- École Normale Supérieure de LYON -
Laboratoire de l'Informatique du Parallélisme

THÈSE

en vue d'obtenir le grade de

Docteur de l'École Normale Supérieure de Lyon - Université de Lyon

Discipline : Informatique

Laboratoire de l'Informatique du Parallélisme
École Doctorale d'Informatique et de Mathématiques

présentée et soutenue publiquement le 5 novembre 2012 par

M. Vincent PICHON

Contribution à la conception à base de composants logiciels d'applications scientifiques parallèles.

Directeur de thèse :	M.	Christian	PÉREZ
Co-encadrant de thèse :	M.	André	RIBES
Après avis de :	M.	Raymond	NAMYST
	M.	Jean-Louis	PAZAT

Devant la commission d'examen formée de :

M.	Raymond	NAMYST	Membre/Rapporteur
M.	Christian	PÉREZ	Membre
M.	Jean-Louis	PAZAT	Membre/Rapporteur
M.	Johan	MONTAGNAT	Membre
M.	Vincent	LEFEBVRE	Membre

Remerciements

Je tiens à remercier ceux qui ont contribué au bon déroulement de cette thèse.

Je remercie les membres de mon jury qui m'ont fait l'honneur d'évaluer les travaux ici présentés : Johan Montagnat qui a présidé le jury, Jean-Louis Pazat et Raymond Namyst qui ont accepté de rapporter cette thèse et Vincent Lefebvre qui a pris part au jury. Je les remercie tous pour leur évaluation.

Je remercie bien sûr Christian Pérez et André Ribes qui m'ont encadré au cours de ces trois ans.

Enfin, je tiens également à remercier les membres des deux équipes qui m'ont accueilli. Tout d'abord le groupe I2A du département SINETICS chez EDF R&D puis l'équipe Avalon au LIP.

Table des matières

Table des figures

Chapitre 1

Introduction

1.1 Conception d'applications scientifiques

Applications scientifiques Les simulations numériques jouent un rôle clé dans un nombre grandissant de domaines de recherche (physique, chimie, biologie, astrophysique, économie, etc.). Elles permettent, grâce à l'évolution des ressources de calcul, de simuler des phénomènes de plus en plus complexes. Ces simulations peuvent être construites par couplage de plusieurs modèles, chacun étant spécialisé dans la simulation d'un phénomène précis.

Par exemple, l'application de simulation d'évolution climatique utilisée dans l'ANR Lego [2] est constituée d'un code de simulation atmosphérique, d'un code de simulation océanique, d'un code de simulation des écoulements fluviaux, d'un code de simulation de la fonte des glaces et d'un coupleur. Chacun de ces codes de simulation numérique a été développé par une équipe de recherche distincte. De plus, les codes ont généralement été conçus avant d'envisager leurs interactions avec d'autres codes. La conception de telles applications est donc une tâche complexe.

Par ailleurs, exécuter efficacement des applications de couplage multi-physiques demande de les adapter aux ressources. C'est une tâche délicate demandant un effort important. Cependant, les ressources de calcul sur lesquelles ces applications vont être exécutées évoluent constamment. Leur puissance de calcul augmente, mais leur architecture change beaucoup. Ainsi, les capacités et caractéristiques des stockages disponibles évoluent ; la topologie réseau des supercalculateurs change souvent d'une machine à l'autre, etc. Par ailleurs, la durée de vie typique d'une application est bien plus grande que celle des ressources matérielles, de l'ordre de trois ans pour une machine et trente ans pour un code. Il est donc nécessaire de régulièrement porter les applications sur de nouvelles machines.

Pour faire face à une telle complexité, l'utilisation d'un modèle de programmation adapté est le bienvenu. Ce modèle de programmation doit simplifier la conception des applications tout en autorisant une exécution efficace. Ainsi, le modèle de programmation doit permettre d'exprimer les interactions et l'architecture d'une application scientifique. Il doit aussi abstraire les ressources d'exécution pour permettre une conception la plus indépendante de celles-ci.

Les modèles de composants logiciels proposent une réponse intéressante à ce problème. Ils permettent d'expliciter les interactions entre les parties d'une application. Comme la programmation objet ou l'usage de librairies, la programmation par modèle de composants logiciels propose de modulariser le code de l'application. Mais contrairement aux objets qui n'expriment que l'interface qu'ils fournissent, les composants logiciels expriment aussi ce dont ils ont besoin.

