=8)
        def fPart(mt):
            return dict(blockWidth=mt.bSize, blockHeight=mt.bSize)
    </setting>
</metatask>
```

FIGURE 4.18 – Exemple de type de métatâche utilisant le langage de relations `align`.

4.5.2 Langages de relations

La figure 4.19 présente la grammaire du langage de relations `align`. Ce langage permet de décrire à gauche de l'expression des fragments dans lesquels les tâches écrivent et à droite les fragments lus (ou des deux côtés pour les fragments en lecture-écriture). Chaque occurrence de `fragId` fait référence à un nom de port de la métatâche (plus précisément il fait référence aux buffers de données partitionnées transmis par le port). Toutes les occurrences de `dimAccess` doivent posséder des `dimId` consécutifs identiques (c.-à-d. pas de transposition possible). Les opérations arithmétiques des membres de droite permettent d'accéder aux fragments voisins (c.-à-d. nombre de fragments). Les buffers de données référencés par les membres de gauche doivent posséder des caractéristiques identiques (c.-à-d. dimension, taille, partitionnement).

La métatâche décrite par la figure 4.18 contient un exemple d'expression acceptée par langage de relations `align`. Dans cet exemple, pour chaque fragment du port `out` indexé par (x, y), on soumet une tâche disposant de cinq fragments, décrits par la partie droite de l'expression, appartenant tous au même port `in`. Les positions des fragments à droite de l'expression sont relatives aux fragments à gauche de l'expression.

4.5.3 Interfaces

Pour fonctionner, un composant glu issu d'une métatâche doit posséder au moins trois ports : un port `go` possédant une seule méthode `go` (sans paramètres) permettant de soumettre des tâches (dont le greffon de relations est responsable), un port

```
integer = ? regular expression: [0-9]+ ?;
fragId = ? regular expression: [a-zA-Z_][a-zA-Z0-9_]+ ?;
dimId = ? regular expression: [xyzwvu] ?;
dimAccess = dimId, [("+"|"-"), integer)];
lterm = fragId, "(", dimId, {",", dimId} ")";
rterm = fragId, "(", dimAccess, {",", dimAccess} ")";
lterms = lterm, {",", lterm};
rterms = rterm, {",", rterm};
expr = lterms, "=", rterms;
```

FIGURE 4.19 – Grammaire EBNF (ISO/IEC 14977 1996) du langage de relations `align`. Les virgules désignent une séquence de parties concaténées. Les crochets dénotent des parties optionnelles. Les accolades indiquent des parties répétables plusieurs fois. Les parenthèses sont des parties prioritaires. Les barres verticales marquent un choix à effectuer parmi plusieurs parties. Les parties entre des points d'interrogation sont des commentaires décrivant une partie de la grammaire.

`compute` visible par les utilisateurs et un port `fragMap` visible par les experts. Les types de ces deux derniers ports doivent être spécifiés par un greffon de relations.

Port de calcul L'exécution de chaque tâche soumise par le port `go` exécute une méthode `compute` de l'interface `compute`. Cette méthode prend en paramètre l'ensemble des fragments associés à la tâche.

La liste des fragments à fournir et la manière de les transmettre à la méthode compute est grandement dépendant du greffon de relations considéré ainsi que de la métatâche. En effet, certains greffons peuvent fournir par exemple un nombre fixe de fragments par tâche alors que d'autre non. Similairement, dans de nombreux cas, chaque fragment est associé à un rôle et doit donc être différentiable des autres. Dans le cas d'un stencil par exemple, la mise en œuvre de la méthode `compute` doit pouvoir différencier les fragments, car des traitements différents peuvent être appliqués sur les fragments en fonction de leur position relative. De plus, il est aussi nécessaire dans le cas des stencils de pouvoir effectuer des traitements particuliers aux conditions limites (fragments aux "bords" des maillages). Enfin, les paramètres fournis, par le biais de cette interface, doivent être des structures de type POD de taille raisonnable (une grande taille peut avoir un effet néfaste sur les performances). Pour toutes ces raisons, HALLEY ne propose pas d'interface généraliste et laisse chaque greffon spécifier un type générique (paramétré par le type de métatâche) d'interface spécialisée. HALLEY spécifie seulement que l'interface `compute` détient une unique méthode `compute` disposant d'un unique paramètre, dont le type, nommée `Fragments`, est généré par le greffon de relations à partir du type de métatâche.

La figure 4.20 montre le type d'interface généré par le greffon de relations `align` à partir du type de la métatâche `StencilMt` de la figure 4.18. Chaque élément de la structure pointe sur un fragment, ou vaut une valeur nulle lorsque le fragment

```
struct Fragments
{
    BlockedArray2D::Fragment* f0_out;
    BlockedArray2D::Fragment* f1_in, f2_in, f3_in,
                                  f4_in, f5_in;
}
```

```
interface StentilMt_ComputePort
{
    void compute(Fragments fragment);
}
```

FIGURE 4.20 – Type du port `compute` du composant glu généré depuis la métatâche `StencilMt` par le greffon de relations `align`.

```
interface ExtFragMapPort<Fragment, FragmentId>
{
    size_t fragmentCount(size_t dimensionId);
    Fragment getFragment(FragmentId fragId);
    void* getDependency(FragmentId fragId);
    bool isValid(FragmentId fragId);
}
```

FIGURE 4.21 – Type de port `fragMap` défini par le greffon de relations `align`.

est en dehors des limites des buffers de données (espace non torique). L'ordre des éléments dans la structure correspond à l'ordre des fragments dans l'expression.

Port d'accès aux fragments Afin de pouvoir énumérer les fragments, ou même d'analyser certaines de leurs propriétés (p. ex. localisation sur les "bords" du maillage dans le cas des stencils), les composants glus générés par les greffons de relations possèdent une interface `fragMap` visant à interagir avec les greffons de partitionnement. Le type de cette interface est intrinsèquement lié à la nature de la relation. Il est donc indépendant du type de métatâche traité. Chaque type d'interface `fragMap` peut être supporté par une classe de greffon de données partitionnées (p. ex. buffers de données multidimensionnels partitionnés en blocs réguliers).

La figure 4.12, présentée en section 4.4.1, fournit le type d'interface `fragMap` défini par le greffon de relations `identity`.

La figure 4.21 montre le type d'interface `fragMap` défini par le greffon de relations `align`. Cette interface a connaissance de la topologie de l'espace des fragments contrairement à celle issue de la relation d'identité. Elle a aussi connaissance de la dimension de l'espace des fragments (connu à la compilation à partir de l'expression de relation et des ports du type de métatâche traité) et du contenu du type `FragmentId`. Pour savoir si un fragment est en dehors des bornes d'un buffer, le

composant glu généré a recours à la méthode `isValid`.

4.5.4 Mise en œuvre et génération de composants glus

La mise en œuvre de greffons de relations est décrit par une classe et un fichier de template dans le compilateur associé à un identifiant (nom du langage de relations). Pour chaque type de métatâche (issu des instances d'un assemblage), le compilateur extrait le nom du langage de relations utilisé. En fonction de l'identifiant, il instancie la classe associée à l'identifiant et lui fournit des informations issues de la phase d'analyse sémantique pour le type de métatâche ciblée. Cela veut dire que les instances de la classe ont accès à l'expression de relation ainsi qu'aux ports de la métatâche (attributs, types de partitionnement, etc.). Le rôle de la classe est d'analyser l'expression afin de produire une structure de données associée au template dans le but de générer la partie fonctionnelle (soumission de tâches et génération des interfaces spécifiques) du code d'un composant glu associé à une métatâche. Le code généré par les greffons est inclus dans un code définissant un composant. Ce dernier contient une partie du code non fonctionnel des composants glus (p. ex. structure du composant) générée par le compilateur à partir un template commun à tous les greffons. Un exemple de composant glu généré est fourni à l'annexe B.7 à la page 176.

4.6 Conclusion

Dans ce chapitre, nous avons analysé la façon dont fonctionne Halley, une mise en œuvre de Comet. À cette fin, nous avons évoqué les objectifs qu'il remplit et présenté son fonctionnement. Puis, nous avons décrit comment fonctionne la compilation source à source, basée sur des méthodes issues de l'ingénierie dirigée par les modèles et rendue extensible par un système de greffons. Ensuite, nous avons décrit comment chaque sorte de greffon fonctionne et quel est le rôle de chaque sorte : des greffons de données, aux greffons de données partitionnés, en passant par les greffons de repartitionnement et les greffons de relations.

Chapitre 5

Évaluation sur des cas d'utilisation synthétiques

Contents

Afin d'évaluer simplement COMET et plus précisément sa mise en œuvre HALLEY, il est essentiel de disposer d'un jeu de cas d'utilisation issus de patrons récurrents dans les applications HPC telles que Gysela et permettant d'effectuer des mesures de performance et relative à la maintenabilité logicielle : *benchmark*. Idéalement, il serait bénéfique d'utiliser un benchmark commun existant afin de pouvoir

comparer plus aisément les approches. Certains benchmarks, tels que le *NAS parallel benchmark* [11] (NPB), *LINPACK* [41], *HPC Challenge* [81] et *HPC Conjugate Gradients* [42] sont des références dans la communauté HPC. Cependant, ils ne permettent pas d'évaluer les propriétés relatives à la conception et la maintenance des applications (p. ex. bénéfices des modèles à composants, notamment pour composition des codes parallèle), car chaque expérience des benchmarks est généralement constituée d'un unique noyau de calcul formant un code concis, atomique et cohérent. L'évaluation du modèle de composants, COMET et plus précisément sa mise en œuvre, HALLEY est donc ardue étant donné qu'il n'existe pas de *benchmark* permettant d'étudier les propriétés des modèles à composants HPC.

Pour résoudre ce problème, ce chapitre réalise une évaluation à partir d'un benchmark synthétique "sur-mesure" dont le contenu est inspiré des applications HPC telles que Gysela ainsi que de benchmarks comme le NPB. Ce chapitre se concentre plus particulièrement sur l'analyse des performances de la composition de codes parallèle utilisant le modèle COMET, ainsi qu'un aperçu des avantages et inconvénients d'une telle méthode sur la conception d'applications HPC. Ce chapitre expose et analyse des comportements complexes qui se manifestent dans des expériences simples. Il est complété par le chapitre 6 qui analyse un cas d'étude réaliste tirée de l'application de production Gysela. Ce dernier fournit une analyse plus développée des considérations relatives à la conception d'application à plus large échelle.

Ce chapitre présente, tout d'abord, les cas d'utilisation utilisés pour évaluation. Ensuite, il traite de l'impact de COMET sur la séparation des préoccupations. Enfin, ce chapitre évalue les performances des cas d'utilisation en comparant des versions utilisant le modèle COMET (plus précisément la mise en œuvre HALLEY) avec des versions utilisant directement OpenMP à travers des codes écrits manuellement (développés spécifiquement pour l'évaluation présente dans ce manuscrit).

5.1 Cas d'utilisation

Aperçu des cas d'utilisation Cette section présente tout d'abord trois cas d'utilisation choisis dans ce chapitre pour évaluer HALLEY : *EP* (pour "embarrassingly parallel"), *ST* (pour "stencil"), et *TRANSP* (pour "transpose-based FFT"). Ils représentent des cas de patrons récurrents pouvant être trouvés dans les applications HPC par exemple disponibles dans le NPB (pour EP et TRANSP) ainsi que dans Gysela (pour EP, ST et TRANSP).

EP est un cas d'utilisation effectuant des opérations indépendantes exécutables massivement en parallèle (embarrassingly parallel). Les opérations appliquées traitent des blocs d'un tableau bidimensionnel (opération mémoire sur-place) et sont limitées par la mémoire (une seule opération flottante est appliquée sur chaque case).

ST est un cas d'utilisation visant à résoudre l'équation de la chaleur par la méthode de Jacobi. C'est un stencil 4-points appliqué sur un tableau bidimensionnel, majoritairement limité par la mémoire. Tout comme pour EP, le tableau est découpé en blocs bidimensionnels. Chaque tâche de ce cas lit cinq blocs d'un buffer et écrit dans un bloc d'un autre buffer.

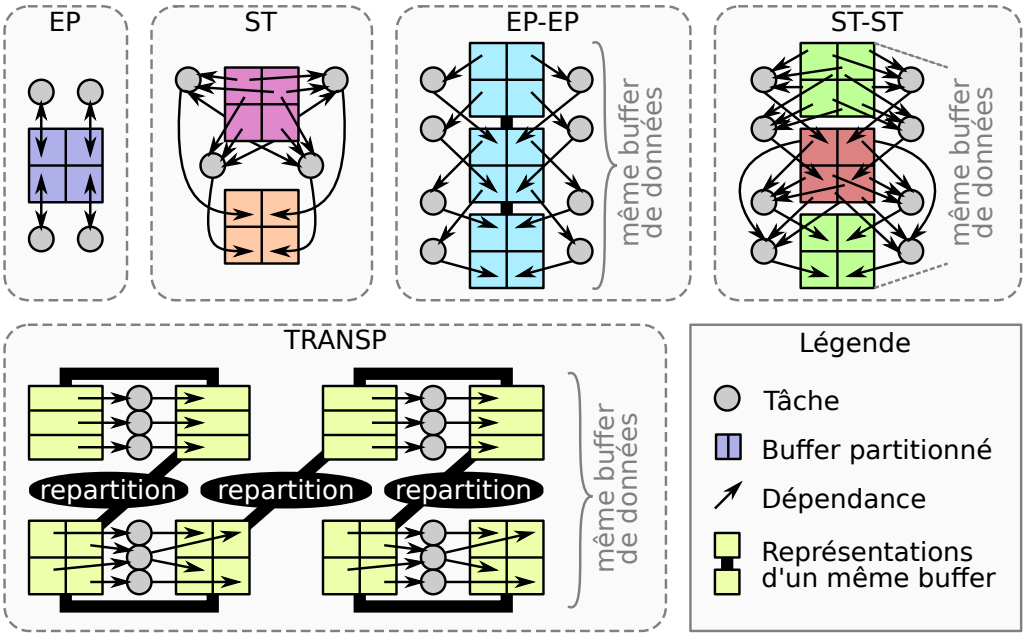

FIGURE 5.1 – Structure des graphes de tâches de chacun des cinq cas d'utilisation. De gauche à droite et de haut en bas : EP, ST, EP-EP, ST-ST, TRANSP.

TRANSP est un cas d'utilisation qui consiste en deux phases de calculs intensifs ligne par ligne d'un tableau bidimensionnel, entrelacées par des transpositions du tableau. L'ensemble des calculs est effectué sur-place en mémoire. Ce patron est assez fréquemment utilisé dans les applications HPC. Par exemple, les algorithmes de *transformé de Fourrier rapide* (FFT) ou encore par des méthodes d'*interpolation à base de splines* ont souvent recours à cette méthode.

Deux autres cas évaluant la composition et l'intérêt du repartitionnement s'ajoutent aux trois premiers : *EP-EP* et *ST-ST*. EP-EP et ST-ST réalisent chacun une exécution consécutive de deux cas EP (et respectivement ST) interdépendants.

La figure 5.1 présente la structure des graphes de tâches des cinq cas d'utilisation. Les opérations de repartitionnement sont affichées explicitement. Tous ces cas d'utilisation utilisent un partitionnement en blocs réguliers (de taille uniforme).

Variantes comparées Pour chaque cas d'utilisation, une *version de référence* est écrite manuellement en OpenMP sans aucun code généré. Elle produit un graphe avec le même nombre de tâches et les mêmes dépendances qu'une version utilisant HALLEY. Le but est d'évaluer le surcoût engendré par HALLEY et d'examiner les bénéfices apportés par COMET lors de la conception des cas d'utilisation.

Assemblages des cas d'utilisation Chaque cas d'utilisation est associé à un assemblage possédant une unique section dataflow. Une exécution d'un cas d'utilisation est constituée de 20 exécutions de la section dataflow (itérations).

L'assemblage HALLEY du cas d'utilisation EP est composé de deux composants (l'un dirige l'assemblage et l'autre fournit une mise en œuvre pour la métatâche de

```
<metatask id="EpMt">
    <parameter id="dWidth" type="uint64"/>
    <parameter id="dHeight" type="uint64"/>
    <parameter id="bWidth" type="uint64"/>
    <parameter id="bHeight" type="uint64"/>
    <inPort id="inout" type="BlockedArray2D" dataBufferSetter="fData"
                                             dataPartitionSetter="fPart"/>
    <relation language="identity"/>
    <setting language="python3">
        def fData(mt):
            return dict(width=mt.dWidth, height=mt.dHeight, elemSize=8)
        def fPart(mt):
            return dict(blockWidth=mt.bWidth, blockHeight=mt.bHeight)
    </setting>
</metatask>
```

FIGURE 5.2 – Type de métatâche utilisé pour le cas d'utilisation EP.

la section) et d'une section contenant une unique instance de métatâche. Le code complet de l'assemblage est fourni en annexe B.8 (page 178). Le type de la métatâche de ce cas d'utilisation est décrit à la figure 5.2. Ce type de métatâche dispose d'un unique port d'entrée-sortie traitant des tableaux bidimensionnels partitionnés en blocs réguliers et a recours au langage de relations `identité`. Chaque tâche opère donc sur un unique fragment et il y a autant de tâches soumises par une métatâche que de blocs dans la partition.

L'assemblage HALLEY du cas d'utilisation ST dispose d'une structure identique à EP, mais les types des composants et des métatâches sont différents (ainsi que les connexions de données). Le code complet de l'assemblage est fourni en annexe B.9 (page 179). La figure 4.18 du chapitre 4 (page 94) fournit une description du type des métatâches du cas d'utilisation ST. Ces métatâches possèdent un port d'entrée et un port de sortie distincts, tous deux acceptent des buffers de données bidimensionnels partitionnés en blocs réguliers et ont recours au langage de relations `align`. Chaque tâche lit le voisinage d'un fragment (cinq fragments) à la position (i, j) issu du port `in` et écrit dans le fragment issu du port `out` qui cette même position (i, j), avec i et j des variables implicites itérant l'espace des fragments. Il y a donc autant de tâches soumises par une métatâche que de fragments du partitionnement du port `out` (identique au nombre de fragments issu du port `in`).

Les assemblages HALLEY respectifs des cas d'utilisation EP-EP et ST-ST sont similaires aux cas EP et ST : dans les deux cas, une nouvelle instance de métatâche est ajoutée et un buffer temporaire est ajouté afin d'assembler les instances de métatâches. Dans le cas d'utilisation ST, un composant dirigeant l'assemblage est remplacé par un autre, car la composition de deux stencils réalisée utilise un buffer permanent et un buffer temporaire plutôt que deux buffers permanents (les interfaces du composant sont différentes et un échange de buffers n'est plus requis dans sa mise

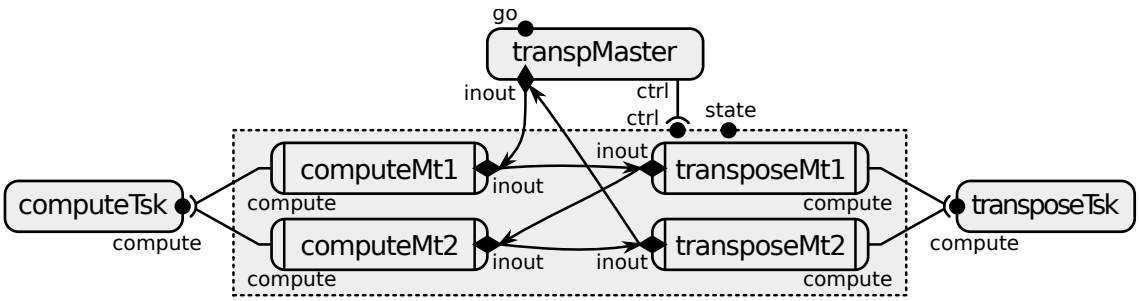

FIGURE 5.3 – Représentation de l'assemblage HALLEY du cas d'utilisation TRANSP.

en œuvre). Les assemblages des cas EP-EP et ST-ST contiennent deux instances de métatâches appelées `Mt1` et `Mt2`. Le code complet de ces assemblages est fourni en annexe B.10 pour EP-EP et en annexe B.11 pour ST-ST (page 180). Le type des deux instances de métatâche présent dans chaque assemblage est le même. Le type des métatâches du cas EP-EP est le même que pour EP. ST-ST est équivalent à EP-EP, mais il emploie des métatâches de ST.

L'assemblage HALLEY du cas d'utilisation TRANSP est décrit par la figure 5.3. Le code complet de l'assemblage est fourni en annexe B.12 (page 182). L'assemblage est constitué d'un composant qui dirige l'exécution du cas d'utilisation, d'une section contenant quatre métatâches et de deux composants fournissant des mises en œuvre pour les métatâches. Ce cas d'utilisation a recours à deux types de métatâches définis à la figure 5.4 : `ComputeMt` qui applique un calcul sur chaque ligne entière (utilisant un partitionnement par blocs de lignes réguliers) et `TransposeMt` qui effectue une transposition du tableau bidimensionnel (utilisant un partitionnement en blocs réguliers). Ces deux métatâches effectuent des opérations sur un unique buffer de donnée. La métatâche `TransposeMt` utilise le langage de relations `transposition`. Celui-ci permet d'appairer des fragments qui se trouvent de chaque côté de la diagonale d'un buffer de données partitionné en blocs réguliers carrés. Pour chaque tâche, une paire de fragments est simultanément lue et écrite : celui à la position (i, j) et celui (j, i) pour tout i et j tel que $i \geqslant j$. Il génère autant de tâches qu'il y a de paires de fragments. Les fragments sur la diagonale n'engendrent qu'une tâche par fragment.

5.2 Discussion sur la séparation des préoccupations

Cette section analyse la capacité de COMET et de HALLEY à offrir des mécanismes permettant de séparer les préoccupations d'une application. Dans un premier temps, elle évalue comment des algorithmes peuvent être scindés et placés dans des unités de composition distincte et dont les mises en œuvre sont indépendantes. Dans un second temps, elle se concentre tout particulièrement sur le fonctionnement et les bénéfices apportés par la composition à base de flux de données disponible dans HALLEY et évalue le degré d'indépendance des codes composés.

```
<metatask id="ComputeMt">
    <parameter id="dWidth" type="uint64"/>
    <parameter id="dHeight" type="uint64"/>
    <parameter id="bHeight" type="uint64"/>
    <inoutPort id="inout" type="BlockArray2D"
               dataBufferSetter="fData2D"
               dataPartitionSetter="fPart2D"/>
    <relation language="identity"/>
    <setting language="python3">
        def fData2D(mt):
            return dict(width=mt.dWidth, height=mt.dHeight, elemSize=8)
        def fPart2D(mt):
            return dict(blockWidth=mt.dWidth, blockHeight=mt.bHeight)
    </setting>
</metatask>

<metatask id="TransposeMt">
    <parameter id="dWidth" type="uint64"/>
    <parameter id="dHeight" type="uint64"/>
    <parameter id="bWidth" type="uint64"/>
    <parameter id="bHeight" type="uint64"/>
    <inoutPort id="inout" type="BlockArray2D"
               dataBufferSetter="fData2D"
               dataPartitionSetter="fPart2D"/>
    <relation language="transposition">
        inout(i,j) = inout(j,i)
    </relation>
    <setting language="python3">
        def fData2D(mt):
            return dict(width=mt.dWidth, height=mt.dHeight, elemSize=8)
        def fPart2D(mt):
            return dict(blockWidth=mt.bWidth, blockHeight=mt.bHeight)
    </setting>
</metatask>
```

FIGURE 5.4 – Types de métatâches utilisés pour le cas d'utilisation TRANSP.

5.2.1 Scission algorithmique dans des unités de composition distinctes

Du point de vue du génie logiciel, le modèle COMET et plus précisément la mise en œuvre, HALLEY permet de définir des morceaux indépendants d'une même application. En effet, étant donné que COMET est un modèle de composants, les unités logicielles telles que les métatâches et les composants ont des interfaces bien définies. Ces interfaces sont les seuls points d'interaction qu'une unité logicielle peut avoir avec les autres. Ainsi, par construction, il est possible de tracer les dépendances entre les unités logicielles d'une même application en analysant les connexions d'un assemblage. Par exemple, dans le cas du cas d'utilisation TRANSP, la métatâche de transposition est indépendante des autres métatâches : la métatâche contient seulement des ports de données et une expression de relations, tandis que le composant responsable de sa mise en œuvre est dépendant uniquement de ces deux dernières informations. Par conséquent, l'adaptation de la méthode de transposition ne nécessite pas de compréhension du fonctionnement interne des unités logicielles. La métatâche de transposition ainsi que sa mise en œuvre peuvent être remplacées par une version alternative possédant, a minima, les mêmes interfaces ou même par une portion d'assemblage compatible. Le remplacement peut être réalisé sans aucune modification de la mise en œuvre des autres composants de l'assemblage. Le même raisonnement est applicable sur les cas d'utilisation EP et ST.

On peut noter toutefois qu'un type de métatâche et ses mises en œuvre sont liés entre eux par une interface associée au greffon de relations choisi par le type de métatâche (à travers le langage de relations). Une modification de l'interface des métatâches (p. ex. changement de partitionnement, ajout d'un port de donnée) peut impacter l'interface du port `compute` des métatâches et donc impacter aussi les composants qui sont responsables de leur mise en œuvre. En effet, l'interface du port `compute` est définie par les greffons de relations. La description d'une interface efficace et suffisamment généraliste pour s'abstraire du partitionnement des ports de données par exemple est un problème ouvert. Lorsque l'interface du port `compute` n'est pas généraliste, le concepteur d'une application peut avoir recours à des composants d'adaptation. Ces composants fournissent un service compatible avec le port `compute` de la métatâche et utilisent un service compatible avec le port fournissant une mise en œuvre pour la métatâche. Une telle opération peut être réalisée dans les cas d'utilisation EP, ST et TRANSP en cas de modification mineure de l'interface des métatâches (p. ex. partitionnement).

Pour résumer, COMET et plus précisément, sa mise en œuvre HALLEY permet d'exprimer différentes parties d'un algorithme dans des unités de composition distinctes assemblées principalement par le biais de flux de données. Par le biais de la composition use-provide (et grâce au port `compute`), COMET permet décrire la mise en œuvre des tâches issues d'une métatâche. Cependant, les métatâches et leur mise en œuvre contenue dans des composants entretiennent un lien fort décrit par l'interface `compute`.

5.2.2 Indépendance des codes composés

Les métatâches issues de COMET peuvent être composées ensemble, malgré l'usage de partitionnement différent. Cela est par exemple visible dans le cas d'utilisation TRANSP. D'une part, les tâches de calcul opèrent sur des blocs de lignes, étant donné que le calcul requiert des données selon une dimension complète. D'autre part, les métatâches de transposition travaillent sur des blocs carrés. Le découpage en bloc permet d'effectuer la transposition en parallèle (mais aussi en même temps que les opérations de calculs) pour être efficace et l'application d'un traitement effectuée sur-place force à utiliser des blocs carrés. Chaque métatâche expose des ports de données partitionnées requis en privilégiant la simplicité et surtout l'efficacité de sa mise en œuvre, indépendamment de l'ensemble de l'application. Elle peut donc être développée et maintenue indépendamment du reste de l'application. La connexion de ces deux types de métatâches (dotées de port de données avec des partitionnements différent) est une opération transparente aux yeux du concepteur d'une application, mais qui peut avoir un impact sur les performances de l'application à l'exécution (pouvant être aussi bénéfique que négatif, p. ex. repartitionnement).

Lorsque deux métatâches composées opèrent sur des buffers partitionnés de différentes manières, les concepteurs d'une application HALLEY font face à un dilemme : considérant chaque métatâche isolée, est-il préférable qu'elle opère sur des partitionnements adaptés à une mise en œuvre efficace, au risque d'effectuer des opérations de repartitionnement potentiellement coûteuses, ou est-il mieux que la métatâche opère directement sur un partitionnement potentiellement inadapté, mais n'impliquant pas de repartitionnement une fois instanciée au sein d'un assemblage ?

Par exemple, dans le cas d'utilisation EP-EP, la taille des blocs peut être adaptée entre les deux codes EP en fonction de la quantité de travail qu'il nécessite de manière à réguler la granularité des tâches (pour permettre une utilisation efficace des ressources tout en modérant les surcoûts liés à la création des tâches). Il en est de même pour ST-ST. Dans le cas d'utilisation TRANSP, plusieurs solutions utilisant le même partitionnement de bout en bout sont possibles. D'une part, les métatâches de transposition peuvent opérer sur des blocs de lignes. Cela a pour effet d'engendrer artificiellement une barrière, car chaque bloc de lignes dépend d'une portion de tous les autres ce qui dégrade fortement les performances (*cf.* impact du repartitionnement évalué en section 5.3). D'autre part, les métatâches de calcul peuvent travailler sur des ensembles de blocs bidimensionnels appartenant à un même bloc de lignes. Ces deux solutions sont néfastes pour la séparation des préoccupations, car le choix des interfaces (c.-à-d. partitionnement) prend en compte le contenu d'un assemblage particulier plutôt que de considérer les entités logicielles de manière isolée et indépendamment d'un usage spécifique. Ainsi, une modification des autres métatâches de l'assemblage telle que le partitionnement a un impact sur la mise en œuvre des autres métatâches. De plus, ces deux cas ne sont pas supportés par les greffons de relations, car il engendre un nombre variable de dépendances par tâche à l'exécution, ce qui n'est pas actuellement supporté par la version actuelle d'OpenMP (version 4.5 et 5.0). Enfin, bien qu'ils n'engendrent pas de repartitionnement (et donc pas de surcoûts liés à cette opération), ces cas augmentent significativement le nombre de

dépendances par tâches et risquent donc d'introduire des surcoûts importants lors de la soumission des tâches. On peut donc en conclure que de telles opérations ont un impact négatif sur la séparation des préoccupations (en rendant la mise en œuvre des composants plus fortement dépendants) et que leur bénéfice en termes de performance est contestable (des recherches complémentaires sont nécessaires concernant l'analyse des différentes solutions sur une large gamme de supports exécutifs supportant notamment un nombre variable de dépendances). Pour cette raison, les cas d'utilisation évalués (ainsi que les greffons de relations mise en œuvre dans Halley) optent pour des partitionnements adaptés à une mise en œuvre efficace de chaque métatâche et ont recours à des repartitionnements.

Il est important de noter que ce dilemme est récurrent lors de la conception des applications HPC, mais qu'il apparaît sous une forme différente. En effet, de nombreuses applications HPC enchaînent des traitements parallèles, soit entrecoupés de barrières (implicites ou explicites), soit optimisés manuellement de manière à ce que l'exécution des parties du traitement soit efficace. Par exemple, une barrière peut être placée entre des codes de calcul intensif et des transpositions. Une autre possibilité est de mélanger les codes de calcul et de transposition, de manière à ce que leur exécution soit entrelacée. Le premier cas peut nuire aux performances des traitements en sur synchronisant les unités de calcul. Le dernier est néfaste pour la séparation des préoccupations due à la présence de plusieurs préoccupations dans un même code dès lors difficile à maintenir. Négliger la séparation des préoccupations entraîne souvent des surcoûts de maintenabilité des applications (c.-à-d. il est plus difficile de supporter de nouvelles méthodes de calcul ou de nouvelles plates-formes).

Comet aide à résoudre ce problème en utilisant des opérations de repartitionnement transparentes : lorsque plusieurs ports de données utilisent le même type de données, mais avec des partitionnements différents, l'implémentation peut utiliser différentes formes de repartitionnement allant de la génération d'une barrière à celle de dépendances entre tâches à grain fin. Par conséquent, les codes peuvent encore être indépendants et leur composition efficace est déléguée à des experts (mise en œuvre dans des composants tiers).

Une barrière peut être injectée par Halley lorsqu'une opération de repartitionnement est nécessaire mais qu'aucune implémentation efficace de repartitionnement permettant une composition à grain fin n'est fournie par des experts, ou bien à la demande de l'utilisateur.

Pour résumer, Halley permet de combiner des codes décrits séparément dont la composition (à base de flux de données) est compatible avec une exécution entrelacée des codes, tout en favorisant l'indépendance des codes à l'aide de repartitionnements.

5.3 Évaluation des performances

Cette section traite de l'évaluation des performances de Halley sur les cinq cas d'utilisation. Elle débute par une description des méthodes et de la configuration expérimentales. Puis, la section évalue le passage à l'échelle en nombre de tâches de l'approche afin de comparer si l'utilisation de Halley introduit des surcoûts lorsque

le nombre de tâches devient imposant. Ensuite, elle étudie les surcoûts introduits par HALLEY lorsque la taille des données est imposante pour une taille de blocs fixe. Après, la section analyse la capacité de HALLEY à combiner des codes efficacement par l'intermédiaire d'une exécution entrelacée, d'une meilleure réutilisation des données en cache ainsi que d'un recouvrement des transferts mémoire par des calculs. Puis, cette section compare les résultats obtenus entre le support exécutif du projet GNU (GOMP) et KOMP afin de déterminer si les résultats observés avec KOMP se généralisent à d'autres supports exécutifs. Enfin, la section discute en détail des résultats avant de conclure ce chapitre.

5.3.1 Méthode et configuration expérimentales

Les performances sont évaluées sur une seule socket la machine de calcul *Brunch* du LIP (Lyon, France). La socket contient un processeur 24 cœurs Intel Xeon E7-8890 v4 (architecture Broadwell, processeur cadencé à 2,20 GHz). Chaque expérience a été réalisée 20 fois et le temps de complétion médian d'une exécution est affiché. Les barres d'erreurs sur les figures sont quasiment invisibles, car les erreurs sont souvent négligeables (inférieure à 1%). Le compilateur utilisé est GNU GCC 6.3, le niveau d'optimisation est configuré au niveau maximum (option `-O3` et `-march=native`). Les assemblages L^2C générés par HALLEY ciblent un code OpenMP respectant la norme 4.5 sans ajout d'extension ainsi que les supports exécutifs présents dans les compilateurs fréquemment utilisés tels que ICC, GCC et Clang. Cependant, nos expériences se concentrent principalement sur les dernières versions du support exécutif KOMP [29] [1] basé sur une version dérivée (c.-à-d. fork) de celui d'Intel (c.-à-d. IOMP). Le choix de ce support exécutif est justifié par des performances généralement meilleures dues à un ordonnancement plus adapté aux cas d'utilisation. De plus, KOMP est codéveloppée dans l'équipe ce qui nous a permis d'avoir un support accru (p. ex. résolution de bugs, amélioration et analyse des performances de cas pathologiques, production de traces permettant de valider les graphes de tâches produits à l'exécution et garantissant la validité de l'ordonnancement). Ce support exécutif inclut notamment des extensions pour fournir un support des codes générés par les compilateurs ICC, GCC et Clang. KOMP fournit aussi un support du vol de travail dont la mise en œuvre est issue de Cilk [50].

Une seule des quatre sockets de la machine *Brunch* est utilisée afin de ne pas subir d'effets NUMA lors des expériences réalisées qui peuvent complexifier l'analyse des résultats expérimentaux. En effet, les architectures ciblées par HALLEY sont des architectures à mémoire partagée avec cohérence de cache qui doivent être supportées par OpenMP. Des extensions à l'interface de programmation d'OpenMP permettant d'exploiter efficacement les architectures NUMA sont disponibles dans de nombreux supports exécutifs de recherche tels que Qthread [92], OmpSs [31] ou celui présenté dans [115] qui décrit le support exécutif KOMP (sélectionné pour nos expérimentations). Cependant, les supports exécutifs actuellement compatibles avec OpenMP (p. ex. GOMP du projet GNU ou IOMP d'Intel) ne disposent pas de ces

1. Code et informations disponibles à l'adresse `http://gitlab.inria.fr/openmp/libkomp`

extensions étant donné qu'il n'y a pas à ce jour d'interface standard dans OpenMP pour les supporter. La mise en place d'une telle interface fait actuellement l'objet de discussion au sein du comité de standardisation d'OpenMP. Il semble raisonnable, dans un futur proche, que les versions ultérieures d'OpenMP (après la version 5.0) disposent de moyens permettant d'exploiter efficacement les architectures NUMA.

5.3.2 Passage à l'échelle du nombre de tâches

Le tableau 5.1 indique le temps de complétion des cas d'utilisation EP et ST avec différentes tailles de blocs et un buffer de données de taille 8192x8192.

Tout d'abord, on observe que le temps de complétion augmente lorsque la taille des blocs diminue. Ceci est dû au surcoût causé par le support exécutif KOMP (utilisé à travers OpenMP) provenant de la gestion des tâches qui devient prépondérant pour un grand nombre de tâches. En effet, lorsque la granularité des tâches est petite, le coût de chaque tâche devient conséquent par rapport à la quantité de calcul à effectuer : avec une taille de bloc 1024x1, chaque tâche s'exécute en approximativement 1,6 µs pour EP et 3,4 µs pour ST, ce qui laisse très peu de temps pour ordonnancer chaque tâche sur l'un des 24 cœurs disponibles tout en respectant les dépendances. Le support exécutif Intel d'origine ou GOMP ont le même comportement, avec un surcoût plus important pour GOMP (*cf.* figure B.1 à la page 183 des annexes). De plus, une analyse détaillée de la trace d'exécution (volumineuse) nous a permis de comprendre que cela est dû à la soumission séquentielle des tâches.

Toutefois, Halley offre des performances similaires à la version OpenMP de référence. Halley est jusqu'à 1,094 fois plus rapide que la version de référence sur le cas d'utilisation EP et 1,037 fois plus lent sur le cas d'utilisation ST. La différence entre les deux versions est d'autant plus grande que le nombre de tâches est important. Dans ces cas, les surcoûts provenant de la gestion des tâches OpenMP deviennent prédominants. Les surcoûts liés à Halley par rapport à version OpenMP de référence sont négligeables (moins de 1%) avec 4096 tâches ou moins (c.-à-d. taille de blocs supérieure ou égale à 1024x16).

Ainsi, on peut conclure que Comet n'introduit pas de surcoûts importants par rapport à la version OpenMP écrite manuellement lorsque la granularité des tâches n'est pas trop petite.

5.3.3 Passage à l'échelle de la taille des buffers de données

Le tableau 5.2 indique le temps par tâche de chaque version (c.-à-d. référence et Halley) et le rapport entre le temps d'exécution de la version Halley et la version de référence sur les cas d'utilisation EP et ST pour différentes tailles de données (de 4096x4096 à 16384x16384) avec une taille de blocs fixée à 1024x128.

La version Halley est aussi rapide que la version de référence correspondante avec un ratio très proche de 1 : Halley n'ajoute pas de surcoût par rapport à la version OpenMP écrite manuellement. De plus, le temps par tâche avec Halley reste stable tout en augmentant la taille du problème et en maintenant le travail par

Y Taille	#Tâches	EP Référence	EP Halley	EP Ratio	SP Référence	SP Halley	SP Ratio
1024	64	0,399	0,399	0,999	0,601	0,603	1,003
512	128	0,401	0,401	0,999	0,607	0,607	0,999
256	256	0,399	0,399	1,000	0,604	0,603	1,000
128	512	0,399	0,399	1,000	0,603	0,605	1,003
64	1024	0,399	0,400	1,001	0,602	0,604	1,003
32	2048	0,401	0,401	1,001	0,604	0,604	1,001
16	4096	0,406	0,403	0,992	0,609	0,612	1,005
8	8192	0,405	0,407	1,006	0,705	0,702	0,996
4	16384	0,590	0,538	0,912	1,156	1,206	1,044
2	32768	1,152	1,071	0,930	2,226	2,336	1,049
1	65536	2,321	2,121	0,914	4,479	4,647	1,038

TABLE 5.1 – Temps de complétion (en secondes) et ratio $\frac{t_{Halley}}{t_{reference}}$ obtenu pour les versions de référence et HALLEY sur les cas d'utilisation EP et ST avec des tailles de blocs variables (1024xY) et un buffer de données de taille 8192x8192. La version de référence correspond à la variante OpenMP écrite manuellement.

Taille	#Tâches	EP μs	EP ratio	ST μs	ST ratio
4096 x 4096	128	42,1	1,0008	60,5	1,0006
8192 x 4096	256	41,7	1,0004	56,5	1,0016
16384 x 8192	1024	41,3	1,0043	59,0	1,0003
16384 x 16384	2048	41,3	1,0001	57,0	1,0032

TABLE 5.2 – Temps par tâche de la version de référence et ratio $\frac{t_{Halley}}{t_{reference}}$ obtenus sur les cas EP et ST pour de nombreuses configurations de tailles de buffers en considérant une taille de bloc fixée à 1024x128.

tâche constante (taille des blocs fixée à 1024x128), c'est-à-dire une forme de mise à l'échelle faible (c.-à-d. weak scaling).

Ainsi, on peut conclure que COMET n'introduit pas de surcoûts par rapport à la version OpenMP écrite manuellement qui seraient liés à la taille des données traitées.

5.3.4 Exploitation de la réutilisation de données en cache

Analyse du cas d'utilisation EP La figure 5.5 montre le temps de complétion du cas d'utilisation EP-EP en utilisant HALLEY sur des blocs de tailles différentes, avec un buffer de données de taille de 8192x8192, avec et sans barrière (*cf.* chapitre 4). Le côté gauche de la figure contient des cas qui n'ont pas besoin de repartitionnement entre les deux codes EP (même taille de bloc) et les cas du côté droit ceux qui nécessitent un repartitionnement en raison de la taille différente des blocs entre les deux codes EP. Les tailles de blocs relatives au premier EP sont désignées par B_{Mt1}

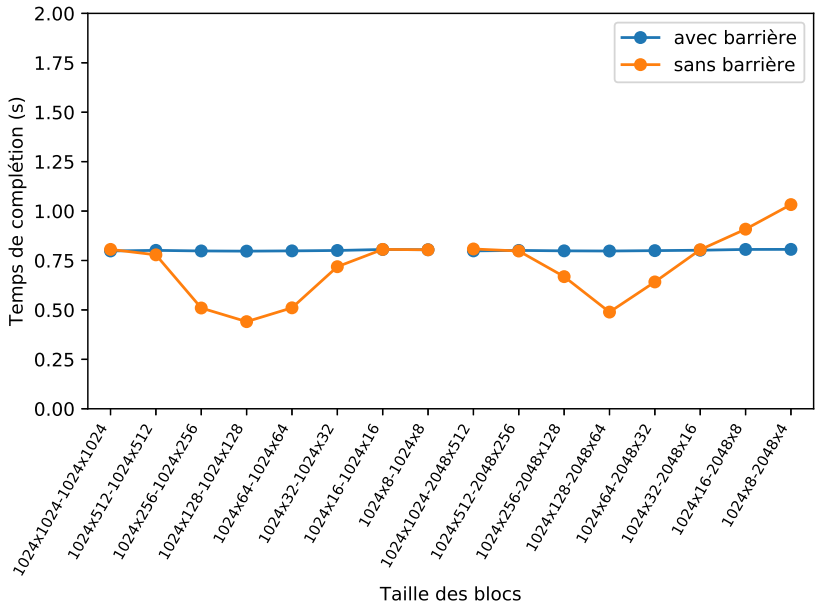

FIGURE 5.5 – Temps de complétion du cas d'utilisation EP-EP pour différentes tailles de blocs avec un buffer de taille 8192x8192. Les tailles de blocs des métatâches `Mt1` et `Mt2` sont représentées sous la forme $B_{Mt1} - B_{Mt2}$. Les versions sans repartitionnement se trouvent à gauche et celles avec repartitionnement à droite.

et celles du second par B_{Mt2}.

La partie gauche de la figure 5.5 montre que les versions sans barrière surpassent toujours les versions avec barrière si aucun repartitionement n'est nécessaire. Dans certains cas, l'éviction de la barrière permet une amélioration des performances d'un facteur 1,809. Cette amélioration provient de la réutilisation des données du cache entre les tâches générées à partir de la métatâche `Mt1` et celles de `Mt2`. La réutilisation des données n'est possible que lorsque les fragments sont suffisamment petits pour être intégrés au cache. En pratique, l'absence de barrière permet à l'ordonnanceur de suivre les dépendances entre les tâches de `Mt1` et `Mt2` : lorsqu'une tâche de `Mt1` se termine, il exécute directement les tâches de `Mt2` prêtes (celles dont les prédécesseurs se sont terminés). A contrario, la version avec barrière force le calcul de toutes les tâches de `Mt1` avant `Mt2`. Puisque l'ensemble des données ne peut pas tenir dans le cache (L3), quasiment aucune réutilisation de cache n'est possible.

Des temps de complétion plus rapides sont observés lorsqu'il y a suffisamment de tâches pour éviter une situation de famine (p. ex. avec une taille de blocs fixée à 1024x128). Une situation de famine survient lorsque l'ordonnanceur n'affecte pas bien les tâches sur les unités de calculs (due à une forte connexité du graphe de dépendances rendant l'ordonnancement des tâches difficile), ou qu'il est trop lent à réaliser cette affectation, ou même lorsque le nombre de tâches soumises est faible. Dans cette expérience, lorsque la taille de blocs est plus grande que 1024x64, la soumission des tâches commence à devenir trop lente, car elle est réalisée séquentiellement comme cela a été évoqué précédemment (ce phénomène est visible sur le cas

ST présenté par la figure 5.7). Une soumission des tâches trop lente par rapport à l'exécution des tâches peut agir comme une barrière si les tâches `Mt1` sont terminées avant la soumission des tâches `Mt2` : l'ordonnanceur n'est pas en mesure de tirer parti des dépendances entre les tâches.

La partie droite de la figure 5.5 montre qu'avec le repartitionnement les mêmes effets sont observés : la version sans barrière est 1,631 fois plus rapide, sauf pour des petites tailles de blocs où elle est jusqu'à 1,281 fois plus lente. L'efficacité de la version sans barrière est moindre par rapport au cas sans repartitionnement. La raison de ce phénomène est double : premièrement, il est plus difficile d'exploiter le cache en présence d'un repartitionnement et deuxièmement, le repartitionnement engendre la soumission de tâches supplémentaires qui augmente le temps de soumission qui constitue un goulot d'étranglement lorsque la taille des blocs est petite (dû au nombre imposant de tâches produites). En pratique, en présence d'un repartitionnement, le gain issu d'une exploitation efficace du cache est caché par les pertes provenant de la soumission séquentielle trop lente qui empêche toute exécution entrelacée (étant donné que les tâches de la métatâche `Mt1` sont en grande partie déjà exécutées avant même que la première tâche de la métatâche `Mt2` soit soumise). Dans le cas extrême où la taille du bloc est trop petite, le temps de complétion est limité par le temps de soumission des tâches, qui est ralenti par les tâches de repartitionnement supplémentaires.

Ainsi, une exécution à grain fin, que ce soit avec ou sans repartitionnement fourni par HALLEY, peut fournir de meilleures performances dans le cas EP-EP que l'utilisation d'une simple barrière.

Analyse du cas d'utilisation ST La figure 5.6 montre les résultats obtenus avec le cas d'utilisation ST-ST de la même manière que les expériences EP-EP présentées dans la figure 5.5.

Globalement, la version sans barrière est jusqu'à 1,182 fois plus rapide sans repartitionnement et 1,070 fois plus rapide avec repartitionnement. Ceci est conforme aux résultats obtenus dans le cas EP-EP. Cependant, le gain provenant de l'éviction de la barrière est moins important, car les dépendances entre les tâches `Mt1` et les tâches `Mt2` sont plus complexes dans ST ce qui laisse moins de possibilités pour ordonnancer les tâches en fonction des dépendances afin augmenter la réutilisation des données en cache.

La figure 5.7 présente le planning d'exécution des tâches (diagramme de Gantt) d'une itération du cas d'utilisation ST-ST obtenu avec une taille de bloc 1024x64 et des buffers de taille 8192x8192. Au centre du diagramme de Gantt, un ordonnancement en profondeur est appliqué et les tâches `Mt2` sont plus rapides qu'à la fin de l'exécution puisqu'elles réutilisent les données en cache. Cela ne se produit pas avant, car la phase de soumission prend du temps et qu'elle empêche ainsi la tâche d'être ordonnancée en profondeur plus tôt : à un instant donné, soit aucune tâche de `Mt2` n'a été soumise pour le moment, soit des tâches ont été soumises, mais les données sur lesquelles elles opèrent ne sont plus en cache (dans ce cas, leur exécution est reportée à la fin de l'ordonnancement).

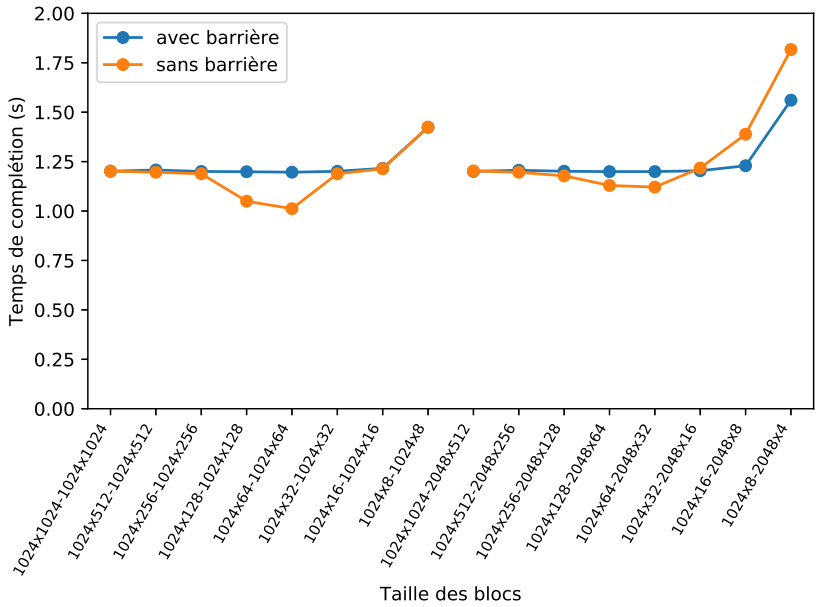

FIGURE 5.6 – Temps de complétion du cas d'utilisation ST-ST pour différentes tailles de blocs avec des buffers de taille 8192x8192. Les tailles de blocs des méta-tâches `Mt1` et `Mt2` sont représentées sous la forme $B_{Mt1} - B_{Mt2}$. Les versions sans repartitionnement se trouvent à gauche et celles avec repartitionnement à droite.

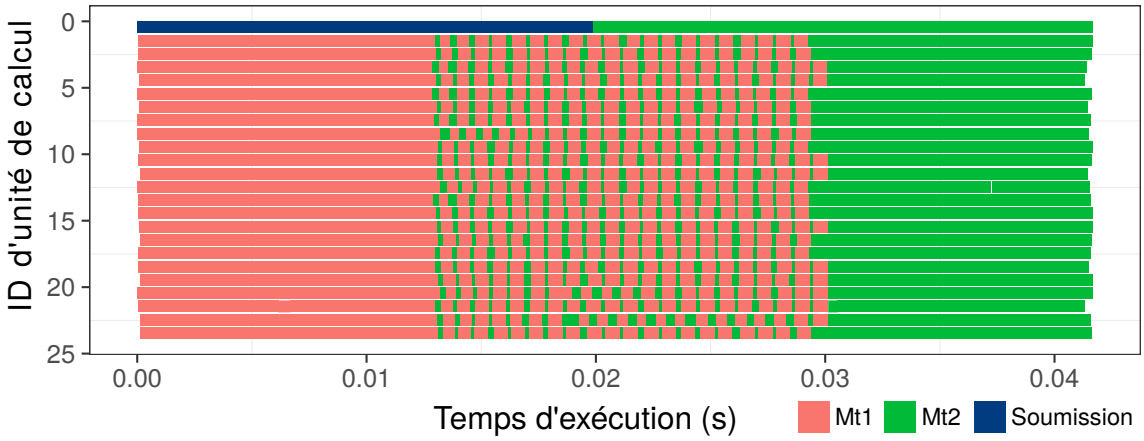

FIGURE 5.7 – Diagramme de Gantt de l'ordonnancement du graphe de tâches pour une seule itération du cas d'utilisation ST-ST obtenu avec une taille de blocs fixée à 1024x64 et des buffers de taille 8192x8192.

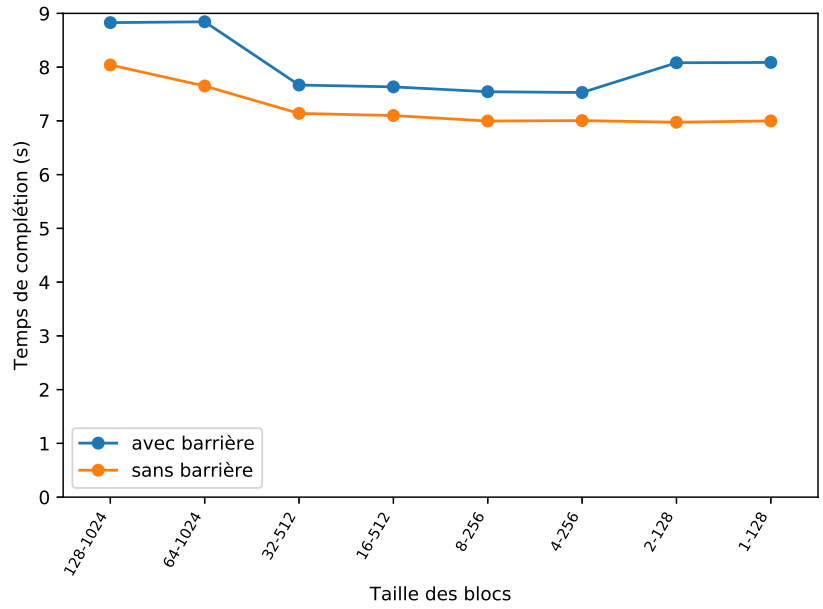

FIGURE 5.8 – Temps de complétion du cas d'utilisation TRANSP pour différentes tailles de blocs avec des buffers de taille 8192x8192. Les tailles de blocs sont représentées sous la forme $B_{compute} - B_{transpose}$ où $B_{compute}$ est le nombre de lignes des blocs des métatâches MtCompute et $B_{transpose}$ est la taille des blocs des métatâches MtTranspose.

Ainsi, une exécution à grain fin (avec ou sans repartitionnement) peut aussi fournir de meilleures performances dans le cas ST-ST que l'utilisation d'une simple barrière bien que les résultats soient bien plus modérés que dans le cas EP-EP à cause d'une marge de manœuvre beaucoup plus réduite (ordonnancement permettant une exploitation efficace du cache en temps borné difficile).

5.3.5 Recouvrement des transferts mémoires par des calculs

La figure 5.8 montre le temps de complétion de la version HALLEY du cas d'utilisation TRANSP avec et sans barrière pour différentes tailles de blocs. Dans cette expérience, la taille du buffer de données est fixée à 8192x8192.

Tout d'abord, la version sans barrière est toujours plus rapide que la version avec barrière : d'un facteur 1,098 à 1,159. Ceci est dû à deux facteurs : le déséquilibre de travail et l'utilisation du débit mémoire.