Ainsi, un composant est une boite noire qui explicite ses besoins et ses contributions par l'intermédiaire de ses ports. La conception d'applications avec un tel modèle de programmation se fait par composition. Les composants logiciels sont assemblés en faisant correspondre

les besoins des uns aux contributions des autres. Ces modèles de programmation permettent de faire apparaître la structure de l'application et d'abstraire sa conception. Les interactions étant clairement définies, chaque composant peut être développé par une équipe différente. Un composant peut également être partagé entre plusieurs applications, ce qui favorise la réutilisation de code. Un composant peut aussi avoir plusieurs implémentations. La définition claire de son interface permet de substituer l'une à l'autre. Les modèles de composants permettent donc de simplifier la conception d'applications scientifiques.

Cependant, les modèles de composants logiciels existants ne répondent pas totalement aux problèmes de la conception d'applications scientifiques. D'une part il ne permettent pas d'abstraire suffisamment la structure des applications scientifiques. D'autre part ils ne permettent pas d'exprimer tous les types de compositions (dynamisme, récursivité) nécessaires à la conception de ces applications.

1.2 Objectifs

L'objectif de cette thèse est d'identifier et de proposer des solutions aux limitations des modèles de programmation par composants logiciels pour la conception d'applications scientifiques.

Simplicité de programmation Un des objectifs d'un modèle de programmation de haut niveau est de simplifier la conception d'applications complexes. Le modèle de programmation doit permettre au développeur de se focaliser sur l'étude des phénomènes liés à son domaine de recherche sans devoir s'occuper des détails d'implémentation, d'autant plus que l'implémentation n'est pas forcément le domaine de compétence de ce chercheur.

Expressivité Un modèle de composants logiciels doit permettre d'exprimer l'ensemble des interactions présentes dans la structure d'une application.

Abstraction des ressources Afin de faciliter le portage de l'application, le modèle de programmation utilisé doit permettre d'abstraire la conception d'applications des ressources visées.

En effet, chaque application doit être optimisée en fonction des ressources d'exécution. Il est donc nécessaire d'adapter certaines parties de l'application. Les modèles de composants logiciels doivent permettre d'exprimer la structure de l'application en distinguant les parties dépendantes des ressources tout en maximisant la réutilisation de code. Dans le cas contraire, il arrive souvent d'avoir une version complète du code spécifique à chaque ressource d'exécution. Il devient alors complexe de propager les évolutions d'une partie de l'application dans toutes les versions de l'application.

Performances Le modèle de programmation doit permettre d'exécuter efficacement l'application conçue. C'est la condition de son utilisation. Pour cela, le modèle de programmation doit permettre à l'assemblage abstrait de composants logiciels de s'adapter finement à la plate-forme d'exécution sans ajouter de surcoût.

Cette thèse s'intéresse aux modèles académiques et industriels. Le modèle de programmation industriel qui est étudié dans cette thèse est celui offert par la plate-forme SALOMÉ. Il possède l'avantage d'être utilisé en production. Il présente par ailleurs des caractéristiques assez proches de modèles académiques de pointe pour nos besoins.

1.3 Contributions

Trois aspects de la programmation par composition ont été étudiés. Le premier est la composition à gros grains avec le modèle de programmation de la plate-forme SALOMÉ. Le second est la composition à grain plus fin avec un modèle de composants logiciels de bas niveau. Enfin, le troisième aborde la composition spatiale et temporelle.

Composition à gros grain Cette première contribution étudie la composition à gros grain au sein de la plate-forme SALOMÉ. Une application de décomposition de domaine, développée par EDF, a servi de base à cette étude. Elle a permis de faire apparaître les limitations de la plate-forme.

Cette application de décomposition de domaine utilise le code de simulation thermo-mécanique Code_ASTER. La première implémentation a été faite directement dans ce code. Cette implémentation ayant été très complexe, une seconde implémentation a été conduite dans l'optique de simplifier la conception. Cette nouvelle implémentation utilise la plate-forme SALOMÉ et ne demande aucune modification de Code_ASTER. Elle a demandé beaucoup moins d'efforts que la précédente. Néanmoins, la conception de ce type d'application n'est pas totalement supportée au sein la plate-forme SALOMÉ. En effet, cette application possède une structure dépendant d'un paramètre : le nombre de sous-domaines. Le modèle de programmation de la plate-forme SALOMÉ ne permet pas d'exprimer cette paramétrisation – des scripts externes sont nécessaires.