Déséquilibre de travail En effet, avec peu de tâches, les unités de calcul sont affamées en raison d'un déséquilibre de travail. Dans la version sans barrière, la version HALLEY exploite les dépendances réelles (granularité fine décrite par les expressions de relations) entre les tâches des différentes métatâches. L'ordonnanceur est capable d'entrelacer l'exécution des tâches issues des métatâches de calcul avec

celle des métatâches de transposition et celles engendrées par les opérations de repartitionnement, ce qui a pour effet de réduire les cas de famine, particulièrement à la fin de chaque phase de traitement (p. ex. période d'exécution des tâches issues d'une même métatâche). Ainsi, la composition de code offerte par COMET permet de répartir plus équitablement le travail à effectuer entre les ressources.

Utilisation du débit mémoire Avec un nombre de tâches suffisant, les tâches de transposition, limitée par le débit mémoire, s'exécutent plus rapidement sans barrière puisque leur période d'exécution peut chevaucher celles de tâches de calcul (qui utilisent très peu la mémoire). En effet, la mémoire est une ressource partagée par l'ensemble des cœurs d'une même socket et il est donc important de répartir son utilisation dans le temps afin de l'exploiter efficacement. L'exécution de tâches de différents types sur une même portion de temps est clairement observable sur le planning d'exécution des tâches de la version sans barrière exposée sur la figure 5.9. On peut voir que les tâches de transposition peuvent s'exécuter juste après le premier groupe de tâches de calcul ou juste avant le second groupe dispersant ainsi les tâches de transposition sur une plus longue période de temps. L'impact de ce phénomène est visible à la figure 5.10 qui présente une décomposition des temps de chaque phase du traitement : le temps des tâches issues des métatâches `Tr1` et `Tr2` dépend de la présence d'une barrière et de la taille des blocs. Pour toutes les configurations où une barrière est utilisée, les temps pour `Tr1` ou `Tr2` sont clairement supérieurs à la version équivalente sans barrière. Lorsqu'une barrière est utilisée, l'ordonnanceur exécute, sur l'ensemble des unités de calcul simultanément, des tâches `Tr1` ou `Tr2` ce qui engendre une pression élevée du trafic mémoire. En raison de la contention dans le trafic, les accès mémoire sont plus longs et donc le temps de complétion augmente. Dans les versions sans barrière, l'ordonnanceur exécute des tâches de calcul (`Mt1` et `Mt2`) en même temps que des tâches qui n'effectuent que des accès de mémoire (`Tr1` et `Tr2`) ce qui diminue la pression sur la mémoire, car les accès à la mémoire sont répartis sur une période de temps plus longue. Ainsi, la composition de code offerte par COMET permet d'exploiter plus efficacement des ressources partagées telles que la mémoire.

5.3.6 Performance du support exécutif GOMP

Afin de déterminer si les résultats observés avec KOMP se généralisent à d'autres supports exécutifs, cette section compare les résultats obtenus entre le support exécutif du projet GNU (GOMP) et KOMP.

Les résultats obtenus avec le support exécutif GOMP (GNU) sont similaires à ceux obtenus avec le support exécutif KOMP sur les expériences de passage à l'échelle en nombre de tâches. Cependant, GOMP introduit une surcharge de tâches plus élevée, ce qui a pour effet d'augmenter considérablement le temps d'exécution atteignant respectivement 14,1 s et 14,8 s sur les cas d'utilisation EP et ST au lieu de 2,1 s et 4,6 s avec KOMP. Ces résultats sont exposés dans le tableau B.1 en annexe (page 183).

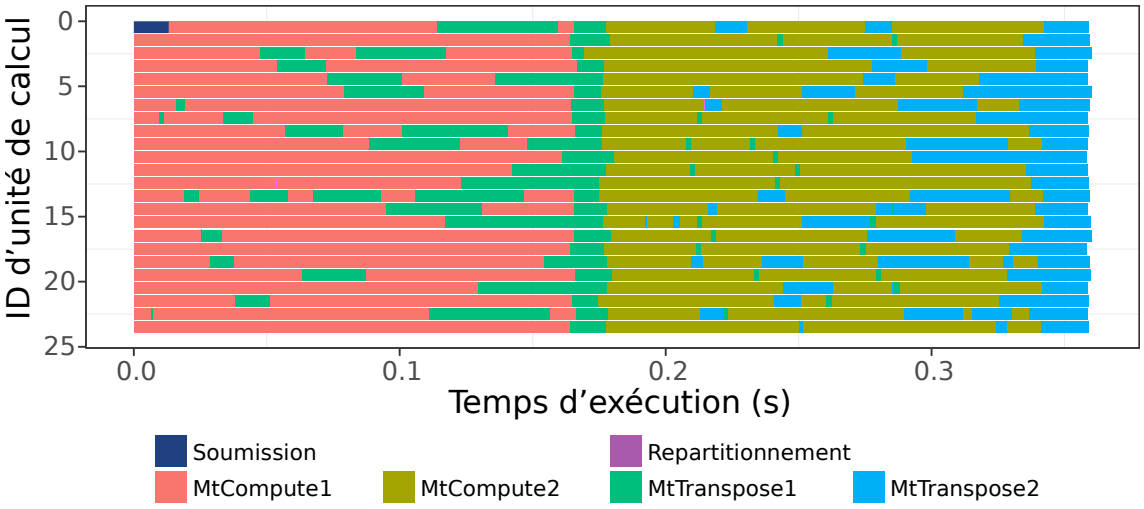

FIGURE 5.9 – Diagramme de Gantt de l'ordonnancement du graphe de tâches pour une seule itération du cas d'utilisation TRANSP obtenu avec un buffer de taille 8192x8192, une taille de blocs fixée à 8192x8 pour les tâches de calcul et 256x256 pour les tâches de transposition.

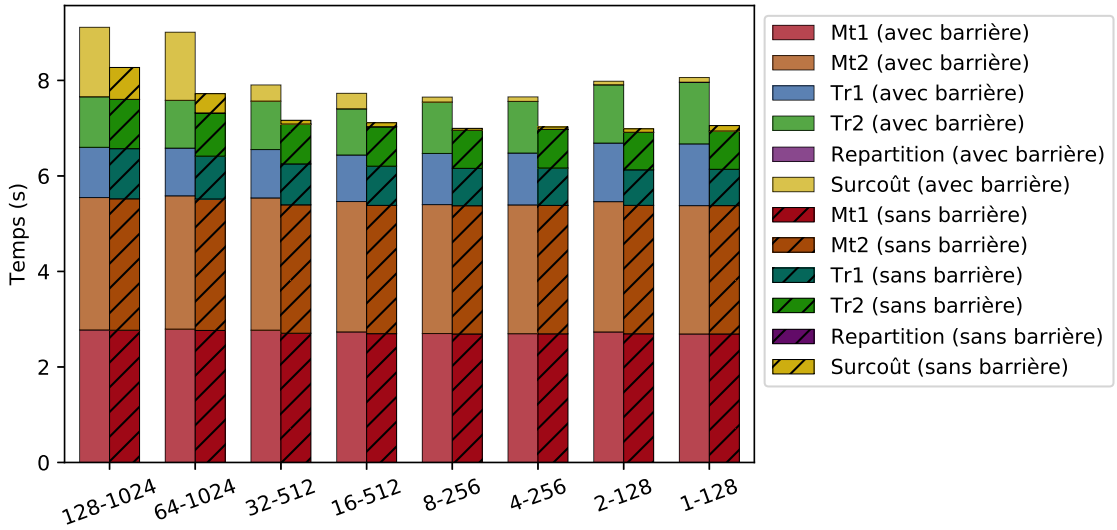

FIGURE 5.10 – Décomposition du temps de complétion du cas d'utilisation TRANSP pour différentes tailles de blocs avec un buffer de taille 8192x8192. Pour chaque taille de bloc, les versions avec et sans barrières sont analysées. Les tailles de blocs sont représentées sous la forme $B_{compute} - B_{transpose}$ où $B_{compute}$ est le nombre de lignes des blocs des métatâches MtCompute et $B_{transpose}$ est la taille des blocs des métatâches MtTranspose.

Les résultats du passage à l'échelle en fonction de la taille des données sont exactement les mêmes avec GOMP (avec une variation inférieure à 1% du temps de complétion). Ces résultats sont exposés dans le tableau B.2 en annexe (page 184).

Les expériences de composition se comportent différemment avec GOMP par rapport à KOMP. Les figures 5.11 et 5.12 présentent respectivement les temps de complétion des cas d'utilisation EP-EP et ST-ST avec le support exécutif GOMP dans des conditions identiques aux figures 5.5 et 5.6. Dans l'ensemble, la version sans barrière est légèrement plus rapide que la version avec, sur le cas d'utilisation EP-EP : jusqu'à 1,135 fois plus rapide sans repartitionnement et de 1,114 fois plus rapide à 1,277 fois plus lente avec repartitionnement. Dans le cas d'utilisation ST-ST, les performances des deux versions (avec et sans barrière) sont très proches. Sans tenir compte des valeurs extrêmes à droite des figures 5.11 et 5.12, la version sans barrière est 1,031 fois plus rapide que celle sans barrière, avec et sans repartitionnement. Avec GOMP, il y a un petit gain à supprimer la barrière lors d'une composition. Cela est dû à une réutilisation réduite du cache, comme nous l'avons déjà mentionné.

Les résultats du cas d'utilisation TRANSP avec GOMP sont similaires à ceux avec KOMP. En effet, la version sans barrière est toujours plus rapide que celle avec : d'un facteur 1,044 à 1,161. GOMP répartit bien les accès à la mémoire en les superposant avec le calcul. Ces résultats sont exposés la figure B.13 en annexe (page 183).

Malgré quelques différences, les résultats obtenus avec GOMP sont similaires à ceux de KOMP ce qui laisse penser que les phénomènes observés précédemment peuvent être généralisés à d'autres supports exécutifs, bien qu'une étude sur plus de supports exécutifs soit nécessaire afin de confirmer cette analyse. Les résultats obtenus avec GOMP se révèlent cependant moins efficaces que ceux de KOMP. Cette perte d'efficacité dans le cas de GOMP est due à un ordonnancement proche de l'ordre de soumission qui n'est pas propice à une réutilisation des données en cache. Ce point est étudié plus en détail dans la section suivante.

5.4 Discussion

Récapitulatif À partir des résultats précédents, nous pouvons conclure que les performances de Halley sont équivalentes aux versions de référence OpenMP écrite manuellement pour les cas d'utilisation évalués, en particulier lorsque des patrons de calcul simple sont utilisés (c.-à-d. cas d'utilisation synthétiques). Cela signifie que le modèle Comet peut être mis en œuvre sans introduire de surcoût conséquent à l'exécution sur des cas synthétiques. Avec la complexité croissante des patrons, surtout en présence de cas de compositions, Comet démontre ses avantages : l'expression de la composition reste simple tandis que la mise en œuvre Halley peut générer automatiquement une exécution efficace en utilisant des synchronisations point à point entre les tâches sans aucune synchronisation globale (barrière OpenMP). Cette solution est très efficace, même pour les petites tâches liées à la mémoire, pour exploiter la réutilisation du cache ainsi que pour utiliser efficacement les res-

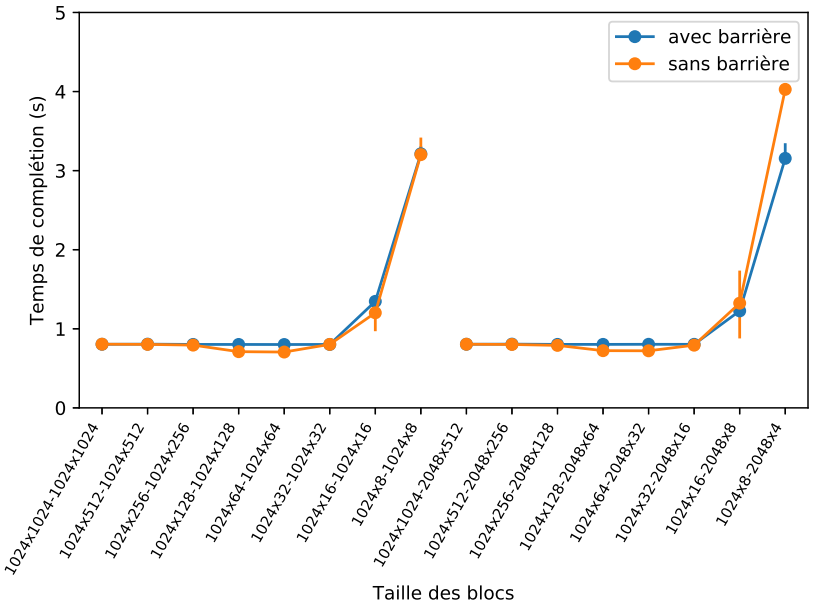

FIGURE 5.11 – Temps de complétion du cas d'utilisation EP-EP pour différentes tailles de blocs avec des buffers de taille 8192x8192 en utilisant le support exécutif GOMP. Les tailles de blocs des métatâches `Mt1` et `Mt2` sont représentées sous la forme $B_{Mt1} - B_{Mt2}$.

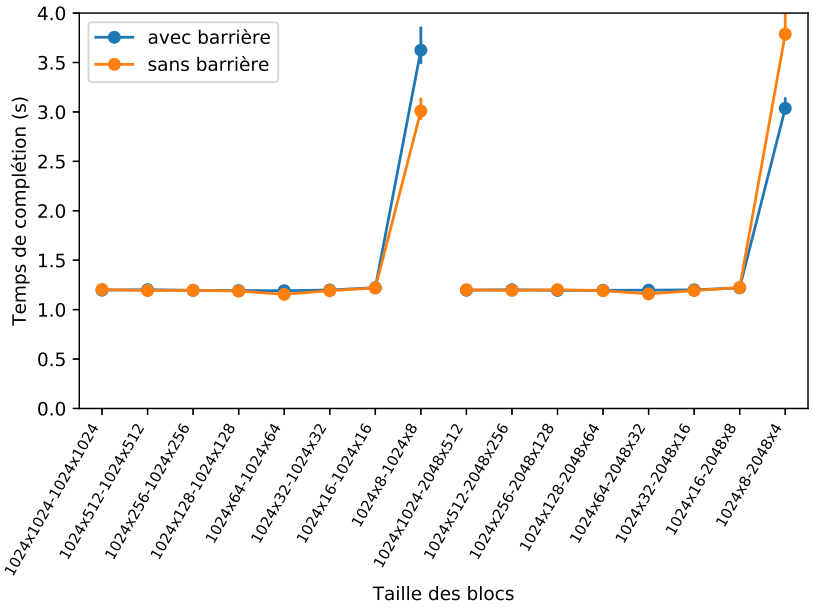

FIGURE 5.12 – Temps de complétion du cas d'utilisation ST-ST pour différentes tailles de blocs avec des buffers de taille 8192x8192 en utilisant le support exécutif GOMP. Les tailles de blocs des métatâches `Mt1` et `Mt2` sont représentées sous la forme $B_{Mt1} - B_{Mt2}$.

sources partagées malgré la présence de partitionnement différent et l'usage d'une composition de codes de haut niveau.

Les résultats obtenus avec les supports exécutifs d'OpenMP de production sont, à première vue, décevants. GOMP (ainsi que le support exécutif d'Intel, dont les résultats ne sont pas présentés ici) ne tire pas parti de l'éviction de barrières permettant d'améliorer significativement les performances. Cependant, il est important de noter que plusieurs hypothèses ont été considérées avant que nous ne nous concentrions sur les performances de Halley.

Sérialisation et ordre de la soumission des tâches En sérialisant la soumission des tâches, le surcoût des tâches devient un facteur limitant : la granularité des tâches doit être revu à la hausse pour compenser les surcoûts [46]. Cela a pour effet d'impacter négativement l'ordonnancement des tâches en introduisant un comportement similaire à la présence d'une barrière. En effet, les tâches qui peuvent exploiter la réutilisation du cache sont *de facto* soumises lorsque la plupart des tâches précédentes ont été exécutées. L'ordonnanceur ne peut donc pas exécuter les tâches en profondeur de manière à améliorer les performances des cas d'utilisation considérés (car il n'a qu'une version partielle du graphe de tâches à un instant donné).

Plusieurs solutions existent pour résoudre le problème de sérialisation de la soumission des tâches. L'une d'elles consiste à adapter Halley pour qu'il puisse soumettre des tâches dans un ordre proche de celui choisi par l'ordonnanceur à l'exécution afin de simplifier son travail (p. ex. soumission des tâches en profondeur), notamment en ayant recours à des stratégies d'ordonnancement statique (appliquée au moment de la compilation) ou en analysant la structure du graphe lors de la première exécution d'une section afin de soumettre les tâches des prochaines exécutions de la section de manière plus efficace. Cette approche requiert des changements majeurs dans Halley, car elle nécessite une analyse globale du graphe qui est le résultat de la composition de morceaux de graphes générés par des greffons indépendants variés (p. ex. greffons de relations et de repartitionnement), dont le comportement précis n'est pas connu par Halley. De plus, cette approche souffre de problèmes de passage à l'échelle : lorsque le nombre d'unités de calcul devient imposant, la soumission des tâches séquentielle risque d'être trop lente pour fournir suffisamment de travail à toutes les unités de calculs. Une solution plus prometteuse consiste à soumettre les tâches en parallèle. Cependant, la soumission de tâches interdépendantes en parallèle n'est pas actuellement supportée par OpenMP 4.5 (ni par la nouvelle version 5.0).

Efficacité des stratégies d'ordonnancement En soumettant en groupe les tâches issues des métatâches, l'ordonnanceur doit fournir un effort conséquent pour réorganiser les tâches de manière à les exécuter en profondeur dans les cas d'utilisation traités. Or, certains ordonnanceurs [26] ont recours à des stratégies employant des fenêtres glissantes pour n'analyser qu'une sous partie du graphe de tâches soumis afin de limiter les coûts en calculs ainsi que l'usage de la mémoire. D'autres [55] décident d'exécuter les tâches lorsque trop de tâches ont été soumises (c.-à-d. cut

off) afin de réduire les surcoûts associés à la taille des structures de données [46]. C'est en particulier le cas de GOMP qui n'exécute plus les tâches en profondeur lorsque le graphe devient trop imposant (p. ex. seuil fixé expérimentalement à 64 tâches). KOMP supporte le vol de travail T.H.E. issu de Cilk [50] qui est basé sur des files d'attente illimitées de tâches. Cette stratégie de vol de travail est particulièrement bien adaptée à la manière dont fonctionne HALLEY car chaque métatâche engendre un ensemble de tâches potentiellement grand qui dépend des groupes précédemment soumis. Toutefois, il est important de noter que l'exécution des tâches en profondeur n'est pas toujours la meilleure méthode d'ordonnancement, notamment lorsque les tâches d'une même métatâche travaillent sur des données partagées (ces cas d'utilisation sont moins sensibles aux problèmes évoqués).

Conception d'une interface efficace et portable pour les supports exécutifs Fournir une mise en œuvre de COMET au-dessus des supports exécutifs d'OpenMP de manière portable est un défi. Outre les problèmes d'ordonnancement de tâches, les supports exécutifs d'OpenMP supportent relativement mal un nombre important de tâches dépendantes (à grains fins). Les surcoûts introduits par GOMP sont plus importants que ceux du support exécutif d'Intel étendu par KOMP et limitent le passage à l'échelle de notre approche. Les dépendances sont la source de ces surcoûts. Le passage des dépendances au support exécutif ainsi que leur résolution, ne peut pas être parallélisé avec OpenMP en raison de limitations dans la spécification [94] qui impose une création séquentielle des tâches interdépendantes. Afin de surmonter cette limitation, nous cherchons à exporter des fonctions spécifiques aux supports exécutifs pour contourner les calculs de dépendances afin que les greffons de relations (*cf.* Section 3.2) puissent les calculer lors de la compilation d'un assemblage HALLEY.

Choix du nombre optimal de cœurs à utiliser Enfin, il est important de noter que nous avons fait des expérimentations en utilisant uniquement le nombre maximum de cœurs présent sur une socket (*cf.* introduction de cette section). Le nombre de cœurs utilisé n'est pas optimal pour les cas d'utilisation limités par la mémoire dans lesquels une contention est observable, par exemple dans les résultats présentés sur ST-ST. Cependant, ce choix a été réalisé pour trois raisons.

Premièrement, de nombreuses applications de production (telles que Gysela) ayant recours à OpenMP utilisent toutes les unités de calcul disponibles. Dans ces applications, les performances optimales de chaque traitement séparé peuvent être atteintes avec un nombre variable d'unités de calcul. Par exemple, les traitements limités par la mémoire peuvent atteindre des performances optimales avec quelques unités de calcul alors que d'autres, plus intensifs, en termes de calcul peuvent demander l'ensemble des unités de calcul mises à disposition. Plus généralement, le choix du nombre d'unités de calcul pour un traitement dépend de sa capacité à passer à l'échelle. Ainsi, si l'adaptation du nombre d'unités de calcul est une solution viable pour un unique traitement, cette méthode n'est souvent pas optimale à l'échelle d'une application. En effet, lorsque les traitements sont composés ensemble, certains groupes de traitements peuvent s'exécuter simultanément, chacun sur un

sous-ensemble de cœurs. Ce phénomène est notamment visible sur le cas d'utilisation TRANSP dans lequel les tâches de calcul et de transposition peuvent s'exécuter simultanément. Cet effet est confirmé dans la thèse de Andra-Ecaterina Hugo [64] qui s'intéresse à l'utilisation de contextes ordonnancement pour le support exécutif StarPU afin de combiner efficacement des codes HPC existants (p. ex. solveurs linéaires).

Deuxièmement, avec moins d'unités de calcul, HALLEY est capable d'exploiter la réutilisation du cache, car l'ordonnanceur aura accès à l'ensemble des dépendances de la tâche plus rapidement, par rapport à nos conditions expérimentales.

Troisièmement, faire des expériences avec le nombre maximum d'unités de calcul tout en gardant la capacité de réordonnancer les tâches de grain fin au-delà de la limite de composition est une étape nécessaire afin de pouvoir exploiter ensuite les architectures massivement multi-cœurs tels que les many-cœurs.

Choix du partitionnement Il est important de noter que le partitionnement utilisé dans les expériences de ce chapitre est uniquement constitué de blocs régulier (cas uniforme). Bien qu'il soit possible d'utiliser une décomposition en blocs irréguliers, l'usage d'un partitionnement uniforme se révèle particulièrement pertinent dans ces expériences puisque le coût des calculs réalisés dépendent uniquement de la quantité de données traitées. L'usage de partitionnement non-uniforme est supporté par COMET et son implémentation mais ce dernier n'a pas été implémenté. L'utilisation de tels partitionnement nécessite la mise en œuvre de greffons de partitionnement et repartitionnement dédiés (ces greffons ont la liberté de traiter chaque blocs indépendamment ou alors de travailler sur une représentation factorisée du partitionnement afin de déterminer les dépendances entre les fragments en cas de repartitionnement).

5.5 Conclusion

Ce chapitre a évalué COMET, et plus précisément la mise en œuvre HALLEY, sur cinq cas d'utilisation synthétiques (EP, ST, TRANSP, EP-EP, ST-ST) représentatifs de traitements inclus dans les applications de production HPC telles que Gysela. L'évaluation a été divisée en deux parties distinctes : l'une s'est concentré sur des aspects relatifs au génie logiciel offert par HALLEY et l'autre s'est intéressé aux performances de HALLEY sur ces cas d'utilisation. La première partie a présenté comment HALLEY permet de séparer les préoccupations des cas d'utilisation à travers une discussion. Elle montre que HALLEY permet de décrire indépendant des codes parallèles séparés dans des métatâches et composants indépendants ensuite composés au sein d'un assemblage. La seconde partie a analysé les performances de la mise en œuvre HALLEY à travers une comparaison des versions utilisant HALLEY avec des versions OpenMP écrites manuellement pour chacun des cas d'utilisation, ainsi que par une comparaison des stratégies de repartitionnement (c.-à-d. impact de la présence d'une barrière). Les résultats montrent que les performances de HALLEY

sont équivalentes aux versions de référence OpenMP écrites manuellement sur les cas d'utilisation synthétiques évalués et donc que COMET peut être mis en œuvre efficacement. Ils montrent aussi que cette mise en œuvre permet de composer efficacement des codes parallèles à base de tâches en utilisant des synchronisations point à point entre les tâches sans aucune synchronisation globale (barrière), offrant ainsi de nombreux bénéfices tels que la réutilisation des données en cache ou encore un usage plus efficace des ressources partagées à l'échelle d'une application. Une analyse sur un cas d'utilisation réaliste tiré de l'application de production Gysela est réalisée dans le chapitre 6.

Chapitre 6

Évaluation du cas de l'advection 2D issue de Gysela

Ce chapitre évalue le modèle COMET et plus précisément sa mise en œuvre HALLEY dans un cas d'utilisation réaliste : une partie de l'application de simulation gyrocinétique de plasmas par confinement magnétique pour la fusion Gysela appelée *advection 2D*.

Ce chapitre rappelle tout d'abord la place de l'advection 2D dans Gysela, décrit précisément son fonctionnement (c.-à-d. les algorithmes, le couplage et les variantes), les différentes versions implémentées et évoque les motivations de l'utilisation de COMET dans ce cadre. Ensuite, le chapitre évalue les propriétés relatives au génie logiciel apportées par COMET en termes de séparation des préoccupations et de réutilisation de portions de codes. Puis ce chapitre évalue les performances des versions HALLEY en termes de temps de complétion et de passage à l'échelle. Pour cela, cette partie étudie le temps de complétion des versions HALLEY par rapport à des versions de référence. Ce chapitre discute des avantages et des limites de l'approche avant de finalement conclure.

6.1 Étude du cas d'utilisation

Cette section décrit dans quel contexte le cas d'utilisation d'advection 2D extrait de Gysela se situe et présente son fonctionnement. En particulier, cette section décrit la structure de l'algorithme utilisé pour effectuer cette advection, ses variantes, la manière dont le parallélisme est introduit dans l'application actuellement et celles que nous avons mises en place afin d'établir une version HALLEY.

6.1.1 Description, objectifs et enjeux

Décrit d'une manière générale dans la section 2.2.5, le solveur Vlasov est une des parties fondamentales de Gysela. Ce solveur consiste en une séquence d'advections unidimensionnelles (c.-à-d. 1D) ou bidimensionnelles (c.-à-d. 2D) entrecoupées par

des transpositions globales visant à ajuster les dimensions du jeu de données traité par les advections.

Les advections sont les opérations les plus coûteuses en temps de calcul dans Gysela. L'optimisation de ces advections constitue donc un objectif important pour les concepteurs du logiciel. Cependant, elle ne doit pas se faire au détriment de la maintenabilité de l'application. En effet, le code de Gysela évolue sans cesse au fil du temps : les méthodes de parallélisation et les schémas numériques sont adaptés afin de les rendre toujours plus précis et efficaces sur les architectures d'aujourd'hui et de demain.

L'advection 2D de Gysela est un exemple qui met en exergue ces problèmes (c.-à-d. conflits entre performance et génie logiciel) : ce cas d'utilisation peut être scindé en plusieurs parties possédant des variantes algorithmiques dont les propriétés sont singulières (p. ex. performance, précision, etc.) et l'optimisation de cas est réalisée manuellement en couplant intelligemment les différentes composantes (c.-à-d. de manière à produire une exécution entrelacée favorisant une bonne localité spatiale et temporelle).

L'objectif de COMET, plus précisément de HALLEY, est de pouvoir exprimer les différentes parties indépendamment afin de les composer ensuite sans avoir à coupler les codes manuellement (à grain fin). La séparation dans des codes indépendants et l'expression explicite des dépendances logicielles permet d'adapter simplement une des parties en limitant l'impact sur le reste du code. Il devient alors possible de remplacer une variante algorithmique par une autre, mais aussi de modifier la stratégie de parallélisation d'une des variantes en minimisant l'impact sur le reste du logiciel.

6.1.2 Algorithme de référence

Structure de l'algorithme L'advection 2D est composée de deux parties : le *calcul de gradients* et l'*advection* proprement dite. La première partie consiste à calculer deux champs de gradients bidimensionnels (pouvant être décrit par quatre champs scalaires) à partir du champ électromagnétique tridimensionnel produit par le solveur de Poisson. La seconde partie consiste à utiliser ces champs de gradient afin d'effectuer une advection semi-lagrangienne [1] du champ scalaire décrivant la densité du plasma en tout point (fonction de distribution $\bar{f}$).

La seconde partie du cas d'utilisation (c.-à-d. l'advection), est elle-même constituée de deux parties : un calcul de *pieds des caractéristiques* et une *interpolation*. En premier lieu, pour chaque position X_{dst} sur le maillage à l'instant t, on détermine l'origine de la caractéristique (pieds des caractéristiques) à partir des lois du mouvement (p. ex. forces électrostatiques). Le résultat est une position X_{src} à l'instant $t - \Delta t$ qui n'appartient pas forcément au maillage. Pour calculer sa valeur, on interpole en utilisant les points du maillage. Plusieurs méthodes d'interpolation peuvent

1. Rappelons qu'un schéma semi-lagrangien cherche à calculer en tout point du maillage des caractéristiques à un instant t à partir d'une interpolation des caractéristiques déjà calculées à l'instant $t - \Delta t$.

être utilisées en pratique : une interpolation basée sur des *splines* ou un polynôme de *Lagrange*.

Bien que Gysela opère sur l'ensemble de la fonction de distribution à cinq dimensions, aucune advection n'intervient dans la dimension μ. On a ainsi affaire à autant de problèmes indépendants que de points dans la dimension μ. Cette dimension est donc trivialement parallélisée en MPI dans Gysela. Chaque processus ne gère qu'une unique valeur de μ et donc un problème à quatre dimensions.

Variante Splines L'algorithme 2 présente de manière simplifiée [2] le fonctionnement de la variante *Splines* de l'advection 2D opérant sur les dimensions r, θ de la fonction $\bar{f}^\star$ (fonction de distribution). Le calcul est divisé en six étapes dont la première correspond au calcul des gradients (première partie de l'algorithme) et les autres sont relatives à l'advection (seconde partie de l'algorithme).

1. Les lignes 4 à 6 consistent à calculer des gradients du champ électromagnétique. Ces calculs sont indépendants selon toutes les dimensions.
2. La ligne 7 effectue des calculs nécessaires à l'interpolation par splines (copie dans des tableaux plus grands avec des bordures et application d'un prétraitement des valeurs). Ces calculs sont indépendants selon toutes les dimensions.
3. La ligne 9 remplit les bords des tableaux prétraités. Cette étape, peu coûteuse, peut être parallélisée bien que cela soit difficile.
4. Les lignes 10 à 15 calculent des coefficients de splines utiles pour effectuer l'interpolation. Le calcul des coefficients peut se faire en appliquant des opérations parallèles indépendantes le long de la dimension θ, puis d'autres parallèles indépendantes le long de la dimension r sur les tableaux intermédiaires (obtenus en itérant le long de la dimension θ). Il est aussi possible d'appliquer une transposition entre les deux opérations parallèles afin de calculer les coefficients, à l'instar du cas d'utilisation TRANSP présenté au chapitre 5.
5. Les lignes 16 à 18 consistent à appliquer un stencil pour calculer des champs de déplacement. Cette étape est ainsi parallélisable selon les dimensions (r, θ) et ressemble fortement au cas d'utilisation ST présenté dans le chapitre 5.
6. Les lignes 19 à 23 effectuent une interpolation à base de splines en fonction de la valeur de $feet_r$ et $feet_\theta$. Ce calcul peut être appliqué entièrement en parallèle. Cependant, il effectue une indirection par valeur sur les tableaux traités qui dépend du contenu de $feet_r$ et $feet_\theta$. Par conséquent, l'ensemble des opérations précédentes sur le plan traité doivent être terminées avant de commencer l'interpolation.

Variante Lagrange L'algorithme 3 présente de manière simplifié le fonctionnement de la variante *Lagrange* de l'advection 2D opérant sur les dimensions r, θ de la fonction de distribution $\bar{f}^\star$. Cette variante se décompose en cinq étapes :

2. Seule la structure générale est décrite, aucune constante n'est présente, les pré-calculs ne sont pas considérés, la gestion des bords des tableaux est minimaliste, certains champs sont agrégés et les tailles des tableaux sont approximatives (à une petite valeur constante près).

Entrées : $\bar{f}^{\star}(N_r, N_\theta, N_\varphi, N_{v_\parallel})$, $electric\_field(N_r, N_\theta, N_\varphi)$, Δt