Cette contribution propose d'étendre le modèle de programmation de la plate-forme SALOMÉ en ajoutant la possibilité de définir dynamiquement la cardinalité des composants logiciels. Cela demande en particulier de gérer les communications de groupes ainsi induites. Cette extension a été implémentée dans la plate-forme SALOMÉ et validée via l'application de décomposition de domaine présentée ci-dessus. Elle a permis d'augmenter l'expressivité du modèle de programmation et ainsi de simplifier la programmation de ce type d'application.

Adaptation à grain fin Cette contribution étudie l'adéquation d'un modèle de composants logiciels de bas niveau (L^2C) supportant nativement plusieurs connecteurs (appel de méthode, MPI, mémoire partagée) à la conception d'applications de décomposition de domaine. Elle montre qu'un modèle de composants logiciels tel que L^2C permet d'optimiser l'architecture de l'application pour les ressources visées (notamment hétérogènes) tout en présentant les mêmes performances que la version native. Grâce à la comparaison de la conception et des performances de plusieurs versions (natives et à base de composants logiciels) d'un code simple dans différentes configurations matérielles, cette étude met en avant le très faible impact de l'utilisation du modèle de composants logiciels sur les performances ainsi que le gain en expressivité. Cette contribution évalue la pénalité de performance, la complexité, le taux de réutilisation qu'implique l'utilisation d'un tel modèle de programmation.

Raffinement de maillage adaptatif La dernière contribution étudie le support des modèles de composants logiciels à la conception d'une application de raffinement de maillage adaptatif. Ce type d'application est similaire aux applications de décomposition de domaine. En effet, l'application de raffinement de maillage adaptatif effectue une décomposition du domaine de simulation mais celle-ci est hiérarchique et évolue au cours de la simulation. Ainsi, des sous-domaines peuvent être décomposés et remplacés par plusieurs nouveaux sous-domaines. Ce type d'application regroupe des caractéristiques (dynamicité, récursivité et hiérarchie) communes à différentes applications : elle représente un moyen pertinent de tester les limites de ces modèles de programmation pour une classe d'applications scientifiques. Une application de raffinement de

maillage a été implémentée en utilisant un modèle de composants logiciels académique (ULCM) puis un modèle de composants logiciels industriel (SALOMÉ). Cette étude a permis de montrer la faisabilité de ces conceptions et les limites de ces modèles de composants logiciels, identifiant ainsi des directions d'évolution.

1.4 Organisation du document

Ce document est divisé en deux parties. La première partie du document est consacrée à l'étude de l'existant. La seconde partie présente les contributions de cette thèse.

La première partie est composée en trois chapitres. Le chapitre 2 présente d'une part un aperçu des différentes ressources matérielles pouvant être utilisées pour exécuter les applications scientifiques et d'autre part les paradigmes usuels orientés HPC utilisés pour leur conception. Le chapitre 3 se focalise sur les modèles de composition, qu'ils soient spatiaux, temporels ou une combinaison des deux. Enfin, le chapitre 4 présente des environnements de programmation spécialisés pour le développement d'applications scientifiques.

La seconde partie présente les trois cas d'étude de cette thèse. Le chapitre 5 présente la première contribution, à savoir l'extension du modèle de programmation de la plate-forme SALOMÉ pour permettre le support d'applications du type décomposition de domaine. Le chapitre 6 étudie la pertinence de l'utilisation d'un modèle de composants logiciels supportant des connecteurs natifs (MPI, mémoire partagée, appel de méthode) pour une conception plus fine des interactions entre composants logiciels. Enfin, le chapitre 7 présente l'étude des extensions nécessaires (abstraction, dynamicité, hiérarchie) au support de la conception d'un type d'application de raffinement de maillage adaptatif.

1.5 Publications

1.5.1 Publication dans un atelier international

- André Ribes, Christian Pérez, and Vincent Pichon. – On the Design of Adaptive Mesh Refinement Applications based on Software Components. In *Proceedings of the 2010 Workshop on Component-Based High Performance Computing* (CBHPC 2010). – Brussels, Belgium. 25-29 October 2010. Collocated with Grid 2010.