Sorties : $\bar{f}^{\diamond}(N_r, N_\theta, N_\varphi, N_{v_\parallel})$

pour tous $v_\parallel$ **faire en parallèle**
  pour tous φ **faire en parallèle**
    pour tous θ **faire**
      // Étape 1 : calcul des champs de déplacement
      pour tous r **faire**
        $velocities(r,\theta) \leftarrow$ derived_from($electric\_field(\star,\theta,\varphi)$, θ, φ, $v_\parallel$);
      fin
      // Étape 2 : préparation de l'interpolation
      $tmprhs(\star,\theta) \leftarrow$ splines_prepare($\bar{f}^{\star}(\star,\theta,\varphi,v_\parallel)$, $velocities(\star,\theta)$, θ, φ, $v_\parallel$);
    fin
    // Étape 3 : suite de la préparation de l'interpolation
    $tmprhs(\star,\star) \leftarrow$ ajust_borders($tmprhs(\star,\star)$);
    // Étape 4 : calcul des coefficients de splines
    pour tous θ **faire**
      $bcoef(\star,\theta) \leftarrow$ compute_splines_coefs_x($tmprhs(\star,\theta)$);
    fin
    pour tous r **faire**
      $splines(r,\star) \leftarrow$ compute_splines_coefs_y($bcoef(r,\star)$);
    fin
    // Étape 5 : calcul des champs de déplacement
    pour tous r,θ **faire**
      $[\Delta r(r,\theta), \Delta\theta(r,\theta)] \leftarrow$ compute_splines_displacement($splines(\star,\star)$, $tmprhs(r,\theta)$, Δt, r, θ);
    fin
    // Étape 6 : interpolation
    pour tous r,θ **faire**
      $feet_r \leftarrow r - \Delta r(r,\theta)$;
      $feet_\theta \leftarrow \theta - \Delta\theta(r,\theta)$;
      $\bar{f}^{\diamond}(r,\theta,\varphi,v_\parallel) \leftarrow$ splines_interpolate($splines(\star,\star)$, $feet_r$, $feet_\theta$);
    fin
  fin
fin

Algorithme 2 : Pseudo-code simplifié de la variante Splines de l'advection 2D opérant sur les dimensions r,θ de la fonction de distribution $\bar{f}^{\star}$. Les identifiants $tmprhs$, $velocities$, $bcoef$, $splines$, Δr et $\Delta\theta$ sont des valeurs temporaires.

1. Les lignes 4 à 6 consistent à calculer des gradients (première partie de l'algorithme) identique à la variante Splines. Les autres étapes sont relatives à l'advection (seconde partie de l'algorithme).
2. Les lignes 7 à 9 sont similaires à la seconde étape de la variante Splines (excepté que la version Lagrange n'utilise pas de bordures).
3. La ligne 11 consiste à copier un plan de la fonction de distribution dans un tableau plus grand avec des bordures utiles pour l'interpolation. Cette étape peut être parallélisée bien que cela n'ait souvent que peu d'intérêt (le débit mémoire est généralement le facteur limitant).
4. Les lignes 12 à 14 sont similaires à l'étape 5 de la variante Splines.
5. Les lignes 15 à 19 effectuent une interpolation, tout comme la variante Splines, mais en utilisant ici des polynômes de Lagrange. Ce calcul aussi est divisible en un ensemble opérations indépendantes réalisables en parallèle selon les dimensions r et θ. Toutefois, contrairement à la variante Splines, chacune des opérations en parallèle nécessite uniquement la copie de la fonction de distribution entière et le pied de la caractéristique (vecteur de déplacement).

Notations Les champs et les fonctions présents dans les algorithmes 2 et 3 sont simplifiés. En effet, certains champs, tels que *velocities* ou $electric\_field$, font en réalité référence à plusieurs champs. Il en est de même pour les fonctions telles que *derive_from*, divisibles en plusieurs autres (chacune d'entre elles traite alors des champs différents). Plus précisément, chaque champ de la liste suivante fait en pratique référence à plusieurs champs scalaires (c.-à-d. tableaux multidimensionnels de scalaires) :

— $electric\_field$: $\partial J_0 \Phi dr$, $\partial J_0 \Phi d\theta$ et $\partial J_0 \Phi d\phi$;
— $velocities$: $vExB\nabla r$, $vExB\nabla\theta$, $vD\nabla r$ et $vD\nabla\theta$;
— $tmprhs$: $tmprhs\_ux$, $tmprhs\_uy$ et $tmprhs\_f$;
— $splines$: $uxspline\_r\theta$, $uyspline\_r\theta$ et $hhspline\_r\theta$;
— $ufield$: ux_val, uy_val.

Stratégie de parallélisation Dans Gysela tout comme dans les versions de référence, l'expression du parallélisme dans ces deux variantes se fait par une simple section parallèle OpenMP et une boucle parallèle sur les dimensions $(\varphi, v_\parallel)$. Des données temporaires sont allouées, configurées et détruites dans la section parallèle (dans chacun des fils d'exécution). Par exemple, la variante Splines stocke des informations nécessaires pour calculer les splines, précalculées par chacun des fils d'exécution au début de la section, évitant ainsi de les recalculer continuellement dans la boucle.

Différences par rapport au code de Gysela Les versions présentées par les algorithmes 2 et 3 constituent les *versions de référence* de ce chapitre. Ces deux versions sont fonctionnellement identiques à celle présentes directement dans le code Gysela

Entrées : $\bar{f}^{\star}(N_r, N_\theta, N_\varphi, N_{v_\parallel})$, $electric\_field(N_r, N_\theta, N_\varphi)$, Δt
Sorties : $\bar{f}^{\diamond}(N_r, N_\theta, N_\varphi, N_{v_\parallel})$

pour tous $v_\parallel$ **faire en parallèle**
  pour tous φ **faire en parallèle**
    pour tous θ **faire**
      // Étape 1 : calcul des champs de déplacement
      pour tous r **faire**
        $velocities(r,\theta) \leftarrow$ derived_from($electric\_field(\star,\theta,\varphi)$, θ, φ, $v_\parallel$);
      fin
      // Étape 2 : préparation de l'interpolation
      pour tous r **faire**
        $ufield(r,\theta) \leftarrow$ lagrange_prepare($velocities(r,\theta)$, r, θ, φ, $v_\parallel$);
      fin
    fin
    // Étape 3 : préparation de l'interpolation
    $f\_borders(\star,\star) \leftarrow$ copy_with_borders($\bar{f}^{\star}(\star,\star,\varphi,v_\parallel)$);
    // Étape 4 : calcul des champs de déplacement
    pour tous r, θ **faire**
      $[\Delta r(r,\theta), \Delta\theta(r,\theta)] \leftarrow$ compute_lagrange_displacement($ufield(\star,\star)$, Δt, r, θ);
    fin
    // Étape 5 : interpolation
    pour tous r, θ **faire**
      $feet_r \leftarrow r - \Delta r(r,\theta)$;
      $feet_\theta \leftarrow \theta - \Delta\theta(r,\theta)$;
      $\bar{f}^{\diamond}(r,\theta,\varphi,v_\parallel) \leftarrow$ lagrange_interpolate($f\_borders(\star,\star)$, $feet_r$, $feet_\theta$);
    fin
  fin
fin

Algorithme 3 : Pseudo-code simplifié de la variante Lagrange de l'advection 2D opérant sur les dimension r, θ de la fonction de distribution $\bar{f}^{\star}$. Les identifiants $velocities$, $ufield$, $f\_borders$, Δr et $\Delta\theta$ sont des valeurs temporaires.

(c.-à-d. elles fournissent les mêmes résultats et mettent en œuvre les mêmes méthodes de calcul ainsi que la même stratégie de parallélisation). Cependant, elles ont été remaniées par rapport au code originel afin de pouvoir analyser plus facilement le flot des données et permettre de mettre en œuvre une version à base de tâches. Les modifications comprennent : la restructuration du code de manière à obtenir un enchaînement de nids de boucles contenant des appels de fonctions simples, la suppression des variables globales utilisées en écriture (et minimisation des variables en lecture), la suppression des portions de code utilisant MPI et adaptation des stratégies d'allocations. Il est important de noter que ces versions ont été extraites du code Gysela (en FORTRAN) afin de mettre au point un prototype en C++, indépendant du reste de l'application.

Analyse L'advection 2D est constituée de deux variantes mettant en œuvre des algorithmes assez différents avec une partie redondante : le calcul des gradients. Ce calcul peut être isolé du reste des deux algorithmes de manière à pouvoir partager du code entre les deux variantes (c.-à-d. réutilisation de code). Cependant, le couplage du calcul des gradients avec la partie qui suit dans les deux variantes n'est pas trivial. En effet, l'entrelacement des deux parties se fait à grain fin et la granularité des opérations réalisées par deux variantes change tout au long des algorithmes (c.-à-d. certaines opérations traitent un plan indivisible, alors que d'autres, des lignes entières ou seulement des cellules). Ces aspects doivent être pris en comptes lors de l'élaboration de versions utilisant COMET.

6.1.3 Parallélisation à base de tâches

L'étude d'une version à base de tâches (basée sur les versions de référence) est nécessaire à l'élaboration d'une version COMET. Cependant, la mise au point d'une version à base de tâches capable de supporter les différents niveaux de granularité présents dans les versions de référence est relativement difficile. Afin de simplifier le problème, nous allons nous concentrer sur une version qui fonctionne à la granularité du plan. Cela signifie que les tâches produites n'opèrent que sur un plan à la fois.

Variante Splines La figure 6.1 décrit le graphe de tâches de la variante Splines basé sur l'algorithme 2 et opérant à la granularité du plan. Il est important de noter que certaines tâches issues de la fonction *derive_from* n'opèrent que sur des constantes qui n'apparaissent pas sur la figure.

Variante Lagrange La figure 6.2 décrit le graphe de tâches de la variante Lagrange basé sur l'algorithme 3 et opérant à la granularité du plan. La légende de cette figure est identique à celle de la figure 6.1.

Stratégie de parallélisation En analysant les dépendances à l'échelle du plan, visibles sur les figures 6.1 et 6.2, il est possible d'extraire le parallélisme de tâches

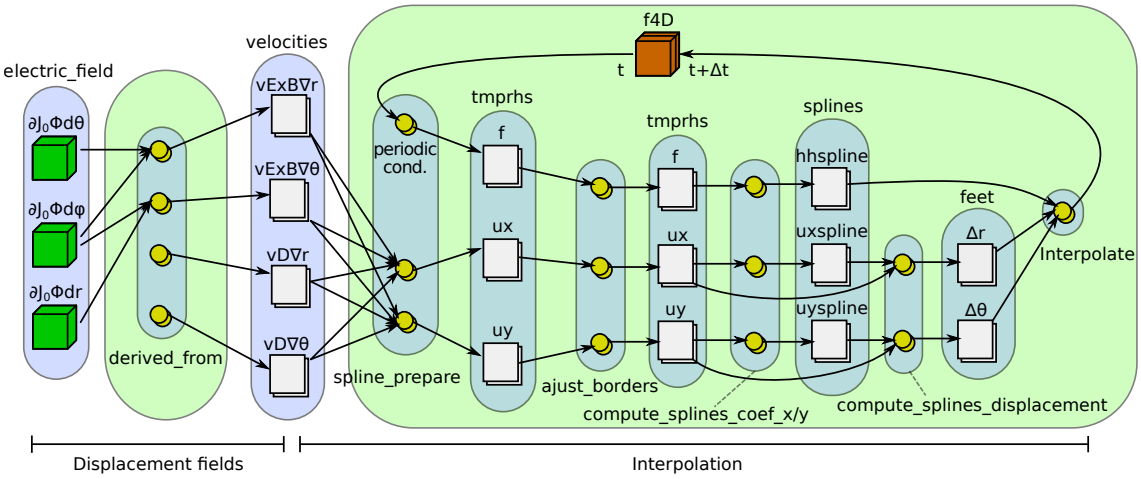

FIGURE 6.1 – Graphe de tâches de la variante Splines basé sur l'algorithme 2 opérant à la granularité du plan. Les cubes verts symbolisent des buffers de données en lecture ($\partial J_0 \Phi dr$, $\partial J_0 \Phi d\theta$ et $\partial J_0 \Phi d\phi$). Le cube marron représente un buffer de données simultanément en lecture et en écriture ($f4D$). Les cercles jaunes désignent des groupes de tâches et les flèches des dépendances. Les zones bleues englobantes représentent des champs ou fonctions de l'algorithme 2 et les zones vertes démarquent les deux parties de l'algorithme.

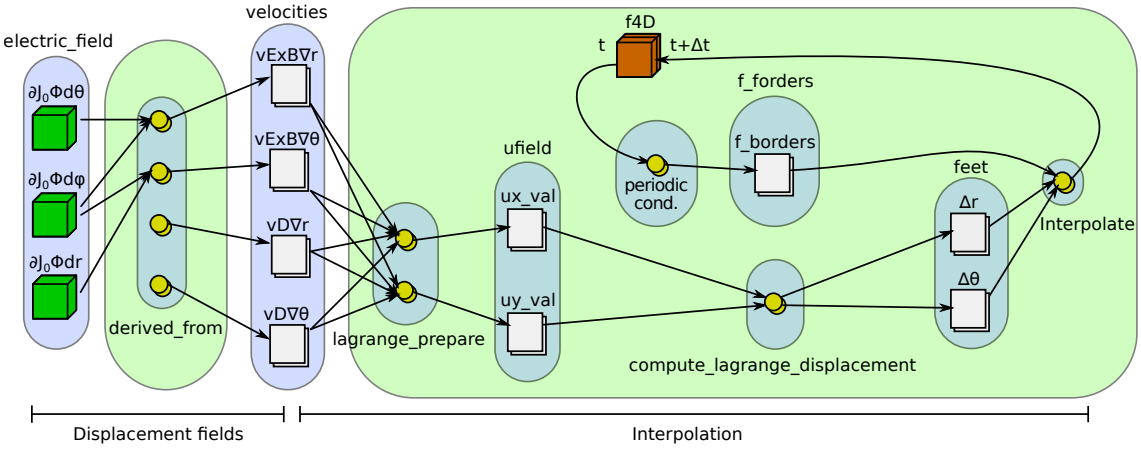

FIGURE 6.2 – Graphe de tâches de la variante Lagrange basé sur l'algorithme 3 opérant à la granularité du plan.

inhérent au calcul de chacun des plans quelque-soit la variante considérée. Cependant, ce n'est pas la seule forme de parallélisme supplémentaire que l'on peut extraire de l'advection 2D.

Réduction de la granularité Il est possible d'affiner encore la granularité des graphes de tâches en considérant les remarques des algorithmes 2 et 3 de la section précédente qui précisent le parallélisme de données inhérent à chacune des fonctions appelées. Construire une version à grain fin consiste à construire un graphe de tâches qui prend en compte ce parallélisme aujourd'hui non exploité par Gysela. La mise au point d'une telle version aurait de nombreux atouts : réduction de l'impact mémoire et amélioration potentielle des performances. En effet, lorsqu'un grand nombre de cœurs est utilisé (p. ex. Xeon Phi), chacun des cœurs opère sur au moins un plan de la fonction de distribution à la fois ainsi que sur quelques plans temporaires qui, cumulés, peuvent rapidement constituer un goulot d'étranglement pour des architectures à base de many-cœurs. De plus, pour une même quantité de données, cette version est capable d'offrir bien plus de parallélisme. Enfin, en travaillant simultanément sur un même plan, les cœurs pourraient utiliser plus efficacement les caches partagés à disposition, car le calcul de chaque plan nécessite de lire continuellement des mêmes constantes volumineuses. Néanmoins, élaborer d'une telle version est long et complexe car cela nécessite de modifier plus en profondeur le code de l'advection 2D, étant donné que ce cas d'utilisation n'a pas été conçu pour un parallélisme à grain fin. De ce fait, nous nous concentrons dans ce chapitre sur des versions fonctionnant à plus gros grain.

Objectifs Avant d'élaborer une telle version, il est pertinent d'évaluer l'approche sur un cas qui se rapproche le plus possible de la version de Gysela. Pour cela, les tâches sont agrégées en deux grands groupes démarqués par les zones vertes sur les figures 6.1 et 6.2. Ainsi, une seule tâche est soumise par groupe et par plan de la fonction de distribution traitée. Cette version référencée sous le nom de "gros grain" est à la base des versions HALLEY implémentées et présentées dans la section suivante. Notez que cette version n'est pas totalement équivalente à celle présente dans Gysela. En effet, rappelons que Gysela entrelace le calcul des champs de gradients avec la préparation de l'interpolation ligne par ligne. Cette dernière version, plus efficace en termes de localité temporelle, est référencée sous le nom de "grain moyen".

6.1.4 Description Comet de l'advection 2D

Les deux variantes Splines et Lagrange de l'advection 2D peuvent être exprimées par deux assemblages HALLEY distincts similaires, illustré à la figure 6.3, qui partagent une même partie du code (composants). Chacun des assemblages contient cinq instances de composants, une section dataflow et deux instances de métatâches incluses dans la section. Chacune de ces instances (composants et métatâches) est de type différent. Décrivons tous d'abord les unités présentes dans l'assemblage.

Composants Les deux assemblages exploitent six types de composants différents [3] : `2D-advection`, `IO-Manager`, `Constants`, `Grad-Task-Impl`, `Spline-Task-Impl` et `Lagrange-Task-Impl`. `2D-advection` est un composant visant à contrôler l'ensemble de l'assemblage (c.-à-d. l'expérience) : il déclenche le chargement des données, les envoie à la section, contrôle la section et récupère les données et répète l'exécution de la section un nombre fixe de fois (réalisé uniquement par le prototype pour mesurer les temps). `IO-Manager` est un composant permettant de charger un jeu de données de Gysela au format HDF5 (nécessaire uniquement pour faire fonctionner ce prototype à partir de données réelles issues de Gysela). `Constants` est un composant qui permet de récupérer de nombreuses constantes prédéfinies dans Gysela telles que la taille des données ou l'état de configuration de la simulation. Les composants `Grad-Task-Impl`, `Spline-Task-Impl` et `Lagrange-Task-Impl` visent à fournir une mise en œuvre des métatâches respectives `Grad`, `SplineInterpolation` et `LagrangeInterpolation`.

Métatâches Les deux assemblages ont recours à trois types de métatâches : `Grad`, `SplineInterpolation` et `LagrangeInterpolation`. `Grad` vise à définir comment soumettre les tâches du calcul des gradients. Cette métatâche détient trois ports d'entrées correspondant aux champs électromagnétiques et quatre ports de sorties relatifs au champ de vélocité et à recours au langage d'alignement `align`. `SplineInterpolation` et `LagrangeInterpolation` sont deux métatâches utilisées pour définir respectivement la soumission des tâches issues des variantes d'interpolation Splines et Lagrange. Chacune de ces deux métatâches possède quatre ports d'entrées associés au champ de gradients, un port d'entrée-sortie pour la fonction de distribution et utilise le langage de relation `identity`. Leur description est quasiment identique, mais leur mise en œuvre diffère.

Assemblage La figure 6.3 présente la structure de l'assemblage HALLEY évalué dans ce chapitre pour la variante Splines. L'assemblage de la variante Lagrange peut être obtenu en remplaçant la zone verte de l'assemblage par celle présente à droite de la figure.

Notez que l'assemblage utilise des données temporaires (T_0 à T_3) pour stocker les champs de vélocités. En effet, ces champs ne sont utilisés que par la section. La stratégie d'allocation conventionnelle mise en place dans HALLEY est d'allouer un buffer de données temporaires dès qu'il est nécessaire (c.-à-d. une tâche opérant dessus est soumise). Cependant, allouer l'ensemble de la donnée peut se révéler trop coûteux étant donné que la taille de ces buffers est similaire à celle décrivant la fonction de distribution, généralement très volumineux (c.-à-d. augmentation de la consommation mémoire par un facteur cinq). Ainsi, les plans sont alloués et désalloués indépendamment, les uns après les autres, directement par les tâches

3. Les assemblages ont recours à quatre types de composants partagés. Chaque assemblage utilise cinq composants distincts (un composant de chaque assemblage n'est pas partagé) et ce qui signifie que six composants sont implémentés.

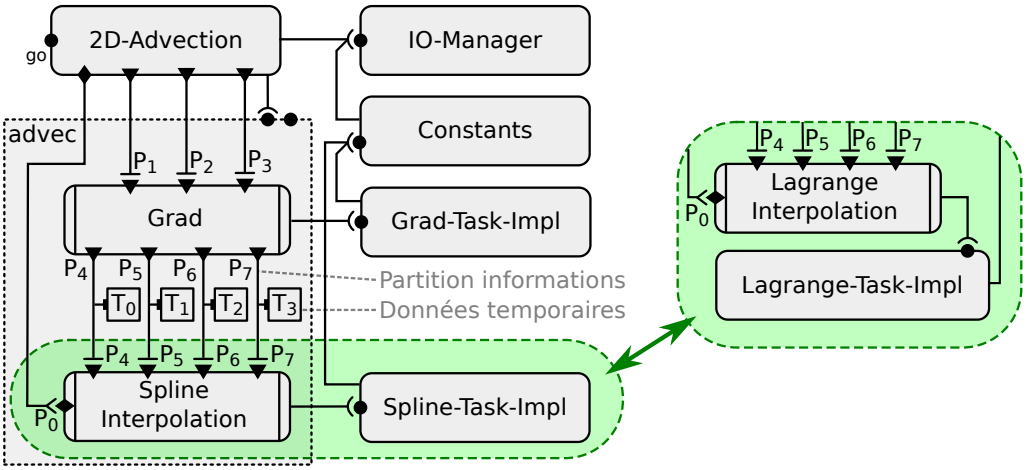

FIGURE 6.3 – Représentation graphique de l'assemblage HALLEY de la variante Splines opérant à la granularité du plan. La zone verte désigne une partie de l'assemblage remplaçable par d'autres composants afin d'obtenir la variante Lagrange.

(c.-à-d. les tâches écrivant les données allouent un plan et celles qui lisent les données les désallouent).

Gestion de variantes Pour passer d'une variante à une autre, les instances de composants et de métatâches spécifiques à une variante, d'un assemblage sont remplacées par de nouvelles. Plus particulièrement, c'est l'instance de la métatâche d'interpolation et l'instance de composant responsable de sa mise en œuvre qui sont remplacées (le type de l'instance).

6.1.5 Récapitulatif des versions

Niveaux de granularité Trois niveaux de granularité sont présentés dans les sections précédentes :

- gros grain : les traitements sont effectués à l'échelle du plan uniquement ;
- grain moyen : un entrelacement des codes est réalisé à l'échelle de la ligne entre le calcul des gradients et l'interpolation, mais l'interpolation est calculé au niveau du plan ;
- grain fin : l'ensemble des traitements sont effectués à l'échelle de blocs de cellules suffisamment petits pour permettre un entrelacement de code et suffisamment gros pour ne pas engendrer de surcoûts significatifs.

Versions et variantes Parmi l'ensemble des versions présentées pour chacune des variantes, on en distingue de trois sortes :

- les *versions de référence* présentées dans la section 6.1.2 opérant à grain moyen ;
- les *versions à base de tâches* décrites en section 6.1.3 opérant à gros grain, basées sur les *versions de référence* ;

		Variantes	
		Splines	Lagrange
Versions	Référence	Grain moyen	Grain moyen
	Intermédiaire	Gros grain	Gros grain
	COMET	Gros grain	Gros grain

TABLE 6.1 – Récapitulatif de l'ensemble des versions de l'advection 2D implémentées et évalués dans ce chapitre et granularité employée pour chacune d'entre elles.

— les *versions* COMET basées sur les *versions à base de tâches*.

Dans ce chapitre, la description des versions à base de tâches sert uniquement à expliquer comment introduire des tâches dans l'advection 2D pour aider à comprendre les versions COMET. Ainsi, les versions à base de tâches ne sont pas évaluées dans ce chapitre.

Afin de comparer des versions opérant à une même granularité, des *versions intermédiaires* sont introduites et évaluées en section 6.3.3. En effet, on peut observer que les versions de référence emploient une granularité moyenne alors que les versions COMET opèrent à gros grain. Les versions intermédiaires introduites sont une adaptation des versions de référence qui opèrent uniquement à gros grain.

Récapitulatif des versions Le tableau 6.1 récapitule l'ensemble des versions implémentées et évaluées dans ce chapitre. Pour chacune des versions et des variantes, il précise quelle est la granularité employée.

6.2 Évaluation des propriétés de génie logiciel

Cette section évalue les propriétés de génie logiciel de COMET sur le cas de l'advection 2D. Plus précisément, elle analyse la séparation des préoccupations, la flexibilité et la réutilisation du code des assemblages COMET détaillées en section 6.1.4 par rapport aux versions de référence présentées en section 6.1.2.

6.2.1 Séparation des préoccupations

Nous avons vu dans les sections 6.1.1 et 6.1.2 que l'advection 2D est décomposée en deux parties distinctes : le calcul des gradients et l'interpolation. Le calcul de l'interpolation peut être calculé de plusieurs manières laissant la place à deux variantes algorithmiques : Splines et Lagrange.

Mise en œuvre dans les versions Comet COMET permet de séparer les différentes parties dans des unités logicielles distinctes décrivant explicitement leurs dépendances. En effet, le calcul des gradients est réalisé par une métatâche dédié (ainsi qu'un composant associé). Il en est de même pour le calcul de la variante

Spline et celui de la variante Lagrange qui ont tous les deux des métatâches et composants spécifiques. Les dépendances des métatâches et les composants responsables de leurs mises en œuvre disposent d'interfaces décrivant l'ensemble de leurs dépendances. Leurs interfaces sont suffisamment généralistes (ports de données transmettant des buffers multidimensionnels partitionnés en blocs réguliers) pour qu'une variante puisse être remplacée par une autre. Le fonctionnement de ce remplacement est visible à la figure 6.3.

Bénéfices Une telle décomposition permet de séparer les préoccupations de l'application. Par exemple, une personne spécialiste de l'interpolation par splines ou de l'interpolation à base de polynômes de Lagrange n'a pas à se préoccuper du calcul des gradients et inversement. Dans le cas contraire, la multiplication des préoccupations dans une même portion du logiciel nuit à sa maintenabilité. La difficulté à maintenir l'application peut augmenter exponentiellement en fonction du nombre de variantes considérées et le nombre de parties couplées et devenir un problème majeur à l'échelle de l'application entière. En effet, le couplage des différentes parties avec une autre peut dépendre des variantes considérées (p. ex. version à grain fin) et le nombre de combinaisons possibles correspond au produit du nombre de variantes de chacune des parties.

Limites des versions Comet actuelles La gestion des données temporaires mise en place manuellement dans les versions Comet réalisées renforce la dépendance entre les deux métatâches composées. En effet, les données temporaires sont actuellement détruites par les tâches issues de la métatâche d'interpolation et donc ces données ne peuvent pas être réutilisées ensuite par d'autres métatâches pouvant être ajoutées dans l'assemblage (p. ex. métatâche d'analyse des données). Or la mise en œuvre des composants d'une application ne devrait pas dépendre de l'assemblage, sous peine de limiter les modifications possibles de l'assemblage et donc de dégrader la maintenabilité de l'ensemble. De plus, il est important de noter que le repartitionnement des données temporaires allouées par morceaux (non implémenté actuellement) peut introduire des copies supplémentaires (obligatoires) et donc un surcoût devant être étudié. La source de ce problème provient de la mise en œuvre Halley qui ne supporte pas l'allocation de données temporaires par morceaux actuellement [4]. La mise en place de mécanisme d'allocation par morceaux efficace et généraliste est actuellement un problème ouvert et l'impact sur le modèle Comet est pour le moment inconnu.

Informations complémentaires Afin de préserver la séparation des préoccupations, il est important de noter que deux types de métatâches ont été définis, car une modification d'une des métatâches ne devrait pas avoir d'impact sur les autres.

4. au lieu d'allouer un buffer de données en entier et de le partitionner ensuite, des fragments sont alloués un à un ou en groupe afin de réduire la quantité de mémoire nécessaire à l'exécution de l'application, mais aussi afin de permettre une réutilisation des fragments alloués et ainsi améliorer l'utilisation du cache tout en limitant les coût relatifs de la pagination

Version	Variante	Lignes de code			Lignes partagées entre les variantes
		Impl.	En-tête	Déf.	
Référence	Splines	953	464	-	426 (dont 426 d'en-têtes)
	Lagrange	758	479	-	
Comet	Splines	1244	586	181	1257 (dont 532 d'implémentation, 571 d'en-têtes et 154 de définitions)
	Lagrange	866	586	181	

Table 6.2 – Nombre de lignes de code C++ (hors espaces, commentaires et assertions) et de lignes réutilisées entre les deux variantes Splines et Lagrange des versions de référence et Comet. Les en-têtes comprennent les définitions des structures de données ainsi que les interfaces de composants. Les définitions comprennent les fichiers de description des métatâches et de l'assemblage. Les abréviations *Impl.* et *Déf.* sont utilisées pour désigner respectivement les mots *Implémentation* et *Définition*. Le nombre de lignes de code n'inclus pas les greffons (réalisés par les experts).

C'est en particulier le cas si une version à grain fin est mise au point pour une des variantes.

6.2.2 Partage de code entre les variantes

Le tableau 6.2 présente le nombre de lignes de code et de lignes réutilisées entre les deux variantes Splines et Lagrange des versions de référence et Comet. Le code des greffons devant être écrits par des experts n'est pas compris dans le nombre de lignes.

Globalement, les versions Comet utilisent une quantité de lignes de code plus grande que celles de référence : de 42% dans le cas Splines à 32% dans le cas Lagrange. Cette différence provient principalement de la description des composants (implémentation), des interfaces (en-têtes), des métatâches (définitions), de l'assemblage (définitions) nécessaire pour faire fonctionner la version Comet.

Toutefois, on constate que la version Comet permet de partager beaucoup de lignes de code entre les versions (2,95 fois plus). En particulier, la version Comet permet de réutiliser simplement une partie de l'implémentation, le calcul des gradients, ce qui n'est pas le cas de la version de référence. En effet, bien que la version Comet ajoute respectivement 291 et 108 lignes de code correspondant à l'implémentation par rapport à la version de référence sur les variantes Splines et Lagrange, 532 lignes d'implémentation sont partagées entre les deux variantes. Dans l'ensemble, les versions Comet contiennent plus de code réutilisable que celles de référence : de 63% (Splines) et 77% (Lagrange) avec Comet à 30% (Splines) et 34% avec la version de référence.

6.2.3 Remplacement de variantes

Le remplacement d'une variante par une autre est traité de manière totalement différente entre les deux versions. D'une part, la version de référence implémente les

deux variantes à l'aide de deux codes monolithiques développés séparément (bien que la partie correspondante au calcul des gradients soit similaire). D'autre part, la version COMET sépare dans des composants distincts les parties réutilisables (calcul des gradients, lecture de fichiers HDF5, accès aux constantes, etc.) des parties spécifiques à chaque variante (interpolation). Chacune des partie spécifique correspondant aux deux variantes est implémentée dans des composants et métatâches dédiés. Ainsi, le passage d'une variante à une autre nécessite, dans la version COMET, de changer uniquement deux identifiants dans l'assemblage (c.-à-d. deux lignes à modifier) : le type de métatâche et le type du composant qui l'implémente.

6.3 Évaluation des performances

Cette section évalue les performances de COMET, plus précisément sa mise en œuvre HALLEY, sur le cas de l'advection 2D. Pour cela, elle compare les performances assemblages décrites en section 6.1.4 avec des versions de référence présentées en section 6.1.2 en faisant varier le nombre de cœurs.

6.3.1 Méthode et configuration expérimentales

Architecture et configuration matérielle Les expériences sont réalisées sur la machine de calcul Brunch du LIP (Lyon, France). Brunch dispose de 4 sockets de 24 cœurs Intel Xeon E7-8890 v4 (architecture Broadwell, processeur cadencé à 2,20 GHz). Une seule socket (ainsi qu'un seul banc mémoire) est utilisée afin de pallier les effets NUMA. Il est important de noter que *Turbo-Boost* et les *C-States* sont désactivés sur cette machine afin d'améliorer l'analyse des performances et leur reproductibilité.

Environnement logiciel Le compilateur utilisé pour l'ensemble des versions évaluées est *Intel ICC 2017*. Le support exécutif OpenMP utilisé est KOMP. L'allocateur *Jemalloc* [48] [5] remplace celui par défaut (glibc), car il passe mieux à l'échelle en pratique et offre de meilleures performances.

Protocole expérimental Les mesures sont répétées 21 fois (1+20 itérations). La première mesure n'est pas retenue. Les valeurs médianes sont affichées. Les barres d'erreurs correspondent au premier et dernier quartile. Les temps de complétion idéaux sont calculés à partir du meilleur temps séquentiel divisé par le nombre de cœurs utilisé. Le jeu de données traitées est directement extrait de Gysela et a une taille de 512x513x64x16 (soit approximativement 2 Go).

5. http://jemalloc.net

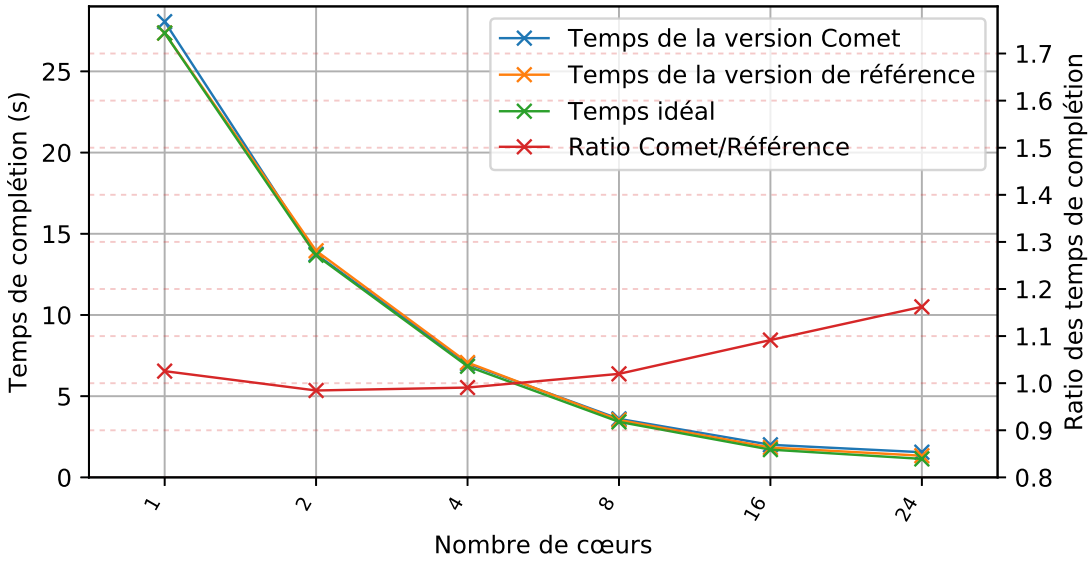

FIGURE 6.4 – Temps de complétion (absolu et relatif) de la variante Splines en fonction du nombre de cœurs utilisés obtenus respectivement avec la version de référence et avec la version COMET sur la machine de calcul Brunch. Le ratio est calculé par rapport à la version de référence.

6.3.2 Comparaison des performances

Cette section compare les performances de la version COMET par rapport à la version de référence décrite respectivement dans les sections 6.1.2 et 6.1.4 sur les deux variantes Splines et Lagrange de l'advection 2D.

Analyse de la variante Splines La figure 6.4 présente les temps de complétion obtenus sur la variante Splines des deux versions analysées (référence et COMET) de 1 à 24 cœurs (de la machine Brunch) ainsi que le ratio des temps de la version COMET par rapport à la version de référence.

On peut voir tout d'abord que toutes les versions passent bien à l'échelle jusqu'à 24 cœurs atteignant une accélération de 18,1 dans le cas de la version COMET et de 20,5 dans le cas de la version de référence. Les temps de complétion des deux versions sont assez proches. Toutefois, on peut observer que la version COMET est très légèrement plus lente que la version de référence en séquentiel (d'un facteur 1,025) et devient de plus en plus lente en parallèle au fur et à mesure qu'un nombre grandissant de cœurs est utilisé atteignant un facteur de 1,162 sur 24 cœurs.

Comme nous le verrons en section 6.3.3, ce surcoût croissant en fonction du nombre de cœurs provient du manque d'entrelacement entre le calcul des gradients et l'interpolation par splines. En effet, alors que la version COMET opère uniquement à la granularité du plan, la version de référence entrelace les deux codes couplés de manière à améliorer la localité temporelle et donc utilise plus efficacement les caches.

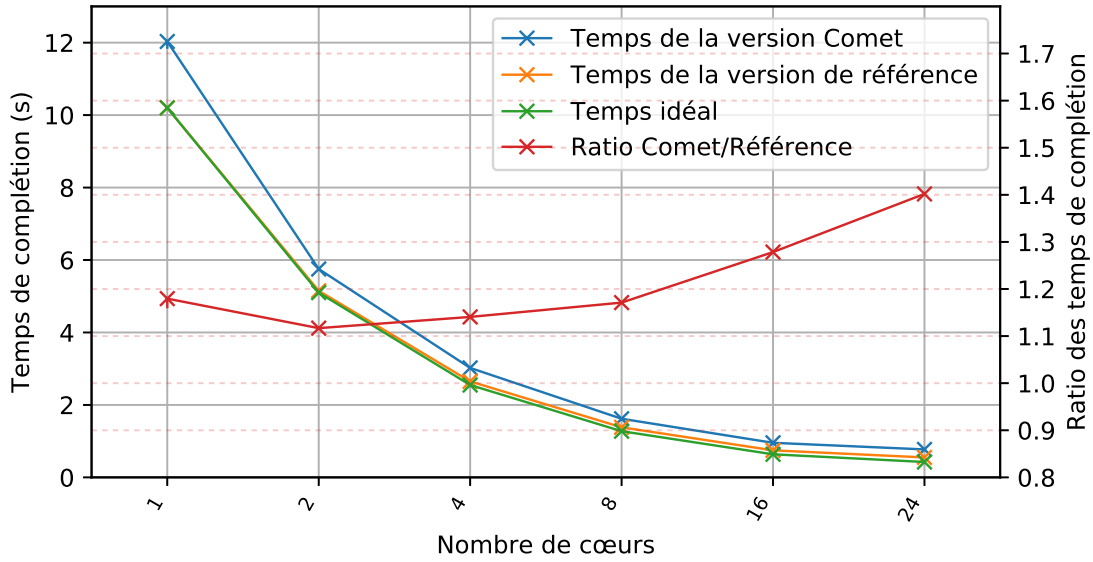

FIGURE 6.5 – Temps de complétion (absolu et relatif) de la variante Lagrange en fonction du nombre de cœurs utilisés obtenus respectivement avec la version de référence et avec la version COMET sur la machine de calcul Brunch. Le ratio est calculé par rapport à la version de référence.

Analyse de la variante Lagrange La figure 6.5 présente les temps de complétion obtenus sur la variante Lagrange des deux versions analysées (référence et COMET) de 1 à 24 cœurs (de la machine Brunch) ainsi que le ratio des temps de la version COMET par rapport à la référence.

Tout comme pour la variante Splines, les versions de la variante Lagrange passent assez bien à l'échelle : sur 24 cœurs la version COMET obtient une accélération de 15,6, tandis que la version de référence atteint 18,5. Le temps de la version COMET est significativement plus grand que celui de la référence. En effet, en séquentiel, COMET est 1,180 fois plus lent que la référence. Sur deux cœurs, ce facteur chute à 1,117 puis ne cesse d'augmenter au fur et à mesure que le nombre de cœurs utilisé grandit, atteignant un facteur de 1,402 sur 24 cœurs.

On peut distinguer deux sources de surcoûts : l'une impacte l'exécution séquentielle et l'autre grandit en fonction du nombre de cœurs utilisés.

Dans le cas d'une exécution séquentielle, le support exécutif sous-jacent (KOMP) est configuré pour exécuter les tâches dans l'ordre où elles sont soumises (ordre d'exécution toujours correct). Or la soumission des tâches est groupée par métatâche : l'ensemble des tâches de calcul des gradients est soumis avant l'ensemble des tâches d'interpolation. Bien que ce choix permette de réduire drastiquement les surcoûts de l'ordonnancement, il a un impact prohibitif sur ce cas d'utilisation. En effet, l'ensemble des données temporaires sont allouées et écrites, puis intégralement lues et libérées (dans un second temps). Par conséquent, le coût des allocations est plus élevé, car l'allocateur (ici Jemalloc) ne peut pas réutiliser des buffers précédemment

Version / Variante	Nombre de cœurs					
	1	2	4	8	16	24
Référence / Splines	347	353	364	388	434	481
COMET/ Splines	2 096	366	377	403	463	523
Référence / Lagrange	343	344	347	354	366	379
COMET/ Lagrange	1 825	352	360	377	412	446

TABLE 6.3 – Nombre de défauts de pages (en milliers) obtenus sur les versions de référence et COMET en fonction du nombre de cœurs utilisé sur Brunch.

alloués au sein d'une même itération, ce qui a pour effet d'augmenter les surcoûts liés à la pagination[6]. Cet effet affecte aussi la variante Splines, mais il est relativement moindre étant donné que le temps de complétion de la variante Splines est plus conséquent et que cette dernière est moins sensible aux effets de caches. Une étude du nombre de défauts de pages des différentes versions en fonction du nombre de cœurs utilisé présenté par le tableau 6.3 confirme la présence de cet effet : le nombre de défauts de pages des versions COMET est très élevé dans le cas séquentiel.

Lorsque le nombre de cœurs grandit, le manque d'entrelacement de la version COMET introduit un surcoût considérable. Bien qu'il soit visible sur les deux variantes, l'efficacité de la variante Lagrange par rapport à la variante Splines accentue les effets provenant de cette optimisation.

6.3.3 Comparaison de versions équivalentes

Méthode de comparaison Une *version intermédiaire* a été mise au point afin de confirmer l'hypothèse selon laquelle la granularité est responsable de la différence des temps de complétion entre la version COMET et la version de référence. Cette version, bien plus proche de celle de COMET, est obtenue en adaptant l'entrelacement effectué entre le calcul des gradients et l'interpolation de manière à opérer à la granularité du plan. Elle permet d'évaluer les surcoûts introduits par HALLEY plus justement, en ne considérant pas les surcoûts engendrés par un choix non optimal de la granularité de la version COMET. L'analyse et la conception d'une version COMET opérant à grain plus fin, aujourd'hui non réalisée, permettrait de comparer plus précisément les surcoûts engendrés par COMET et la mise en œuvre HALLEY.

Les figures 6.6 et 6.7 comparent les temps de complétion de la version COMET par rapport à la version intermédiaire (issue de la version de référence, mais opérant uniquement à la granularité du plan) de 1 à 24 cœurs (de Brunch) et avec les variantes Splines et Lagrange.

On peut observer que les versions COMET sont quasiment aussi rapides que les versions intermédiaires, excepté dans le cas séquentiel (dû au choix fait par le support exécutif KOMP énoncé précédemment). En effet, la version COMET est au

6. Augmentation des défauts de pages auquel est ajouté un coût d'initialisation des pages à zéro (sur les noyaux Linux récents par défaut) effectuée pour des raisons de sécurité telles que l'isolation des processus.

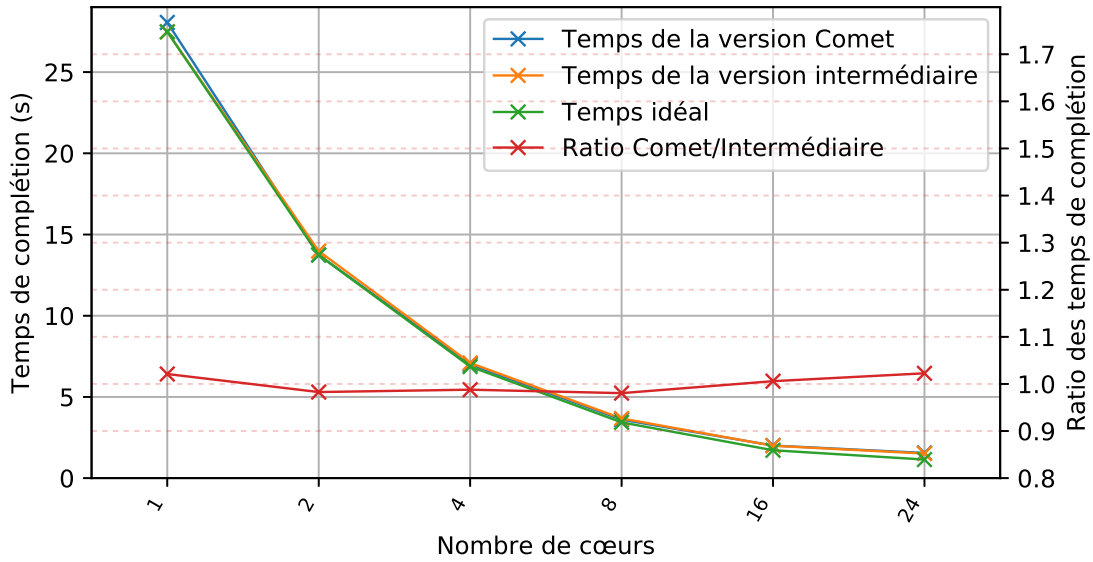

FIGURE 6.6 – Temps de complétion (absolu et relatif) de la variante Splines en fonction du nombre de cœurs utilisés obtenus respectivement avec la version de référence et avec la version COMET sur la machine de calcul Brunch. Le ratio est calculé par rapport à la version de référence.

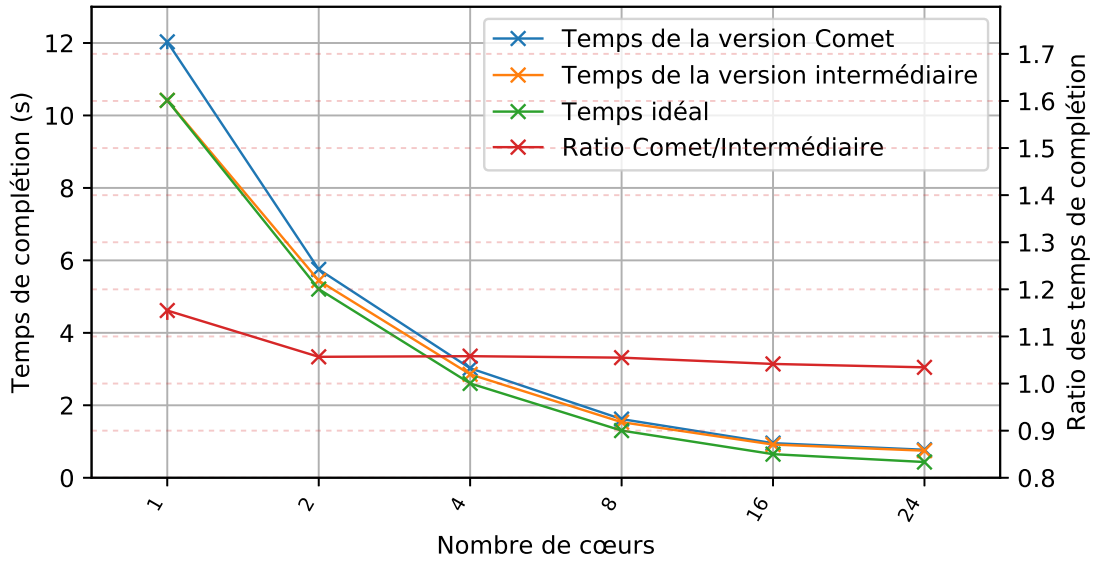

FIGURE 6.7 – Temps de complétion (absolu et relatif) de la variante Lagrange en fonction du nombre de cœurs utilisés obtenus respectivement avec la version de référence et avec la version COMET sur la machine de calcul Brunch. Le ratio est calculé par rapport à la version de référence.

plus 1,023 fois plus lente dans le cas de la variante Splines et au plus 1,058 dans le cas de la variante Lagrange. On peut donc déduire que COMET n'introduit pas de surcoût significatif à l'exécution sur le cas d'utilisation d'advection 2D lorsque la granularité employée est au niveau du plan.

6.4 Discussion

Résultats en termes de génie logiciel Les résultats portant sur l'analyse des propriétés de génie logiciel offertes par COMET montrent que le modèle permet de séparer les deux parties de l'advection 2D de Gysela dans des composants distincts améliorant la maintenabilité du code. De plus, les résultats obtenus montrent que même si les versions COMET sont plus volumineuses en termes de nombre de lignes de code que les versions de référence, des portions de l'implémentation peuvent être facilement partagées entre les variantes améliorant la réutilisation des codes. La séparation des préoccupations permise par COMET, combinée à une réutilisation accrue de portion logicielle, permet de remplacer facilement une variante de l'interpolation par une autre en adaptant l'assemblage. Toutefois, l'indépendance des codes couplés est affectée par le support manuel des données temporaires dans le cas de la version actuelle de HALLEY.

Résultats en termes de performance Les résultats portant sur l'analyse des performances montrent qu'à la granularité d'un plan, les versions COMET n'introduisent pas de surcoûts significatifs (excepté en séquentiel). Cependant, les versions opérant à une granularité plus petite offrent de meilleures performances grâce à l'entrelacement ligne par ligne du calcul des gradients avec celui de l'interpolation. La conception et l'analyse des performances d'une version COMET travaillant au même grain que la version de référence restent aujourd'hui à étudier.

Discussion sur les résultats En combinant l'ensemble des résultats obtenus dans ce chapitre, nous pouvons déduire que COMET favorise une amélioration de la séparation des préoccupations et de la réutilisation de portion logicielle tout en offrant des performances similaires par rapport aux versions intermédiaires. De plus, nous pouvons aussi confirmer que l'entrelacement de codes HPC à grain fin permet d'obtenir de meilleures performances sur un cas d'utilisation réel tel que l'advection 2D. Néanmoins, une étude approfondie à grain plus fin (grain moyen) est nécessaire. Mais, pour cela, certains problèmes tels qu'une meilleure gestion des données temporaires doivent être traités.

Gestion des données temporaires Le support avancé des données temporaires dans HALLEY ou COMET est une des préoccupations nécessaires pour traiter entièrement le cas de l'advection 2D à grain moyen ou à grain fin : les données doivent être allouées et libérées morceau par morceau, et l'ordonnancement contrôlé pour ne pas dégrader les performances. Or, HALLEY alloue actuellement les buffers de

données temporaires entièrement lors du déclenchement de la section et libère les données lors sa terminaison, ce qui impacte négativement les performances, car il engendre un nombre de défauts de pages conséquents (effet visible dans le cas séquentiel). Afin de ne pas dégrader considérablement les performances, le prototype réalisé découpe le buffer de données en plan et permet aux concepteurs (responsables de l'implémentation des métatâches) d'allouer les plans uniquement avant d'écrire dedans et de les libérer juste après la lecture. Toutefois, le contrôle de l'allocation ne devrait pas être réalisé par les concepteurs de composant étant donné qu'il dépend de la structure de l'assemblage.

Il serait plus pertinent que HALLEY soit responsable de l'allocation des données temporaires. Pour cela, les greffons de données partitionnées doivent fournir des informations sur la manière d'allouer efficacement les morceaux de données temporaires à l'aide d'interfaces dédiées restant à définir. Chacun de ces morceaux de données temporaires correspond à un ou plusieurs fragments. Le choix de la granularité optimale des morceaux dépend de nombreux facteurs tels que les performances de l'allocateur ou encore la taille des caches (dépendant de l'architecture).

Ordonnancement de tâches dans le cas séquentiel Les problèmes survenant dans le cas séquentiel sont dus à un ordonnancement suivant l'ordre de soumission plutôt qu'un parcours en profondeur. Étant donné que les tâches sont soumises en groupe pour chacune des métatâches, cet ordonnancement conduit à une exécution inefficace dans le cas de l'advection 2D ainsi qu'à l'allocation de la majorité des données temporaires. Idéalement, le support exécutif devrait avoir connaissance des données temporaires afin d'adapter l'ordonnancement des tâches pour limiter l'impact sur les performances et sur la consommation mémoire. Cependant, il serait aussi préférable de soumettre les tâches dans un ordre qui favorise une exécution efficace, bien que cela soit difficile en pratique, car cela dépend de l'architecture considérée.

Compilation et optimisation des versions Notez que la version COMET est parfois légèrement plus rapide. Cela provient du compilateur choisi : le compilateur ICC d'Intel. En effet, les optimisations réalisées par ce compilateur peuvent varier significativement d'une version de code évaluée à une autre (le binaire généré dépend de plusieurs heuristiques sensibles à de nombreux paramètres). C'est en particulier le cas des versions COMET et intermédiaires (ainsi que celles de référence). Nous nous sommes donc évertués à ce que les binaires générés par le compilateur soit les plus équivalents possible en termes de performance en comparant des parties de code une à une de manière à avoir une différence de performance négligeable ($<5\%$). En outre, le changement du processus de compilation (utilisation de bibliothèques dynamiques dans le cas des versions COMET contre un exécutable monolithique autrement), de la structure du code (utilisation de tâches) et de l'accès aux constantes (composant dédié pour les versions COMET contre des variables globales sinon) sont des facteurs qui peuvent avoir un impact considérable sur le processus d'optimisation. Plus particulièrement, les optimisations issues de l'analyse du chevauchement des tableaux (*aliasing*) et la vectorisation automatique sont deux sources de variabilité

des codes que nous avons visé à maîtriser en précisant explicitement à ICC quelles optimisations doivent être réalisées.

6.5 Conclusion

Ce chapitre a évalué le modèle COMET, plus précisément sa mise en œuvre HALLEY, sur l'advection 2D, un cas d'utilisation réaliste tiré d'une sous-partie de Gysela. Il a tout d'abord rappelé la place de ce cas d'utilisation dans Gysela, décrit précisément son fonctionnement, les différentes versions implémentées et évoqué les motivations de l'utilisation de COMET dans ce cadre. Ce chapitre a ensuite évalué les propriétés relatives au génie logiciel apportées par COMET en termes de séparation des préoccupations, et de réutilisation de portions de codes. Puis, ce chapitre a évalué les performances des versions HALLEY en termes de temps de complétion et de passage à l'échelle. Pour cela, il a étudié le temps de complétion des versions HALLEY par rapport à des versions de référence écrites à la main. Des versions intermédiaires ont été introduites afin de comprendre la source des surcoûts engendrés par les versions HALLEY et s'il est possible de les supprimer. Enfin, ce chapitre s'est terminé par une argumentation laissant entrevoir les avantages et les limites de l'approche ainsi que des perspectives.

Les résultats obtenus dans ce chapitre sont plutôt positifs. En effet, il montre qu'à la granularité du plan, il est possible de mettre en œuvre une version COMET qui améliore la séparation des préoccupations et la réutilisation du code de l'advection 2D, simplifiant ainsi le remplacement de variantes, tout en étant aussi efficace qu'une version référence opérant à cette même granularité. Cependant, bien que ces expériences montrent la faisabilité de l'approche sur un cas d'utilisation réel, ces expériences restent préliminaires : une étude plus approfondie sur des versions à grain moyen est requise pour confirmer les résultats obtenus.

À terme, l'élaboration d'une version à grain fin pourrait offrir de bonnes propriétés de génie logiciel tout en améliorant les performances de Gysela. Une étude préliminaire a été réalisée afin d'évaluer certaines limitations de l'approche (coût des tâches et de l'ordonnancement et de la gestion des données temporaires à cette échelle). Cette étude se concentre sur la version Lagrange de référence et la modifie afin d'adopter une parallélisation à grain fin basée sur des tâches. Les premiers résultats obtenus avec le compilateur GCC 6.4 sont pour le moment positifs, montrant un léger gain de l'ordre de 10%. Le gain de performance semble dû à une meilleure utilisation du cache, recouvrant les surcoûts engendrés par l'approche à cette échelle, mais doivent être confirmés par une analyse détaillée.

Chapitre 7

Conclusion et perspectives

Contents

7.1 Conclusion

Afin de maintenir plus facilement les applications HPC qui s'exécutent sur des architectures matérielles complexes et hautement parallèles, cette thèse a pour but de combiner simultanément les approches à base de graphes de tâches et les modèles de composants afin de proposer une nouvelle forme de composition. L'objectif est de pouvoir exprimer des codes parallèles indépendamment et de les assembler de manière à ce qu'ils puissent exploiter efficacement les ressources disponibles. Pour cela, nous nous sommes intéressé aux mécanismes de composition de codes parallèles des modèles de composants logiciels existant dans le cadre des applications de calcul

de haute performance (HPC). Nous avons analysé la faisabilité d'une composition de codes HPC ayant recours à l'ordonnancement de graphes de tâches et proposé le modèle de composants COMET intégrant une nouvelle forme de composition pour supporter ces types de codes. Nous avons montré l'applicabilité de la méthode à travers l'élaboration de la mise en œuvre HALLEY du modèle COMET et évalué cette approche sur des cas d'utilisations synthétiques ainsi que sur une sous-partie de l'application de simulation gyrocinétique de plasmas par confinement magnétique pour la fusion Gysela.

Contexte et travaux connexes La première partie de cette thèse, développée dans le chapitre 2, a étudié le contexte et l'état de l'art des approches relatives au calcul de haute performance. Pour cela, le chapitre s'est tout d'abord intéressé aux architectures matérielles HPC et a mis en évidence leur complexité et leur variabilité. Il a ensuite évoqué les enjeux relatifs à la conception d'applications numériques et a plus particulièrement examiné le cas de l'application Gysela. Ce chapitre a mis en évidence des compromis entre les performances et la maintenance des applications devant être réalisés par les concepteurs. Il a ensuite examiné les formes de parallélisme et d'interactions utilisées pour concevoir des applications HPC et montré la diversité des paradigmes de programmation existants. Enfin, ce chapitre a étudié les méthodes utilisées pour concevoir des applications et comment les maintenir en tirant parti des modèles de composants. En particulier, ce chapitre a analysé le problème de la composition de codes HPC et montré les bénéfices d'une composition de codes HPC ayant recours à l'ordonnancement de graphes de tâches avant d'énoncer les objectifs de cette thèse.

Comet Le chapitre 3 a présenté le modèle de composants COMET dédié au calcul haute performance qui supporte l'ordonnancement de graphes de tâches et qui cible en particulier la mémoire partagée. Ce modèle vise à assembler simplement des codes parallèles indépendants, tout en permettant une exécution efficace. Pour cela, COMET est basé sur le modèle de composants existant L^2C qui offre deux formes de composition à travers des connexions de type *use-provide* et *MPI*. COMET étend L^2C en supportant une nouvelle forme d'interactions entre les composants à base de flux de données constituée d'unités d'ordonnancement appelées métatâches qui manipulent des données partitionnées. Cette extension impacte les ports *use* et *provide* qui doivent spécifier explicitement s'ils sont utilisables dans le contexte d'une tâche (c.-à-d. *task-safe*).

Halley Le chapitre 4 a présenté HALLEY, une mise en œuvre modulaire de COMET basée simultanément sur les modèles de composants et l'ingénierie dirigée par les modèles, construite au-dessus de L^2C et d'OpenMP. Le chapitre a détaillé le fonctionnement de HALLEY et décrit comment l'étendre à l'aide de greffons de trois types différents (greffons de données, de données partitionnées, de repartitionnement et de relations). À travers HALLEY, le chapitre montre ainsi la possibilité de mettre au point une mise en œuvre extensible de COMET.

Évaluation Les chapitres 5 et 6 ont évalué COMET et plus spécifiquement la mise en œuvre HALLEY sur respectivement deux types de cas d'utilisations. Le chapitre 5 a étudié la séparation des préoccupations et les performances de cas d'utilisations synthétiques tels que des cas "embarassingly parallel", des stencils et un cas faisant intervenir une transposition. Pour cela, deux variantes ont été comparées dans chacun des cas d'utilisations : une variante OpenMP écrite manuellement et une autre utilisant HALLEY. Ces cas d'utilisations sont principalement limités par le débit mémoire à l'exception du dernier entrelaçant des opérations limitées par la vitesse de calcul et par la mémoire (c.-à-d. latence et débit). Les résultats ont montré que COMET est capable de séparer des portions de codes dans des unités de compositions développées indépendamment et ensuite composées de manière à utiliser efficacement les ressources disponibles à l'exécution (p. ex. amélioration de la localité temporelle des calculs, recouvrement d'opérations limitées par la mémoire avec des opérations limitées par le calcul). Le chapitre 6 a examiné l'utilisation de HALLEY sur l'advection 2D, un cas d'utilisation réaliste tiré d'une partie de Gysela. Plus précisément, ce chapitre a présenté le cas d'utilisation puis évalué des propriétés relatives au génie logiciel apportées par COMET (séparation des préoccupations et de réutilisation de code) ainsi que les performances des versions COMET par rapport à des versions de référence. Il montre qu'à gros grain, il est possible de mettre en œuvre une version COMET qui améliore la séparation des préoccupations et la réutilisation du code de l'advection 2D, simplifiant ainsi le remplacement de variantes, tout en étant aussi efficace qu'une version référence opérant à cette même granularité.

Conclusion Pour résumer, cette thèse, à la croisée de plusieurs domaines de recherche, propose une méthode visant à résoudre de bout en bout des problèmes relatifs à la composition de codes HPC : de l'étude de la faisabilité par l'élaboration d'un modèle de composants à son application sur un cas d'utilisation réel telle qu'une sous-partie de l'application HPC de production Gysela.

7.2 Perspectives

L'élaboration du modèle COMET et de sa mise en œuvre HALLEY montre la faisabilité de l'approche. Les évaluations sur des cas d'utilisation incluant une sous-partie de l'application Gysela révèlent la pertinence de cette méthode. Cependant, il reste, à ce jour, plusieurs problèmes ouverts à traiter.

Extension de Halley et analyse des limitations de Comet Bien que la mise en œuvre HALLEY ait été conçue de manière à pouvoir supporter tout type de structure de données telle que les maillages réguliers et irréguliers, seuls des cas réguliers (p. ex. grilles multidimensionnelles, planes ou toriques) ont été analysés. Il serait donc intéressant d'étudier la capacité de HALLEY à supporter de telles structures de données (et d'examiner les répercussions sur COMET en conséquence)

ainsi que d'évaluer les performances obtenues. De même, il serait aussi opportun de vérifier si HALLEY supporte des opérations de repartitionnement sur de telles structures de données afin d'évaluer l'extensibilité de l'approche.

Bien que ces expériences montrent la faisabilité de l'approche sur un cas d'utilisation réel tel que l'advection 2D à la granularité du plan, une étude approfondie de versions COMET opérant à la même granularité que Gysela est requise pour confirmer les résultats obtenus jusque-là sur ce cas d'utilisation. À plus long terme, il semble pertinent d'analyser les limites de l'approche sur le cas de l'advection à grain fin. En effet, ce cas d'utilisation est suffisamment complexe pour être représentatif d'une gamme non négligeable d'applications HPC.

Seuls quelques cas de langages de relations simples ou spécialisés ont été mis au point. La conception de nouveaux langages de relations, permettant de supporter de nouveaux patrons de relations tels que des cas de réduction ou stencils plus complexes, reste à étudier. Étant donné que la version actuelle du modèle COMET peut receler des limitations qui empêchent de supporter une gamme plus large de patrons de relations, ces limitations doivent être analysées et le modèle étendu si besoin. Afin de supporter des cas de relations spécifiques à une application et/ou peu fréquents, il est possible d'étendre HALLEY en permettant aux concepteurs d'applications de fournir un composant spécifiant les relations dans un langage de bas niveau tel que C++. De tels composants ont été mis au point dans des versions préliminaires de COMET et ont montré l'applicabilité de cette méthode [37]. Cependant, ces composants sont dépendants d'autres greffons (p. ex. greffons de données partitionnées). Il serait donc pertinent d'étudier comment les intégrer plus simplement. L'existence d'un modèle de relation généraliste, de plus haut niveau, est un problème ouvert.

Gestion avancée des données temporaires Lorsque les buffers de données temporaires sont volumineux, il est opportun de les allouer et de les libérer morceau par morceau afin de limiter la consommation mémoire et d'améliorer la localité des opérations (données continuellement contenues dans les caches). Cependant, une allocation par morceaux ajoute de nouveaux problèmes : le découpage des données n'est pas transparent aux yeux des concepteurs d'application, car il impacte la contiguïté des données et donc la mise en œuvre des tâches et leur efficacité. Cela a donc un impact sur COMET et sa mise en œuvre HALLEY. De plus, la granularité des morceaux doit être contrôlée tout en respectant l'atomicité des fragments : si les morceaux sont trop grands, cela risque d'avoir un impact négatif sur les performances et la mémoire ; si les morceaux sont trop petits, l'allocateur mémoire risque de dégrader les performances globales et devenir un goulot d'étranglement. Enfin, les supports exécutifs doivent avoir connaissance de ces informations sans quoi ils risquent d'employer des stratégies d'ordonnancement inadéquates (p. ex. ordonnancement du graphe de tâches en largeur imposant l'allocation de la majeure partie des données temporaires). Idéalement, les supports exécutifs devraient même contrôler le cycle de vie des morceaux de données temporaires et adapter l'ordonnancement en conséquence, de manière à maximiser les performances tout en limitant l'occupation mémoire. Cependant, les approches HPC existantes ne supportent que partiellement

ces fonctionnalités.

Support des accélérateurs de calcul L'utilisation d'approches à base de tâches se révèle particulièrement prometteuse lorsqu'il s'agit d'exploiter efficacement des architectures matérielles hétérogènes, telles que celles disposant de GPU. Or, depuis quelques années, ces architectures peuvent être utilisées pour effectuer des calculs généralistes (GPGPU) afin d'accélérer certains types de codes HPC. Pour supporter ce type de matériel, une extension possible est de remplacer le port `compute` des métatâches par plusieurs interfaces : chaque interface est associée à un type d'unités de calcul spécifique (p. ex. CPU, GPU, FPGA, etc.) et peut être connectée à un composant responsable de la mise en œuvre d'une des variantes. L'ensemble des variantes peut être fourni à un support exécutif tel que StarPU capable de sélectionner automatiquement celles qui sont les plus adaptées à l'exécution afin d'exploiter le plus efficacement les ressources à disposition.

Support des architectures NUMA Bien que les supports exécutifs soient capables de supporter les architectures NUMA [115] contrairement à OpenMP [1], Halley ne supporte pas efficacement ces architectures, car il ne prend pas en compte l'affinité entre les données et les nœuds NUMA. Le support de l'affinité dans Halley peut être ajouté en spécifiant, pour chaque tâche, sur quel nœud NUMA l'exécuter (directement ou indirectement en précisant que la tâche doit s'exécuter au même endroit qu'un fragment). Néanmoins, la faisabilité et le passage à l'échelle de cette approche restent à évaluer. De plus, l'affinité des tâches n'est pas le seul aspect à considérer pour supporter pleinement ce matériel : les données temporaires doivent être efficacement distribuées sur les nœuds NUMA, idéalement en considérant les métatâches qui opèrent sur ces données temporaires ainsi que la distribution de données des buffers sur lesquelles les métatâches travaillent (optimisation réalisable à l'échelle de l'assemblage uniquement). L'élaboration d'une méthode permettant d'exploiter pleinement des architectures NUMA est un problème ouvert qui peut impacter Comet ainsi que sa mise en œuvre.

Support étendu des architectures distribuées Comet supporte les architectures distribuées grâce aux communicateurs MPI hérités directement de L^{2}C et permet d'exploiter les architectures à mémoire partagée par le biais des sections dataflow. Cependant, l'utilisation d'une section entrecoupée de communications collectives MPI engendre des synchronisations inutiles et donc une sous-utilisation des ressources de calcul. Une solution pour résoudre ce problème consiste à adapter les sections dataflow afin qu'elles intègrent des métatâches de communications MPI. Certains supports exécutifs tels que StarPU ou OmpSs sont ensuite capables de calculer un ordonnancement efficace en prenant en compte ces tâches de communications de manière à recouvrir les communications MPI par l'exécution d'autres tâches soumises. Une alternative serait que la section soit distribuée sur plusieurs

1. La version 5.0 d'OpenMP ne supporte pas les spécificités des architectures NUMA

ressources permettant ainsi un ordonnancement global sur l'ensemble d'une plateforme de calcul et une répartition du travail entre les nœuds.

Sélection automatique de paramètres Dans un assemblage COMET, certaines propriétés telles que le type des buffers de données, le type de partitionnement, les paramètres de partitionnement, l'expression des dépendances ou la présence et la réutilisation de données temporaires doivent être choisies explicitement et manuellement. Déterminer les valeurs optimales peut être difficile et fastidieux, car l'optimalité des propriétés dépend généralement de l'ensemble de l'assemblage ainsi que de l'architecture matérielle sous-jacente. Ces choix pourraient donc être automatisés.

Conception d'un modèle de plus haut niveau Afin d'offrir un niveau d'abstraction plus élevé propice à la conception des applications HPC, un modèle de plus haut niveau est à construire au-dessus de COMET. Un tel modèle pourrait être hiérarchique, générique et reconfigurable. De plus, ce modèle de plus haut niveau devrait idéalement être capable d'exprimer des traitements cycliques ou facultatifs dans les assemblages de COMET tout en palliant les problèmes de synchronisation issus des sections dataflow. Les modèles à base de flux de travaux offrent des mécanismes de composition qui semblent adaptés à la résolution de ce problème.

Chapitre A

Bibliographie

[1] S. Agostinelli et al. “Geant4 – a simulation toolkit”. In : *Nuclear Instruments and Methods in Physics Research Section A : Accelerators, Spectrometers, Detectors and Associated Equipment* (2003).

[2] Marco Aldinucci et al. “FastFlow : High-Level and Efficient Streaming on Multi-core”. In : *Programming Multicore and Many-core Computing Systems*. Wiley, 2014. isbn : 978-0-470-93690-0.

[3] Marco Aldinucci et al. “STKM on SCA : A Unified Framework with Components, Workflows and Algorithmic Skeletons”. In : *Proceedings of the International Conference on Parallel and Distributed Computing*. Euro-Par’2009. Springer, 2009.

[4] B. A. Allan et al. “A Component Architecture for High-Performance Scientific Computing”. In : *International Journal of High Performance Computing Applications* (2006).

[5] Jeremie Allard et al. “FlowVR : a Middleware for Large Scale Virtual Reality Applications”. In : *Proceedings of the International Conference on Parallel and Distributed Computing*. Euro-Par’2004. Springer, 2004.

[6] Stephen F. Altschul et al. “Basic Local Alignment Search Tool”. In : *Journal of Molecular Biology* (1990).

[7] Gene M. Amdahl. “Validity of the Single-Processor Approach to Achieving Large Scale Computing Capabilities”. In : *Proceedings of the Joint Computer Conference*. AFIPS’1967 (Spring). ACM, 1967.

[8] Gabriel Antoniu et al. “Enabling Transparent Data Sharing in Component Models”. In : *Proceedings of the International Symposium on Cluster Computing and the Grid*. CCGrid’2006. IEEE, 2006.

[9] Cédric Augonnet et al. “StarPU : A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures”. In : *Concurrency and Computation : Practice and Experience, Special Issue : Euro-Par 2009* (2011).

[10] Olivier Aumage et al. “Combining Both a Component Model and a Task-based Model for HPC Applications : A Feasibility Study on Gysela”. In : *Proceedings of the IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. CCGrid'17. IEEE, 2017.

[11] David H Bailey et al. “The NAS Parallel Benchmarks”. In : *The International Journal of High Performance Computing Applications* (1991).

[12] Ananda Basu, Marius Bozga et Joseph Sifakis. “Modeling Heterogeneous Real-time Components in BIP”. In : *Proceedings of the IEEE International Conference on Software Engineering and Formal Methods*. SEFM'2006. IEEE, 2006.

[13] Françoise Baude et al. “GCM : A Grid Extension to Fractal for Autonomous Distributed Components”. In : *Annals of Telecommunications* (2009).

[14] Françoise Baude et al. “Interactive and Descriptor-based Deployment of Object-Oriented Grid Applications”. In : *Proceedings of the International Symposium on High Performance Distributed Computing*. HPDC'2002. IEEE, 2002.

[15] Michael Bauer et al. “Legion : Expressing Locality and Independence with Logical Regions”. In : *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC'2012. IEEE, 2012.

[16] Siegfried Benkner et al. “The PEPPHER Approach to Programmability and Performance Portability for Heterogeneous many-core Architectures”. In : *Proceedings of the International Conference on Parallel Computing*. ParCo'2011. IOS press, 2011.

[17] Jean Bézivin. “In Search of a Basic Principle for Model Driven Engineering”. In : *Novatica Journal (Special Issue)* (2004).

[18] Julien Bigot. “Du support générique d'opérateurs de composition dans les modèles de composants logiciels, application au calcul scientifique.” Thèse de doct. INSA de Rennes (France), 2010. URL : https://tel.archives-ouvertes.fr/tel-00626698v2.

[19] Julien Bigot et Christian Pérez. “High Performance Composition Operators in Component Models”. In : *High Performance Computing : From Grids and Clouds to Exascale*. HPC'2011. IOS Press, 2011.

[20] Julien Bigot et al. “A low level component model easing performance portability of HPC applications”. In : *Computing* (2014).

[21] Julien Bigot et al. “Building and Auto-Tuning Computing Kernels : Experimenting with BOAST and StarPU in the GYSELA code”. In : *ESAIM : Proceedings and Surveys*. 2017.

[22] Julien Bigot et al. *COMET : An High-Performance Model for Fine-Grain Composition*. Inria research report. LIP laboratory, ENS Lyon, France, 2017. URL : https://hal.inria.fr/hal-01566288.

[23] Julien Bigot et al. “Scaling GYSELA code beyond 32K-cores on Blue Gene/Q”. In : *ESAIM : Proceedings* (2013).

[24] Robert D. Blumofe et al. “Cilk : An Efficient Multithreaded Runtime System”. In : *Journal of parallel and distributed computing* (1996).

[25] Barry Boehm. “A View of 20th and 21st Century Software Engineering”. In : *Proceedings of the 28th International Conference on Software Engineering*. ICSE'2006. ACM, 2006.

[26] George Bosilca et al. “Scalable Dense Linear Algebra on Heterogeneous Hardware”. In : (2013).

[27] Hinde Lilia Bouziane, Christian Pérez et Thierry Priol. “A Software Component Model with Spatial and Temporal Compositions for Grid Infrastructures”. In : *Proceedings of the International Conference on Parallel and Distributed Computing*. Euro-Par'2008. Springer, 2008.

[28] Hinde Lilia Bouziane, Christian Pérez et Thierry Priol. “Extending Software Component Models with the Master-worker Paradigm”. In : *Parallel Computing* (2010).

[29] François Broquedis, Thierry Gautier et Vincent Danjean. “LIBKOMP, an Efficient OpenMP Runtime System for Both Fork-join and Data Flow Paradigms”. In : *Proceedings of the International Conference on OpenMP*. IWOMP'2012. Springer, 2012.

[30] Eric Bruneton et al. “The FRACTAL Component Model and Its Support in Java : Experiences with Auto-adaptive and Reconfigurable Systems”. In : *Software Practice Experience* (2006).

[31] Javier Bueno et al. “Productive Programming of GPU Clusters with OmpSs”. In : *Proceedings of the International Symposium on Parallel and Distributed Processing*. IPDPS'2012. IEEE, 2012.

[32] Paul Caspi et Marc Pouzet. “Synchronous Kahn Networks”. In : *SIGPLAN Notices*. ICFP'1996. ACM, 1996.

[33] Bradford L Chamberlain, David Callahan et Hans P Zima. “Parallel Programmability and the Chapel Language”. In : *The International Journal of High Performance Computing Applications* (2007).

[34] Philippe Charles et al. “X10 : An Object-oriented Approach to Non-uniform Cluster Computing”. In : *SIGPLAN Notices*. OOPSLA'2005. ACM, 2005.

[35] Philipp Ciechanowicz, Michael Poldner et Herbert Kuchen. *The Münster Skeleton Library Muesli : A comprehensive overview*. Rapport de recherche. Working Papers, ERCIS-European Research Center for Information Systems, 2009.

[36] Melvin E. Conway. “A Multiprocessor System Design”. In : *Proceedings of the Fall Joint Computer Conference*. AFIPS'1963 (Fall). ACM, 1963.

[37] Hélène Coullon, Christian Pérez et Jérôme Richard. *Feasibility Study of a Runtime Component-based Model Integrating Task Graph Concept on a 1D Advection Case Study*. Deliverable D3.2.1 of the ELCI PIA project. LIP laboratory, ENS Lyon, France, 2016. URL : https://hal.inria.fr/hal-01348204.

[38] Ole-Johan Dahl. *SIMULA 67 Common Base Language*. Sous la dir. de Norwegian Computing Center. 1968. ISBN : B0007JZ9J6.

[39] Ole-Johan Dahl, Edsger W. Dijkstra et C. A. R. Hoare, éds. *Structured Programming*. Academic Press, 1972. ISBN : 0122005503.

[40] Zachary DeVito et al. "Liszt : A Domain Specific Language for Building Portable Mesh-based PDE Solvers". In : *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis*. SC'2011. ACM, 2011.

[41] Jack J. Dongarra et al. *LINPACK Users' Guide*. SIAM, 1979. ISBN : 978-0-89871-172-1.

[42] Jack Dongarra, Michael A Heroux et Piotr Luszczek. "High-performance conjugate-gradient benchmark : A new metric for ranking high-performance computing systems". In : *The International Journal of High Performance Computing Applications* (2016).

[43] Matthieu Dreher et Tom Peterka. "Bredala : Semantic Data Redistribution for In Situ Applications". In : *Proceedings of the International Conference on Cluster Computing*. Cluster'2016. IEEE, 2016.

[44] Matthieu Dreher et Tom Peterka. *Decaf : Decoupled Dataflows for In Situ High-Performance Workflows*. Rapp. tech. Argonne National Lab.(ANL), Argonne, IL (United States), 2017. URL : http://www.ipd.anl.gov/anlpubs/2017/07/136823.pdf.

[45] Desmond F. D'Souza et Alan Cameron Wills. *Objects, Components, and Frameworks with UML : The Catalysis Approach*. Addison-Wesley Longman Publishing, 1999. ISBN : 0-201-31012-0.

[46] Alejandro Duran, Julita Corbalán et Eduard Ayguadé. "An Adaptive Cut-off for Task Parallelism". In : *Proceedings of the Conference on Supercomputing*. SC'2008. ACM, 2008.

[47] August Ernstsson, Lu Li et Christoph Kessler. "SkePU 2 : Flexible and Type-Safe Skeleton Programming for Heterogeneous Parallel Systems". In : *International Journal of Parallel Programming* (2017).

[48] Jason Evans. "A scalable concurrent malloc implementation for FreeBSD". In : *Proceedings of the BSDCAN conference*. BSDCAN'2006. 2006.

[49] Jean-Marie Favre. "Towards a Basic Theory to Model Model Driven Engineering". In : *Proceeding of the Workshop in Software Model Engineering*. WiSME'2004. Springer, 2004.

[50] Matteo Frigo, Charles E. Leiserson et Keith H. Randall. “The Implementation of the Cilk-5 Multithreaded Language”. In : *SIGPLAN Notices* (1998).

[51] François Galilée et al. “Athapascan-1 : On-Line Building Data Flow Graph in a Parallel Language”. In : *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. PACT’1998. IEEE, 1998.

[52] Thierry Gautier et al. “XKaapi : A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures”. In : *International Symposium on Parallel and Distributed Processing*. IPDPS’2013. IEEE, 2013.

[53] Tarek El-Ghazawi et Lauren Smith. “UPC : Unified Parallel C”. In : *Proceedings of the Conference on Supercomputing*. SC’2006. ACM, 2006.

[54] W Wayt Gibbs. “Software’s Chronic Crisis”. In : *Scientific American* (1994).

[55] *GNU OpenMP Runtime (GOMP) Implementation*. GNU Project. url : https://gcc.gnu.org/projects/gomp/.

[56] Horacio González-Vélez et Mario Leyton. “A Survey of Algorithmic Skeleton Frameworks : High-level Structured Parallel Programming Enablers”. In : *Software : Practice and Experience* (2010).

[57] Virginie Grandgirard. *High-Q club : Highest Scaling Codes on JUQUEEN – GYSELA : GYrokinetic SEmi-LAgrangian code for plasma turbulence simulations*. Mar. 2015.

[58] Virginie Grandgirard et al. “A 5D gyrokinetic full-f global semi-Lagrangian code for flux-driven ion turbulence simulations”. In : *Computer Physics Communications* (2016).

[59] John L. Gustafson. “Reevaluating Amdahl’s Law”. In : *Communications of the ACM* (1988).

[60] George T. Heineman et William T. Councill, éds. *Component-based Software Engineering : Putting the Pieces Together*. Addison-Wesley Longman Publishing, 2001. isbn : 0-201-70485-4.

[61] Maurice Herlihy et J. Eliot B. Moss. “Transactional Memory : Architectural Support for Lock-free Data Structures”. In : *Proceedings of the 20th Annual International Symposium on Computer Architecture*. ISCA’93. ACM, 1993.

[62] Berk Hess et al. “GROMACS 4 : Algorithms for Highly Efficient, Load-Balanced, and Scalable Molecular Simulation”. In : *Journal of Chemical Theory and Computation* (2008).

[63] Richard M. Hodur. “The Naval Research Laboratory’s Coupled Ocean/Atmosphere Mesoscale Prediction System (COAMPS)”. In : *Monthly Weather Review* (1997).

[64] Andra-Ecaterina Hugo. “Composability of Parallel Codes on Heterogeneous Architectures”. Thèse de doct. Université de Bordeaux (France), 2014. url : https://tel.archives-ouvertes.fr/tel-01162975.

[65] *Intel OpenMP Runtime (IOMP) Implementation*. Intel. URL : http://clang-omp.github.io.

[66] Bruce JACOB, Spencer NG et David WANG. *Memory Systems : Cache, DRAM, Disk*. Morgan Kaufmann, 2007. ISBN : 9780080553849.

[67] Jean-Marc JÉZÉQUEL, Benoît COMBEMALE et Didier VOJTISEK. *Ingénierie Dirigée par les Modèles : des concepts à la pratique*. Ellipses, 2012. ISBN : 013359162X.

[68] Gilles KAHN. "The Semantics of a Simple Language for Parallel Programming". In : *Information Processing*. IFIP'1974. North Holland, Amsterdam, 1974.

[69] Laxmikant V. KALE et Sanjeev KRISHNAN. "CHARM++ : A Portable Concurrent Object Oriented System Based on C++". In : *SIGPLAN Notice* (1993).

[70] Ken KENNEDY, Charles KOELBEL et Hans ZIMA. "The Rise and Fall of High Performance Fortran : An Historical Object Lesson". In : *Proceedings of the Conference on History of Programming Languages*. HOPL III. ACM, 2007.

[71] Volodymyr KINDRATENKO et Pedro TRANCOSO. "Trends in High-Performance Computing". In : *Computing in Science & Engineering* (2011).

[72] Charles H. KOELBEL et Mary E. ZOSEL. *The High Performance FORTRAN Handbook*. MIT Press, 1993. ISBN : 0262111853.

[73] Jakub KURZAK et Jack DONGARRA. *Fully Dynamic Scheduler for Numerical Scheduling on Multicore Processors*. Rapp. tech. Report LAWN (LAPACK Working Note) 220, UT-CS-09-643. University of Tennessee, 2009.

[74] Christoph LAMETER. "NUMA (Non-Uniform Memory Access) : An Overview". In : *ACM Queue* (2013).

[75] Vincent LANORE, Christian PEREZ et Jérôme RICHARD. *Evaluating Component Assembly Specialization for 3D FFT*. PRACE-2IP White Paper. LIP laboratory, ENS Lyon, France, 2014. URL : http://www.prace-ri.eu/IMG/pdf/WP182.pdf.

[76] Vincent LANORE, Christian PEREZ et Jérôme RICHARD. "Towards Application Variability Handling with Component Models : 3D-FFT Use Case Study". In : *Euro-Par 2015 : Parallel Processing Workshops*. UCHPC'2015. Springer, 2015.

[77] Kung-Kiu LAU et al. "A Component Model That is Both Control-driven and Data-driven". In : *Proceedings of the International Symposium on Component Based Software Engineering*. CBSE'2011. ACM, 2011.

[78] Jinpil LEE et Mitsuhisa SATO. "Implementation and Performance Evaluation of XcalableMP : A Parallel Programming Language for Distributed Memory Systems". In : *Proceedings of the International Conference on Parallel Processing Workshops*. ICPPW'10. IEEE Computer Society, 2010.

[79] Jean-Denis Lesage et Bruno Raffin. “A Hierarchical Component Model for Large Parallel Interactive Applications”. In : *Journal of Supercomputing* (2012).

[80] Barbara H. Liskov et Jeannette M. Wing. “A Behavioral Notion of Subtyping”. In : *ACM Transactions on Programming Languages and Systems* (1994).

[81] Piotr R Luszczek et al. “The HPC Challenge (HPCC) Benchmark Suite”. In : *Proceedings of the Conference on Supercomputing*. SC’2006. ACM, 2006.

[82] Henry Markram et al. “Reconstruction and Simulation of Neocortical Microcircuitry”. In : *Cell* (2015).

[83] M. D. McIlroy. “Mass-produced Software Components”. In : *Proc. NATO Conf. on Software Engineering, Garmisch, Germany* (1968).

[84] Sally A. McKee et Robert W. Wisniewski. “Memory Wall”. In : *Encyclopedia of Parallel Computing*. Sous la dir. de David Padua. Springer, 2011, p. 1110–1116. isbn : 978-0-387-09766-4. doi : 10.1007/978-0-387-09766-4_234.

[85] Johan Montagnat et al. “A Data-driven Workflow Language for Grids Based on Array Programming Principles”. In : *Proceedings of the Workshop on Workflows in Support of Large-Scale Science*. WORKS’2009. ACM, 2009.

[86] Gordon E. Moore. “Cramming More Components Onto Integrated Circuits”. In : *Electronics* (1965).

[87] Gordon E. Moore. “Progress in Digital Integrated Electronics”. In : *Electron Devices Meeting*. IEDM’1975. IEEE, 1975.

[88] *MPI : A Message-Passing Interface Standard, Version 2.2*. Specification. Message Passing Interface Forum, sept. 2009. url : http://mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf.

[89] *MPI : A Message-Passing Interface Standard, Version 3.1*. Specification. Message Passing Interface Forum, juin 2015. url : http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.

[90] Peter Naur et Brian Randell, éds. *Software Engineering : Report of a Conference Sponsored by the NATO Science Committee (7-11 Oct. 1968), Garmisch, Germany*. NATO, Science Committee, 1969.

[91] Robert W. Numrich et John Reid. “Co-array Fortran for Parallel Programming”. In : *SIGPLAN Fortran Forum* (1998).

[92] Stephen Olivier et al. “OpenMP Task Scheduling Strategies for Multicore NUMA Systems”. In : *The International Journal of High Performance Computing Applications* (2012).

[93] Rob van Ommering et al. “The Koala Component Model for Consumer Electronics Software”. In : *Computer* (2000).

[94] OpenMP Architecture Review Board. *OpenMP Application Programming Interface Version 4.5*. Nov. 2015. URL : http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf.

[95] David A. Patterson. "Latency Lags Bandwith". In : *Communications of the ACM* (2004).

[96] Christian Pérez, Thierry Priol et André Ribes. "A Parallel CORBA Component Model for Numerical Code Coupling". In : *International Journal of High Performance Computing Applications* (2003).

[97] William H Press. *The Art of Scientific Computing*. 3th. Cambridge University Press, 2007. ISBN : 0521880688.

[98] *Proceedings of the International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*. WOLFHPC. ACM.

[99] James Reinders. *Intel Threading Building Blocks : Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, 2007. ISBN : 0596514808.

[100] Jérôme Richard. *Vers un modèle de composants supportant l'ordonnancement de tâches pour le calcul de haute performance*. 2015. URL : https://hal.inria.fr/hal-01192661.

[101] Martin C. Rinard. "The Design, Implementation and Evaluation of Jade : A Portable, Implicitly Parallel Programming Language". Thèse de doct. Stanford University (USA), 1994. URL : http://i.stanford.edu/pub/cstr/reports/csl/tr/94/638/CSL-TR-94-638.pdf.

[102] James Rumbaugh et al. *Object-Oriented Modeling and Design*. Prentice-hall Englewood Cliffs, 1991. ISBN : 0136298419.

[103] Jason Sanders et Edward Kandrot. *CUDA by Example : An Introduction to General-Purpose GPU Programming, Portable Documents*. Addison-Wesley Professional, 2010. ISBN : 0131387685.

[104] Mary Shaw, Robert DeLine et Gregory Zelesnik. "Abstractions and Implementations for Architectural Connections". In : *Proceedings of the International Conference on Configurable Distributed Systems*. ICCDS. IEEE, 1996.

[105] Elliott Slaughter et al. "Regent : a high-productivity programming language for HPC with logical regions". In : *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC'2015. ACM, 2015.

[106] *Specification of the CORBA Component Model, Version 4*. Specification. Object Management Group, avr. 2006. URL : http://www.omg.org/spec/CCM/4.0/PDF.

[107] J. Ramanujam Sriram Krishnamoorthy et P. Sadayappan, éds. *Journal of Parallel and Distributed Computing* 74.12 (2014) : *Domain-Specific Languages and High-Level Frameworks for High-Performance Computing*.

[108] John E Stone, David Gohara et Guochun Shi. “OpenCL : A Parallel Programming Standard for Heterogeneous Computing Systems”. In : *Computing in Science & Engineering* (2010).

[109] Gilbert Strang. “On the Construction and Comparison of Difference Schemes”. In : *SIAM Journal on Numerical Analysis* (1968).

[110] *Superstructure Specification of the Unified Modeling Language, Version 2.4.1*. Specification. Object Management Group, août 2009. URL : http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF.

[111] Clemens Szyperski. *Component Software : Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN : 0201745720.

[112] Haruto Tanno et Hideya Iwasaki. “Parallel Skeletons for Variable-Length Lists in SkeTo Skeleton Library”. In : *Proceedings of the International Conference on Parallel and Distributed Computing*. Euro-Par’2009. Springer, 2009.

[113] Romain Teyssier. “Cosmological Hydrodynamics with Adaptive Mesh Refinement : a new high resolution code called RAMSES”. In : *Astronomy & Astrophysics* (2002).

[114] *TOP500 Supercomputer Site*. URL : http://www.top500.org.

[115] Philippe Virouleau et al. “Description, Implementation and Evaluation of an Affinity Clause for Task Directives”. In : *Proceedings of the International Conference on OpenMP : Memory, Devices, and Tasks, IWOMP 2012*. IWOMP’2016. Springer, 2016.

[116] Anatoly Vlasov. “On the Kinetic Theory of an Assembly of Particles with Collective Interaction”. In : *Journal of Physics USSR* (1945).

[117] Nanbor Wang, Douglas C Schmidt et David Levine. “Optimizing the CORBA component model for high-performance and real-time applications”. In : *Work-in-Progress’ session at the Middleware 2000 Conference*. Middleware’2000. ACM, 2000.

[118] Sandra Wienke et al. “OpenACC : First Experiences with Real-world Applications”. In : *Proceedings of the International Conference on Parallel and Distributed Computing*. Euro-Par’2012. Springer, 2012.

[119] Niklaus Wirth. “Modula : A Language for Modular Multiprogramming”. In : *Software : Practice and Experience* (1977).

[120] Wei Wu et al. “Hierarchical DAG Scheduling for Hybrid Distributed Systems”. In : *Proceedings of the International Parallel and Distributed Processing Symposium*. IPDPS’2015. IEEE, 2015.

[121] Edward Yourdon et Larry L. Constantine. *Structured Design : Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, 1979. ISBN : 0138544719.

Chapitre B

Annexes

B.1 Modèle source de la mise en œuvre HALLEY

```
        // COMET version 1.2 abstract syntax (logical information model)

// Modeling notes on identifiers and references:
//  Information about identifiers/reference constraints are not
//   formally defined in this document (e.g. whether identifiers are
//   unique in a predefined namespace or whether references refer to
//   elements of the same predefined namespace).
//  When no information are provided, references can refer to any
//   element of the model.
//  All references hold by connections in the model can only refer to
//   the recursive children of the parent element that contains the
//   connection itself
//  When the logical model is derived to a more physical one
//   (implementation), one of the choices has to be made:
//      - Id are local and element references are derived either to a
//         tuple (namespaceId, elementId) or a unique elementId attribute;
//      - All id must be global and not local (including components,
//         metatask and section instances as well as ports)
//         breaking locality.

package main;


    // External types (out-of-scope definitions, i.e. model interfaces)

// Reference to elements defined outside the model:
// low-level language types (e.g. C++ type) or to specific IDL types

// Basically a special C++ abstract class or an IDL type
datatype DataBuffPortTypeDef : String;

// Basically a special C++ abstract class or an IDL type
datatype DataParPortTypeDef : String;

// Basically a C++ abstract class or an IDL type
datatype ServicePortTypeDef : String;

// Basically a C++ class that implement abstract class
datatype ComponentImplDef : String;

// Basically a primitive C++ type or predefined structured C++ type
// or a primitive IDL type
datatype AttributeTypeDef : String;
```

```
44
45  // Basically a template parameter type for metatask,
46  // also seen as an attribute at runtime, thus map to a C++ type
47  datatype ParameterTypeDef : String;
48
49  // A set of mapping language are predefined within the model
50  datatype MappingLanguage : String;
51
52  // A raw string that match the mapping language
53  datatype MappingExpression : String;
54
55  // A function definition that define the parameters of the type
56  // according to metatask parameter values
57  datatype ParameterSettingFunction : String;
58
59
60      // External type relation
61
62  class PartitioningTypeRelation
63  {
64      ref DataParPortTypeDef partType;
65      ref DataBuffPortTypeDef dataType;
66  }
67
68
69      // Types
70
71  class AttributeDef
72  {
73      id attr String name;
74      ref AttributeTypeDef llType;
75  }
76
77  class ParameterDef
78  {
79      id attr String name;
80      ref ParameterTypeDef type;
81  }
82
83  class MpiPortDef
84  {
85      id attr String name;
86  }
87
88  class UsePortDef
89  {
90      id attr String name;
91      ref ServicePortTypeDef itfType;
92      attr Boolean taskSafe;
93  }
94
95  class ProvidePortDef
96  {
97      id attr String name;
98      ref ServicePortTypeDef itfType;
99      attr Boolean taskSafe;
100 }
101
102 enum ComponentDataPortKind
103 {
104     IN; OUT; INOUT;
105 }
106
107 // A port that own/share data
108 class ComponentDataPortDef
109 {
110     id attr String name;
```

```
111     attr ComponentDataPortKind kind;
112     ref DataBuffPortTypeDef dataType;
113 }
114
115 class ComponentDef
116 {
117     id attr String name;
118     val AttributeDef[] attributes;
119     val ProvidePortDef[] providePorts;
120     val UsePortDef[] usePorts;
121     val MpiPortDef[] mpiPorts;
122     val ComponentDataPortDef[] dataPorts;
123
124     // Implementation details (?)
125     ref ComponentImplDef llType;
126 }
127
128 enum MetataskDataPortKind
129 {
130     IN; OUT; INOUT;
131 }
132
133 class MetataskDataPortDef
134 {
135     id attr String name;
136     attr MetataskDataPortKind kind;
137     ref DataParPortTypeDef dataParType;
138
139     // Implementation details (?)
140     ref ParameterSettingFunction dataBuffSetter;
141     ref ParameterSettingFunction dataParSetter;
142 }
143
144 // This entity holds an implicit task-safe use port called "compute"
145 // with a predefined type.
146 // The predefined type depend on both the mapping language and the ports.
147 // Simple "alignment" language can be used to generate bags of
148 // independant tasks at runtime.
149 // More complex "alignment" language can be used to generate partial
150 // DAG of task with complex deps.
151 class MetataskDef
152 {
153     id attr String name;
154     val ParameterDef[] parameters;
155     val MetataskDataPortDef[] dataPorts;
156     ref MappingLanguage mappingLanguage;
157     opt val MappingExpression mappingExpression;
158 }
159
160 // Unit that defines how to deal with a virtual or physical data buffer
161 class DataBuffDef
162 {
163     id attr String name;
164     ref DataBuffPortTypeDef type;
165     val AttributeDef[] attributes;
166
167     // Implementation details (?)
168     ref ComponentImplDef llType;
169 }
170
171 // Unit that defines how to split a data in fragments
172 class DataParDef
173 {
174     id attr String name;
175     ref DataParPortTypeDef type;
176     val AttributeDef[] attributes;
177
```

```
178     // Implementation details (?)
179     ref ComponentImplDef llType;
180 }
181
182
183     // Instances, connections and assembly
184
185 class AttributeSetting
186 {
187     id attr AttributeDef attribute;
188     attr String value;
189 }
190
191 class ParameterSetting
192 {
193     id attr ParameterDef attribute;
194     attr String value;
195 }
196
197 class ComponentInstance
198 {
199     id attr String name;
200     ref ComponentDef type;
201     val AttributeSetting[] attributeSettings;
202 }
203
204 class MetataskPartitionSetting
205 {
206     ref MetataskDataPortDef port;
207
208     // Refers to the expert component implementation, automatically
209     // chosen when one compliant component is available
210     ref DataParDef type;
211 }
212
213 class MetataskInstance
214 {
215     id attr String name;
216     ref MetataskDef type;
217
218     // Meta-information such as the buffer size
219     // (related both to the type and the interface)
220     val ParameterSetting[] parameterSettings;
221
222     // Implementation details (?)
223     opt val MetataskPartitionSetting partitionSettings;
224 }
225
226 class DataBuffInstance
227 {
228     id attr String name;
229
230     // Refers to the expert component implementation, automatically
231     // chosen when one compliant component is available
232     opt attr DataBuffDef type;
233 }
234
235 // In-section data end-point
236 interface DataEndPoint
237 {
238
239 }
240
241 enum AccessKind
242 {
243     IN; OUT;
244 }
```

```
245
246 class MetataskDataEndPoint extends DataEndPoint
247 {
248     ref MetataskInstance metatask;
249     ref MetataskDataPortDef port;
250     // Only use for inout port that act as in+out ports
251     opt val AccessKind accessKind;
252 }
253
254 // Semantics: a port can be connected only once
255 class ComponentDataEndPoint extends DataEndPoint
256 {
257     ref ComponentInstance instance;
258     ref ComponentDataPortDef port;
259     // Only use for inout port that act as in+out ports
260     opt val AccessKind accessKind;
261 }
262
263 // Data connector
264 // Semantics: Define the flow of information between data ports
265 // Connectability delayed to the semantics:
266 // e.g. input port must be connected to output port, two components
267 // cannot be directly connected...
268 class DataConnection
269 {
270     ref DataEndPoint source;
271     ref DataEndPoint destination;
272 }
273
274 // Data buffer connector
275 // Semantics: define metainformation of the buffer
276 class DataBuffConnection
277 {
278     ref DataEndPoint dataPort;
279     ref DataBuffInstance dataBuffer;
280 }
281
282 // This entity holds an implicit "control" provide port as well an
283 // implicit "status" provide port.
284 // These ports are closely related to the runtime life-cycle of the section.
285 class DataflowSectionInstance
286 {
287     id attr String name;
288     val MetataskInstance[] metatasks;
289     val DataBuffInstance[] dataBuffers;
290     val DataConnection[] dataConnections;
291 }
292
293 interface UseEndPoint
294 {
295
296 }
297
298 class ComponentUseEndPoint extends UseEndPoint
299 {
300     ref ComponentInstance instance;
301     ref UsePortDef port;
302 }
303
304 class MetataskUseEndPoint extends UseEndPoint
305 {
306     ref MetataskInstance instance;
307     //ref UsePortDef port; // Implicit port
308 }
309
310 class ProvideEndPoint
311 {
```

```
312     ref ComponentInstance instance;
313     ref ProvidePortDef port;
314 }
315
316 class UseProvideConnection
317 {
318     val UseEndPoint user;
319     val ProvideEndPoint provider;
320 }
321
322 class MpiProcess
323 {
324     ref ProvideEndPoint entryPoint;
325     val ComponentInstance[] components;
326     val DataflowSectionInstance[] sections;
327     val UseProvideConnection[] useProvideConnections;
328 }
329
330 class MpiEndPoint
331 {
332     ref ComponentInstance instance;
333     ref MpiPortDef port;
334 }
335
336 class MpiConnection
337 {
338     ref MpiEndPoint[] endPoint;
339 }
340
341 class Assembly
342 {
343     val MpiProcess[] processes;
344     val MpiConnection[] mpiConnections;
345 }
346
347 class ModelInstance
348 {
349     // User-defined components
350     val ComponentDef[] components;
351     val MetataskDef[] metatasks;
352
353     // Expert-defined components
354     val DataBuffDef[] data;
355     val DataParDef[] dataPar;
356
357     val PartitioningTypeRelation[] dataParRelation;
358
359     val Assembly assembly;
360 }
```

FIGURE B.1 – Définition du modèle source de HALLEY. Le langage utilisé pour décrire le modèle est Emfatic de ECORE (fondation Éclipse).

B.2 Définition Emfatic des assemblages L²C

```
1 @namespace(uri="http://www.inria.fr/avalon/lad/1.0.0", prefix="lad")
2 package lad;
3
4 import "http://www.eclipse.org/emf/2003/XMLType";
5
```

```
@ExtendedMetaData(name="",
                  kind="mixed")
class DocumentRoot
{
    @ExtendedMetaData(kind="elementWildcard",
                      name=":mixed")
    !unique attr ecore.EFeatureMapEntry[*] mixed;

    @ExtendedMetaData(kind="attribute",
                      name="xmlns:prefix")
    transient !resolve val ecore.EStringToStringMapEntry[*] xMLNSPrefixMap;

    @ExtendedMetaData(kind="attribute",
                      name="xsi:schemaLocation")
    transient !resolve val ecore.EStringToStringMapEntry[*] xSISchemaLocation;

    @ExtendedMetaData(kind="element",
                      name="lad",
                      namespace="##targetNamespace")
    volatile transient derived !resolve val Lad[1] lad;
}

@ExtendedMetaData(name="lad",
                  kind="elementOnly")
class Lad
{
    @ExtendedMetaData(kind="element",
                      name="mpi",
                      namespace="##targetNamespace")
    !resolve val MpiGroup[*] mpi;

    @ExtendedMetaData(kind="element",
                      name="process",
                      namespace="##targetNamespace")
    !resolve val Process[*] process;
}

@ExtendedMetaData(name="mpi_group",
                  kind="elementOnly")
class MpiGroup
{
    @ExtendedMetaData(kind="element",
                      name="process",
                      namespace="##targetNamespace")
    !resolve val Process[+] processes;

    @ExtendedMetaData(kind="element",
                      name="communicator",
                      namespace="##targetNamespace")
    !resolve val MpiCommunicator[*] communicator;

    @ExtendedMetaData(kind="attribute",
                      name="id")
    attr type.String ~id;
}

@ExtendedMetaData(name="mpi_communicator", kind="elementOnly")
class MpiCommunicator
{
    @ExtendedMetaData(kind="element",
                      name="peer",
                      namespace="##targetNamespace")
    !resolve val Endpoint[2..*] peer;
}

@ExtendedMetaData(name="process", kind="elementOnly")
class Process
```

```
{
    @ExtendedMetaData(kind="element",
                      name="start_property",
                      namespace="##targetNamespace")
    !resolve val Endpoint start_property;

    @ExtendedMetaData(kind="element",
                      name="instance",
                      namespace="##targetNamespace")
    !resolve val ComponentInstance[+] instance;

    @ExtendedMetaData(kind="attribute",
                      name="id")
    attr type.String ~id;
}

@ExtendedMetaData(name="instance", kind="elementOnly")
class ComponentInstance
{
    @ExtendedMetaData(kind="element",
                      name="property",
                      namespace="##targetNamespace")
    !resolve val PropertyValue[*] property;

    @ExtendedMetaData(kind="attribute",
                      name="from")
    attr type.String from;

    @ExtendedMetaData(kind="attribute",
                      name="id")
    id attr type.ID[1] ~id;

    @ExtendedMetaData(kind="attribute",
                      name="type")
    attr type.String[1] type;
}

@ExtendedMetaData(name="property_value", kind="elementOnly")
class PropertyValue
{
    @ExtendedMetaData(kind="group", name="group:0")
    !unique attr ecore.EFeatureMapEntry[*] group;

    @ExtendedMetaData(kind="element",
                      name="value",
                      namespace="##targetNamespace",
                      group="#group:0")
    volatile transient derived !resolve val DataValue[*] value;

    @ExtendedMetaData(kind="element",
                      name="cppref",
                      namespace="##targetNamespace",
                      group="#group:0")
    volatile transient derived !resolve val Endpoint[*] cppref;

    @ExtendedMetaData(kind="element",
                      name="corbaref",
                      namespace="##targetNamespace",
                      group="#group:0")
    volatile transient derived !resolve val Endpoint[*] corbaref;

    @ExtendedMetaData(kind="element",
                      name="systemref",
                      namespace="##targetNamespace",
                      group="#group:0")
    volatile transient derived !resolve val SystemRef[*] systemref;

```

```
140     @ExtendedMetaData(kind="attribute",
141                       name="id")
142     attr type.String[1] ~id;
143 }
144 
145 @ExtendedMetaData(name="data_value", kind="simple")
146 class DataValue
147 {
148     @ExtendedMetaData(name=":0", kind="simple")
149     attr type.String value;
150 
151     @ExtendedMetaData(kind="attribute",
152                       name="type")
153     unsettable attr Datatype[1] type;
154 }
155 
156 @ExtendedMetaData(name="data_type")
157 enum Datatype
158 {
159     int8 = 0;
160     uint8 = 1;
161     int16 = 2;
162     uint16 = 3;
163     int32 = 4;
164     uint32 = 5;
165     int64 = 6;
166     uint64 = 7;
167     double = 8;
168     complexDouble = 9;
169     string = 10;
170 }
171 
172 @ExtendedMetaData(name="data_type:Object",
173                   baseType="datatype")
174 datatype DataTypeObject : org.eclipse.emf.common.util.Enumerator;
175 
176 @ExtendedMetaData(name="endpoint", kind="empty")
177 class Endpoint
178 {
179     @ExtendedMetaData(kind="attribute",
180                       name="instance")
181     attr type.IDREF[1] instance;
182 
183     @ExtendedMetaData(kind="attribute",
184                       name="property")
185     attr type.String[1] property;
186 }
187 
188 @ExtendedMetaData(name="system_ref",
189                   kind="empty")
190 class SystemRef
191 {
192     @ExtendedMetaData(kind="attribute",
193                       name="name")
194     attr type.String[1] name;
195 }
```

FIGURE B.2 – Définition du modèle de description des assemblages L²C. Le langage utilisé pour décrire le modèle est Emfatic de ECORE (fondation Éclipse).

B.3 Définition de schéma XML pour la description de métatâches HALLEY

```xml
<?xml version="1.0" encoding="UTF-8"?>

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" version="1.2">
    <xsd:complexType name="Parameter">
        <xsd:attribute name="id" type="xsd:NCName" use="required"/>
        <xsd:attribute name="type" type="xsd:NCName" use="required"/>
    </xsd:complexType>

    <xsd:complexType name="Port">
        <xsd:attribute name="id" type="xsd:NCName" use="required"/>
        <xsd:attribute name="type" type="xsd:NCName" use="required"/>
        <xsd:attribute name="dataBufferSetter" type="xsd:NCName"/>
        <xsd:attribute name="dataPartitionSetter" type="xsd:NCName"/>
    </xsd:complexType>

    <xsd:complexType name="Alignment">
        <xsd:simpleContent>
            <xsd:extension base="xsd:string">
                <xsd:attribute name="language"
                        type="xsd:NCName" use="required"/>
            </xsd:extension>
        </xsd:simpleContent>
    </xsd:complexType>

    <xsd:complexType name="Setting">
        <xsd:simpleContent>
            <xsd:extension base="xsd:string">
                <xsd:attribute name="language"
                        type="xsd:NCName" use="required"/>
            </xsd:extension>
        </xsd:simpleContent>
    </xsd:complexType>

    <xsd:complexType name="Metatask">
        <xsd:all>
            <xsd:element minOccurs="0" maxOccurs="unbounded"
                    name="parameter" type="Parameter"/>
            <xsd:element minOccurs="0" maxOccurs="unbounded"
                    name="inPort" type="Port"/>
            <xsd:element minOccurs="0" maxOccurs="unbounded"
                    name="outPort" type="Port"/>
            <xsd:element minOccurs="0" maxOccurs="unbounded"
                    name="inoutPort" type="Port"/>
            <xsd:element minOccurs="1" maxOccurs="1"
                    name="alignment" type="Alignment"/>
            <xsd:element minOccurs="0" maxOccurs="1"
                    name="setting" type="Setting"/>
        </xsd:all>
        <xsd:attribute name="id" type="xsd:NCName"/>
    </xsd:complexType>

    <xsd:complexType name="Definitions">
        <xsd:choice minOccurs="0" maxOccurs="unbounded">
            <xsd:element name="metatask" type="Metatask"/>
        </xsd:choice>
    </xsd:complexType>

    <xsd:element name="cmtd" type="Definitions"/>
</xsd:schema>
```

FIGURE B.3 – Schéma XML (XSD) spécifiant le format des fichiers XML de HALLEY permettant de la décrire des métatâches.

B.4 Définition de schéma XML pour la description de greffons HALLEY

```xml
<?xml version="1.0" encoding="UTF-8"?>

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" version="1.2">
    <xsd:complexType name="DataType">
        <xsd:choice minOccurs="0" maxOccurs="unbounded">
            <xsd:element name="parameter" type="Parameter"/>
        </xsd:choice>
        <xsd:attribute name="id" type="xsd:NCName"/>
    </xsd:complexType>

    <xsd:complexType name="PartitionType">
        <xsd:choice minOccurs="0" maxOccurs="unbounded">
            <xsd:element name="parameter" type="Parameter"/>
        </xsd:choice>
        <xsd:attribute name="id" type="xsd:NCName"/>
        <xsd:attribute name="datatype" type="xsd:NCName"/>
    </xsd:complexType>

    <xsd:complexType name="DataExpertImpl">
        <xsd:attribute name="id" type="xsd:NCName"/>
        <xsd:attribute name="type" type="xsd:NCName"/>
    </xsd:complexType>

    <xsd:complexType name="PartExpertImpl">
        <xsd:attribute name="id" type="xsd:NCName"/>
        <xsd:attribute name="type" type="xsd:NCName"/>
    </xsd:complexType>

    <xsd:complexType name="RepartExpertImpl">
        <xsd:attribute name="id" type="xsd:NCName"/>
        <xsd:attribute name="srcType" type="xsd:NCName"/>
        <xsd:attribute name="dstType" type="xsd:NCName"/>
    </xsd:complexType>

    <xsd:complexType name="Definitions">
        <xsd:choice minOccurs="0" maxOccurs="unbounded">
            <xsd:element name="dataType" type="DataType"/>
            <xsd:element name="partitionType" type="PartitionType"/>
            <xsd:element name="dataImpl" type="DataExpertImpl"/>
            <xsd:element name="partitionImpl" type="PartExpertImpl"/>
            <xsd:element name="repartitionImpl" type="RepartExpertImpl"/>
        </xsd:choice>
    </xsd:complexType>

    <xsd:element name="cmtd" type="Definitions"/>
</xsd:schema>
```

FIGURE B.4 – Schéma XML (XSD) spécifiant le format des fichiers XML de HALLEY permettant de la décrire des greffons.

B.5 Définition de schéma XML pour la description d'assemblages HALLEY

```xml
<?xml version="1.1" encoding="UTF-8"?>

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" version="1.2">
    <xsd:complexType name="AttributeConfiguration">
        <xsd:simpleContent>
            <xsd:extension base="xsd:string">
                <xsd:attribute name="id" type="xsd:NCName" use="required"/>
                <xsd:attribute name="type" type="xsd:string" use="required"/>
            </xsd:extension>
        </xsd:simpleContent>
    </xsd:complexType>

    <xsd:complexType name="ParameterConfiguration">
        <xsd:simpleContent>
            <xsd:extension base="xsd:string">
                <xsd:attribute name="id" type="xsd:NCName" use="required"/>
            </xsd:extension>
        </xsd:simpleContent>
    </xsd:complexType>

    <xsd:complexType name="EndPoint">
        <xsd:attribute name="instance" type="xsd:NCName" use="required"/>
        <xsd:attribute name="port" type="xsd:NCName" use="required"/>
    </xsd:complexType>

    <xsd:complexType name="UseConfiguration">
        <xsd:choice minOccurs="0" maxOccurs="unbounded">
            <xsd:element name="endPoint" type="EndPoint"/>
        </xsd:choice>
        <xsd:attribute name="port" type="xsd:NCName" use="required"/>
    </xsd:complexType>

    <xsd:complexType name="ComponentInstance">
        <xsd:choice minOccurs="0" maxOccurs="unbounded">
            <xsd:element name="attribute" type="AttributeConfiguration"/>
            <xsd:element name="use" type="UseConfiguration"/>
        </xsd:choice>
        <xsd:attribute name="id" type="xsd:NCName" use="required"/>
        <xsd:attribute name="from" type="xsd:string"/>
        <xsd:attribute name="type" type="xsd:NCName" use="required"/>
    </xsd:complexType>

    <xsd:complexType name="MetataskInstance">
        <xsd:all>
            <xsd:element minOccurs="0" maxOccurs="unbounded"
                    name="parameter" type="ParameterConfiguration"/>
            <xsd:element minOccurs="1" maxOccurs="1"
                    name="compute" type="EndPoint"/>
        </xsd:all>
        <xsd:attribute name="id" type="xsd:NCName" use="required"/>
        <xsd:attribute name="type" type="xsd:NCName" use="required"/>
        <xsd:attribute name="from" type="xsd:string"/>
    </xsd:complexType>

    <xsd:complexType name="DataBufferInstance">
        <xsd:choice minOccurs="0" maxOccurs="unbounded">
            <xsd:element name="parameter" type="ParameterConfiguration"/>
        </xsd:choice>
        <xsd:attribute name="id" type="xsd:NCName" use="required"/>
        <!-- interface data type -->
        <xsd:attribute name="type" type="xsd:NCName"/>
```

```
62          <!-- component implementation file path or name -->
63          <xsd:attribute name="from" type="xsd:string"/>
64      </xsd:complexType>
65
66      <xsd:complexType name="DataBufferConnection">
67          <xsd:attribute name="instance" type="xsd:NCName" use="required"/>
68          <xsd:attribute name="port" type="xsd:NCName" use="required"/>
69          <xsd:attribute name="data" type="xsd:NCName" use="required"/>
70      </xsd:complexType>
71
72      <xsd:complexType name="DataConnection">
73          <xsd:all>
74              <xsd:element name="from" type="EndPoint"/>
75              <xsd:element name="to" type="EndPoint"/>
76          </xsd:all>
77      </xsd:complexType>
78
79      <xsd:complexType name="DataflowSectionInstance">
80          <xsd:choice minOccurs="0" maxOccurs="unbounded">
81              <xsd:element name="metataskInstance"
82                      type="MetataskInstance"/>
83              <xsd:element name="dataBufferInstance"
84                      type="DataBufferInstance"/>
85              <xsd:element name="dataBufferConnection"
86                      type="DataBufferConnection"/>
87              <xsd:element name="dataConnection"
88                      type="DataConnection"/>
89          </xsd:choice>
90          <xsd:attribute name="id" type="xsd:NCName" use="required"/>
91          <xsd:attribute name="from" type="xsd:string"/>
92      </xsd:complexType>
93
94      <xsd:complexType name="MpiProcess">
95          <xsd:all>
96              <xsd:element minOccurs="1" maxOccurs="1"
97                      name="entryPoint" type="EndPoint"/>
98              <xsd:element minOccurs="0" maxOccurs="unbounded"
99                      name="componentInstance" type="ComponentInstance"/>
100             <xsd:element minOccurs="0" maxOccurs="unbounded"
101                     name="dataflowSectionInstance"
102                     type="DataflowSectionInstance"/>
103         </xsd:all>
104     </xsd:complexType>
105
106     <xsd:complexType name="MpiCommunicator">
107         <xsd:choice minOccurs="1" maxOccurs="unbounded">
108             <xsd:element name="mpiEndPoint" type="EndPoint"/>
109         </xsd:choice>
110     </xsd:complexType>
111
112     <xsd:complexType name="Assembly">
113         <xsd:all>
114             <!-- Note: the order is important -->
115             <xsd:element minOccurs="1" maxOccurs="unbounded"
116                     name="mpiProcess" type="MpiProcess"/>
117             <xsd:element minOccurs="0" maxOccurs="unbounded"
118                     name="mpiCommunicator" type="MpiCommunicator"/>
119         </xsd:all>
120     </xsd:complexType>
121
122     <xsd:element name="cmta" type="Assembly"/>
123 </xsd:schema>
```

FIGURE B.5 – Schéma XML (XSD) spécifiant le format des fichiers XML permettant de décrire assemblages de HALLEY.

B.6 Définition de schéma XML pour la description d'assemblages L²C

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!--
    This file is part of LLCMc++ (a.k.a. llcmcpp)
    Copyright (C) 2011 INRIA Julien Bigot <julien.bigot@inria.fr>
    Copyright (C) 2011 INRIA Christian Pérez <christian.perez@inria.fr>

    LLCMc++ is free software: you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published
    by the Free Software Foundation, either version 3 of the License,
    or (at your option) any later version.

    LLCMc++ is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.    See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public
    License along with LLCMc++. If not, see <http://www.gnu.org/licenses/>.
-->
<xsd:schema
     xmlns="http://www.inria.fr/graal/lad/1.0.0"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     targetNamespace="http://www.inria.fr/graal/lad/1.0.0">
    <xsd:element name="lad" type="lad"/>
    <xsd:complexType name="lad">
        <xsd:sequence>
            <xsd:element form="qualified" maxOccurs="unbounded"
                    minOccurs="0" name="mpi" type="mpi_group"/>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="mpi_group">
        <xsd:sequence>
            <xsd:element form="qualified" maxOccurs="unbounded"
                    name="process" type="process"/>
            <xsd:element form="qualified" maxOccurs="unbounded"
                    minOccurs="0" name="communicator"
                    type="mpi_communicator"/>
        </xsd:sequence>
        <xsd:attribute name="id" type="xsd:string"/>
    </xsd:complexType>
    <xsd:complexType name="mpi_communicator">
        <xsd:sequence>
            <xsd:element form="qualified" maxOccurs="unbounded"
                    minOccurs="2" name="peer" type="endpoint"/>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="process">
        <xsd:sequence>
            <xsd:element form="qualified" minOccurs="0"
                    name="start_property" type="endpoint"/>
            <xsd:element form="qualified" maxOccurs="unbounded"
                    name="instance" type="instance"/>
        </xsd:sequence>
        <xsd:attribute name="id" type="xsd:string"/>
    </xsd:complexType>
    <xsd:complexType name="instance">
```

```
57          <xsd:sequence>
58              <xsd:element form="qualified" maxOccurs="unbounded"
59                      minOccurs="0" name="property" type="property_value"/>
60          </xsd:sequence>
61          <xsd:attribute name="from" type="xsd:string"/>
62          <xsd:attribute name="id" type="xsd:ID" use="required"/>
63          <xsd:attribute name="type" type="xsd:string" use="required"/>
64      </xsd:complexType>
65      <xsd:complexType name="property_value">
66          <xsd:sequence>
67              <xsd:choice maxOccurs="unbounded">
68                  <xsd:element form="qualified" minOccurs="0"
69                          name="value" type="data_value"/>
70                  <xsd:element form="qualified" minOccurs="0"
71                          name="cppref" type="endpoint"/>
72                  <xsd:element form="qualified" minOccurs="0"
73                          name="corbaref" type="endpoint"/>
74                  <xsd:element form="qualified" minOccurs="0"
75                          name="systemref" type="system_ref"/>
76              </xsd:choice>
77          </xsd:sequence>
78          <xsd:attribute name="id" type="xsd:string" use="required"/>
79      </xsd:complexType>
80      <xsd:complexType name="data_value">
81          <xsd:simpleContent>
82              <xsd:extension base="xsd:string">
83                  <xsd:attribute name="type" type="data_type" use="required"/>
84              </xsd:extension>
85          </xsd:simpleContent>
86      </xsd:complexType>
87      <xsd:simpleType name="data_type">
88          <xsd:restriction base="xsd:string">
89              <xsd:enumeration value="int8"/>
90              <xsd:enumeration value="uint8"/>
91              <xsd:enumeration value="int16"/>
92              <xsd:enumeration value="uint16"/>
93              <xsd:enumeration value="int32"/>
94              <xsd:enumeration value="uint32"/>
95              <xsd:enumeration value="int64"/>
96              <xsd:enumeration value="uint64"/>
97              <xsd:enumeration value="double"/>
98              <xsd:enumeration value="complexDouble"/>
99              <xsd:enumeration value="string"/>
100         </xsd:restriction>
101     </xsd:simpleType>
102     <xsd:complexType name="endpoint">
103         <xsd:attribute name="instance" type="xsd:IDREF" use="required"/>
104         <xsd:attribute name="property" type="xsd:string" use="required"/>
105     </xsd:complexType>
106     <xsd:complexType name="system_ref">
107         <xsd:attribute name="name" type="xsd:string" use="required"/>
108     </xsd:complexType>
109 </xsd:schema>
```

FIGURE B.6 – Schéma XML (XSD) spécifiant le format des fichiers XML permettant de la décrire assemblages de L²C.

B.7 Exemple de code de composant glu généré issu d'un type de métatâche

```
// All needed includes
#include <comet.hpp>
#include <omp.h>
#include <BlockArray2D.hpp>

// Note:
// BlockArray2D is an opaque type defined by a partitionned-data plugin

// Generated structures
namespace itfGen
{
    struct SampleMt
    {
        struct Fragments
        {
            BlockArray2D::Fragment inout;
        };

        virtual void compute(const Fragments&) = 0;
    };
}

// Note:
// - in L²C, a use port is a pointer to a provide port (object interface)
// - A provide port can be obtained using class inheritance

// A glue component called "comet_metatask_section_SampleMt"
struct comet_metatask_section_SampleMt : public comet::go
{
    // A partitionning port (use port) of the component
    BlockArray2D* part_p0_use;

    // A compute port (use port) of the component
    itfGen::SampleMt* compute_use;


    // Method called by the section to generate tasks for this metatask
    void go()
    {
        // Retrieve the number of tasks to submit
        const int taskCount = part_p0_use->fragmentCount();

        // Submit all the tasks
        for(int i=0 ; i<taskCount ; ++i)
        {
            // Build the list of the fragment need for the task to be executed
            itfGen::SampleMt::Fragments fragments;

            // Retrieve all the fragments from the partitionning components
            // (only one fragment here)
            fragments.inout = part_p0_use->getFragment(i);

            // Retrieve the dependency pointer then transmitted to the OpenMP runtime
            char* dep_inout = (char*)(part_p0_use->getDependency(i));

            // Most OpenMP compilers generate wrong code without copying 'this'
            // Indeed, 'this' is not implicitly copied by most compilers
            // So, the code crash at runtime without it (bug to be reported)
            comet_metatask_section_SampleMt* thisComponent = this;

            // Submit the task and copy the fragments
```

```
62              // (the copy is safe since fragments must be defined as a POD type)
63              #pragma omp task \
64                      depend(inout: dep_inout[0]) \
65                      firstprivate(fragments) \
66                      firstprivate(localParent)
67              thisComponent->compute_use->compute(fragments);
68          }
69      }
70  }
71
72  // Some annotations to declare the component to Comet
73  COMET_COMPO(comet_metatask_section_SampleMt)
74      COMET_PROVIDE(comet::Go, go);
75      COMET_USE(BlockArray2D, part_p0_use);
76      COMET_USE(itfGen::SampleMt, compute_use);
77  COMET_END
```

FIGURE B.7 – Exemple de code d'un composant glu généré issu d'un type de métatâche `SampleMt` utilisant le langage de relations `identity` et possédant un unique port de données `inout` d'entrée-sortie.

B.8 Assemblages HALLEY des cas d'utilisation évalués

B.8.1 Cas d'utilisation EP

```
1   <?xml version="1.0" encoding="UTF-8"?>
2
3   <cmta xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4         xsi:noNamespaceSchemaLocation="comet-cmta.xsd">
5       <mpiProcess>
6           <entryPoint instance="epMaster" port="go"/>
7
8           <componentInstance id="epMaster" type="EpMaster">
9               <use port="sectionCtl">
10                  <endPoint instance="section" port="control"/>
11              </use>
12              <attribute id="dataSizeX" type="uint64">8192</attribute>
13              <attribute id="dataSizeY" type="uint64">8192</attribute>
14              <attribute id="maxIter" type="uint64">20</attribute>
15          </componentInstance>
16
17          <componentInstance id="impl" type="Impl"/>
18
19          <dataflowSectionInstance id="section">
20              <metataskInstance id="epMt1" type="EpMt">
21                  <parameter id="dataSizeX">8192</parameter>
22                  <parameter id="dataSizeY">8192</parameter>
23                  <parameter id="blockSizeX">1024</parameter>
24                  <parameter id="blockSizeY">16</parameter>
25                  <compute instance="impl" port="compute"/>
26              </metataskInstance>
27
28              <metataskInstance id="epMt2" type="EpMt">
29                  <parameter id="dataSizeX">8192</parameter>
30                  <parameter id="dataSizeY">8192</parameter>
31                  <parameter id="blockSizeX">1024</parameter>
```

```
                <parameter id="blockSizeY">16</parameter>
                <compute instance="impl" port="compute"/>
            </metataskInstance>

            <dataConnection>
                <from instance="epMaster" port="inout"/>
                <to instance="epMt1" port="inout"/>
            </dataConnection>
            <dataConnection>
                <from instance="epMt1" port="inout"/>
                <to instance="epMt2" port="inout"/>
            </dataConnection>
            <dataConnection>
                <from instance="epMt2" port="inout"/>
                <to instance="epMaster" port="inout"/>
            </dataConnection>
        </dataflowSectionInstance>
    </mpiProcess>
</cmta>
```

FIGURE B.8 – Code XML de l'assemblage du cas d'utilisation EP. La valeur des paramètres (p. ex. taille des donnée et des blocs) est fixée arbitrairement en guise d'exemple étant donné que les valeurs peuvent varier entre les expériences.

B.8.2 Cas d'utilisation ST

```
<?xml version="1.0" encoding="UTF-8"?>

<cmta xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:noNamespaceSchemaLocation="comet-cmta.xsd">
    <mpiProcess>
        <entryPoint instance="epMaster" port="go"/>

        <componentInstance id="epMaster" type="EpMaster">
            <use port="sectionCtl">
                <endPoint instance="section" port="control"/>
            </use>
            <attribute id="dataSizeX" type="uint64">8192</attribute>
            <attribute id="dataSizeY" type="uint64">8192</attribute>
            <attribute id="maxIter" type="uint64">20</attribute>
        </componentInstance>

        <componentInstance id="impl" type="Impl"/>

        <dataflowSectionInstance id="section">
            <metataskInstance id="epMt1" type="EpMt">
                <parameter id="dataSizeX">8192</parameter>
                <parameter id="dataSizeY">8192</parameter>
                <parameter id="blockSizeX">1024</parameter>
                <parameter id="blockSizeY">16</parameter>
                <compute instance="impl" port="compute"/>
            </metataskInstance>

            <metataskInstance id="epMt2" type="EpMt">
                <parameter id="dataSizeX">8192</parameter>
                <parameter id="dataSizeY">8192</parameter>
                <parameter id="blockSizeX">1024</parameter>
                <parameter id="blockSizeY">16</parameter>
                <compute instance="impl" port="compute"/>
            </metataskInstance>
```

```
35
36              <dataConnection>
37                  <from instance="epMaster" port="inout"/>
38                  <to instance="epMt1" port="inout"/>
39              </dataConnection>
40              <dataConnection>
41                  <from instance="epMt1" port="inout"/>
42                  <to instance="epMt2" port="inout"/>
43              </dataConnection>
44              <dataConnection>
45                  <from instance="epMt2" port="inout"/>
46                  <to instance="epMaster" port="inout"/>
47              </dataConnection>
48          </dataflowSectionInstance>
49      </mpiProcess>
50  </cmta>
```

FIGURE B.9 – Code XML de l'assemblage du cas d'utilisation ST. La valeur des paramètres (p. ex. taille des donnée et des blocs) est fixée arbitrairement en guise d'exemple étant donné que les valeurs peuvent varier entre les expériences.

B.8.3 Cas d'utilisation EP-EP

```
1   <?xml version="1.0" encoding="UTF-8"?>
2
3   <cmta xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4         xsi:noNamespaceSchemaLocation="comet-cmta.xsd">
5       <mpiProcess>
6           <entryPoint instance="epMaster" port="go"/>
7
8           <componentInstance id="epMaster" type="EpMaster">
9               <use port="sectionCtl">
10                  <endPoint instance="section" port="control"/>
11              </use>
12              <attribute id="dataSizeX" type="uint64">8192</attribute>
13              <attribute id="dataSizeY" type="uint64">8192</attribute>
14              <attribute id="maxIter" type="uint64">20</attribute>
15          </componentInstance>
16
17          <componentInstance id="impl" type="Impl"/>
18
19          <dataflowSectionInstance id="section">
20              <metataskInstance id="mt1" type="EpMt">
21                  <parameter id="dataSizeX">8192</parameter>
22                  <parameter id="dataSizeY">8192</parameter>
23                  <parameter id="blockSizeX">1024</parameter>
24                  <parameter id="blockSizeY">64</parameter>
25                  <compute instance="impl" port="compute"/>
26              </metataskInstance>
27
28              <metataskInstance id="mt2" type="EpMt">
29                  <parameter id="dataSizeX">8192</parameter>
30                  <parameter id="dataSizeY">8192</parameter>
31                  <parameter id="blockSizeX">2048</parameter>
32                  <parameter id="blockSizeY">32</parameter>
33                  <compute instance="impl" port="compute"/>
34              </metataskInstance>
35
36              <dataConnection>
37                  <from instance="epMaster" port="inout"/>
38                  <to instance="mt1" port="inout"/>
```

```
            </dataConnection>
            <dataConnection>
                <from instance="mt1" port="inout"/>
                <to instance="mt2" port="inout"/>
            </dataConnection>
            <dataConnection>
                <from instance="mt2" port="inout"/>
                <to instance="epMaster" port="inout"/>
            </dataConnection>
        </dataflowSectionInstance>
    </mpiProcess>
</cmta>
```

FIGURE B.10 – Code XML de l'assemblage du cas d'utilisation EP-EP. La valeur des paramètres (p. ex. taille des donnée et des blocs) est fixée arbitrairement en guise d'exemple étant donné que les valeurs peuvent varier entre les expériences.

B.8.4 Cas d'utilisation ST-ST

```
<?xml version="1.0" encoding="UTF-8"?>

<cmta xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:noNamespaceSchemaLocation="comet-cmta.xsd">
    <mpiProcess>
        <entryPoint instance="stencilMaster" port="go"/>

        <componentInstance id="stencilMaster" type="StencilMasterCompo">
            <use port="sectionCtl">
                <endPoint instance="section" port="control"/>
            </use>
            <attribute id="dataSizeX" type="uint64">8192</attribute>
            <attribute id="dataSizeY" type="uint64">8192</attribute>
            <attribute id="maxIter" type="uint64">20</attribute>
        </componentInstance>

        <componentInstance id="impl" type="Impl"/>

        <dataflowSectionInstance id="section">
            <metataskInstance id="mt1" type="StencilMt">
                <parameter id="dataSizeX">8192</parameter>
                <parameter id="dataSizeY">8192</parameter>
                <parameter id="blockSizeX">1024</parameter>
                <parameter id="blockSizeY">64</parameter>
                <compute instance="impl" port="compute"/>
            </metataskInstance>

            <dataBufferInstance id="tmpData"/>
            <dataBufferConnection instance="mt1" port="out" data="tmpData"/>

            <metataskInstance id="mt2" type="StencilMt">
                <parameter id="dataSizeX">8192</parameter>
                <parameter id="dataSizeY">8192</parameter>
                <parameter id="blockSizeX">2048</parameter>
                <parameter id="blockSizeY">32</parameter>
                <compute instance="impl" port="compute"/>
            </metataskInstance>

            <dataConnection>
                <from instance="stencilMaster" port="inout"/>
                <to instance="mt1" port="in"/>
```

```
            </dataConnection>
            <dataConnection>
                <from instance="mt1" port="out"/>
                <to instance="mt2" port="in"/>
            </dataConnection>
            <dataConnection>
                <from instance="mt2" port="out"/>
                <to instance="stencilMaster" port="inout"/>
            </dataConnection>
        </dataflowSectionInstance>
    </mpiProcess>
</cmta>
```

FIGURE B.11 – Code XML de l'assemblage du cas d'utilisation ST-ST. La valeur des paramètres (p. ex. taille des donnée et des blocs) est fixée arbitrairement en guise d'exemple étant donné que les valeurs peuvent varier entre les expériences.

B.8.5 Cas d'utilisation TRANSP

```
<?xml version="1.0" encoding="UTF-8"?>

<cmta xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:noNamespaceSchemaLocation="comet-cmta.xsd">
    <mpiProcess>
        <entryPoint instance="TranspMaster" port="go"/>

        <componentInstance id="transpMaster" type="TranspMaster">
            <use port="sectionCtl">
                <endPoint instance="section" port="control"/>
            </use>
            <attribute id="dataSizeX" type="uint64">8192</attribute>
            <attribute id="dataSizeY" type="uint64">8192</attribute>
            <attribute id="maxIter" type="uint64">20</attribute>
        </componentInstance>

        <componentInstance id="computeTsk" type="Impl"/>

        <componentInstance id="transposeTsk" type="Impl"/>

        <dataflowSectionInstance id="section">
                <metataskInstance id="computeMt1" type="ComputeMt">
                    <parameter id="dWidth">8192</parameter>
                    <parameter id="dHeight">8192</parameter>
                    <parameter id="bHeight">512</parameter>
                    <compute instance="computeTsk" port="computeLines"/>
                </metataskInstance>

                <metataskInstance id="transposeMt1" type="TransposeMt">
                    <parameter id="dWidth">8192</parameter>
                    <parameter id="dHeight">8192</parameter>
                    <parameter id="bSizeX">16</parameter>
                    <parameter id="bSizeY">16</parameter>
                    <compute instance="transposeTsk" port="computeTransposition"/>
                </metataskInstance>

                <metataskInstance id="computeMt2" type="ComputeMt">
                    <parameter id="dWidth">8192</parameter>
                    <parameter id="dHeight">8192</parameter>
                    <parameter id="bHeight">512</parameter>
                    <compute instance="computeTsk" port="computeLines"/>
```

```
42                  </metataskInstance>
43
44                  <metataskInstance id="transposeMt2" type="TransposeMt">
45                      <parameter id="dWidth">8192</parameter>
46                      <parameter id="dHeight">8192</parameter>
47                      <parameter id="bWidth">16</parameter>
48                      <parameter id="bHeight">16</parameter>
49                      <compute instance="transposeTsk" port="computeTransposition"/>
50                  </metataskInstance>
51
52                  <dataConnection>
53                      <from instance="transpMaster" port="inout"/>
54                      <to instance="computeMt1" port="inout"/>
55                  </dataConnection>
56                  <dataConnection>
57                      <from instance="computeMt1" port="inout"/>
58                      <to instance="transposeMt1" port="inout"/>
59                  </dataConnection>
60                  <dataConnection>
61                      <from instance="transposeMt1" port="inout"/>
62                      <to instance="computeMt2" port="inout"/>
63                  </dataConnection>
64                  <dataConnection>
65                      <from instance="computeMt2" port="inout"/>
66                      <to instance="transposeMt2" port="inout"/>
67                  </dataConnection>
68                  <dataConnection>
69                      <from instance="transposeMt2" port="inout"/>
70                      <to instance="transpMaster" port="inout"/>
71                  </dataConnection>
72          </dataflowSectionInstance>
73      </mpiProcess>
74  </cmta>
```

FIGURE B.12 – Code XML de l'assemblage du cas d'utilisation TRANSP. La valeur des paramètres (p. ex. taille des donnée et des blocs) est fixée arbitrairement en guise d'exemple étant donné que les valeurs peuvent varier entre les expériences.

B.9 Résultats obtenus avec le support exécutif GOMP

Y Taille	#Tâches	EP			SP		
		Référence	HALLEY	Ratio	Référence	HALLEY	Ratio
1024	64	0.402	0.401	0.998	0.603	0.603	1.000
512	128	0.403	0.402	0.998	0.606	0.606	1.000
256	256	0.401	0.401	1.000	0.604	0.602	0.997
128	512	0.400	0.401	1.002	0.603	0.602	0.998
64	1024	0.400	0.401	1.002	0.601	0.602	1.002
32	2048	0.401	0.401	1.000	0.600	0.601	1.002
16	4096	0.671	0.646	0.963	0.606	0.606	1.000
8	8192	1.593	1.618	1.016	1.611	1.624	1.008
4	16384	3.353	3.407	1.016	3.390	3.563	1.051
2	32768	6.857	6.997	1.020	7.120	7.291	1.024
1	65536	13.868	14.076	1.015	14.304	14.794	1.034

TABLE B.1 – Temps de complétion (en secondes) et ratio $\frac{t_{Halley}}{t_{reference}}$ obtenu pour les versions de référence et HALLEY sur les cas d'utilisation EP et ST avec des tailles de blocs variables (1024xY) et un buffer de données de taille 8192x8192.

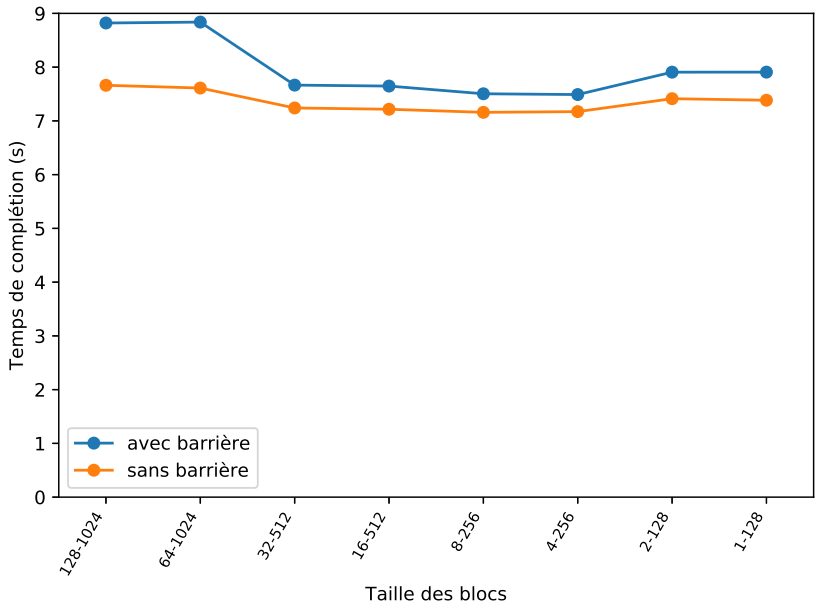

FIGURE B.13 – Temps de complétion du cas d'utilisation TRANSP pour différentes tailles de blocs avec des buffers de taille 8192x8192. Les tailles de blocs sont représentés sous la forme $B_{compute} - B_{transpose}$ où $B_{compute}$ est le nombre de lignes des blocs des métatâches MtCompute et $B_{transpose}$ est la taille des blocs des métatâches MtTranspose.

Taille	#Tâches	EP		ST	
		μs	ratio	μs	ratio
4096 x 4096	128	41.8	1.0013	59.3	1.0000
8192 x 4096	256	41.3	1.0006	55.9	1.0004
16384 x 8192	1024	41.0	1.0007	58.8	1.0004
16384 x 16384	2048	41.2	1.0003	56.5	1.0005

TABLE B.2 – Temps par tâche de la version de référence et ratio $\frac{t_{Halley}}{t_{reference}}$ obtenus sur les cas EP and ST pour de nombreuses configurations de tailles de buffers en considérant une taille de bloc fixée à 1024x128.