1.5.2 Articles en cours de soumission dans des conférences internationales

- André Ribes, Christian Pérez, and Vincent Pichon. - Easing Domain Decomposition Application Programming within SALOME with Node and Port Cloning.
- Julien Bigot, Christian Perez, Zhengxiong Hou, Vincent Pichon. - A low level component Model enabling resource specialization of HPC Application.

Première partie

Contexte

Chapitre 2

Ressources matérielles et paradigmes de programmation

2.1 Introduction

L'évolution des ressources matérielles permet d'envisager la conception d'applications scientifiques modélisant des phénomènes de plus en plus complexes. Les simulations numériques multi-physiques par couplage de codes en sont un bon exemple. La complexification croissante de ces applications est due à deux phénomènes : le nombre et la forme des modèles utilisés les constituants et l'évolution des ressources matérielles. Ce chapitre se décompose en deux parties. La première présente un aperçu des caractéristiques des différentes ressources matérielles susceptibles d'exécuter une application scientifique. La seconde partie présente les principaux paradigmes de programmation utilisés pour la conception de ce genre d'application.

2.2 Ressources matérielles

La puissance des ressources matérielles demandée par les applications scientifiques est de plus en plus importante. Elles nécessites des processeurs de plus en plus puissants, des capacités de stockage plus importantes ainsi que des communications entre unités de calcul de plus en plus rapides. Cette section a pour but de donner un aperçu des différentes architectures disponibles pour le calcul scientifique.

2.2.1 Des mono-processeurs aux machines parallèles

La plus simple des ressources de calcul est l'ordinateur mono-processeur mono-coeur. Cette ressource ne permet qu'un seul fil d'exécution. C'est l'unité de base de toute les formes de parallélisation présentées par la suite. Sa puissance de calcul est en particulier caractérisée par la fréquence de son horloge. L'énergie dissipée dépendant en principalement de la fréquence du processeur, elle est devenue un frein à son augmentation. Les concepteurs de processeurs se sont donc orientés vers le parallélisme et les architectures multi-coeur pour augmenter la puissance de calcul. De plus, à un instant donné de l'évolution technologique, le seul moyen d'augmenter le nombre d'instructions exécutées est d'avoir plusieurs unités de calcul en parallèle. Ainsi, certains processeurs mono-coeur peuvent avoir plusieurs unités flottantes.

Deux stratégies sont possibles : les multi-processeurs (multi coeur), c'est-à-dire augmenter le nombre de processeurs par ordinateur (SMP, NUMA ou GPU) ou les multi-ordinateurs qui regroupent plusieurs ordinateurs au sein d'un plus grand ensemble (Grappes, supercalculateurs, grilles, cloud).

2.2.2 Machines multi-processeurs

SMP Une machine SMP (Symetric MultiProcessing) possède plusieurs processeurs permettant ainsi plusieurs fils d'exécutions simultanés. Dans une telle architecture la mémoire est partagée entre tous les processeurs et chacun y accède par le même bus. Ce bus devient le goulot d'étranglement de l'application. En effet, lorsque les fils d'exécutions d'une application accèdent en même temps à la mémoire le débit du bus diminue. Ce qui limite le passage à l'échelle en nombre de processeurs.

NUMA L'architecture NUMA (Non Uniform Memory Access) est une solution aux limitations de l'architecture SMP. Celle-ci partitionne la mémoire et l'associe à chaque processeur. Cette approche limite le nombre de processeur pouvant accéder en même temps à une portion de mémoire. L'ensemble de la mémoire reste accessible à tous les processeurs mais plus la mémoire est éloignée d'un processeur plus son accès est lent. Par exemple, dans certaines architectures (Intel Xeon, AMD Opteron), la mémoire associée à un ensemble de processeurs n'est accessible aux autres que par l'intermédiaire des premiers.

Cette architecture nécessite d'adapter l'exécution des applications en descendant à un niveau très bas dans les détails de conception. En effet, si on veut profiter des temps accès mémoire rapides il faut que les fils d'exécutions qui utilisent des données partagées soient placés sur des processeurs voisins dans la hiérarchie de mémoires.

2.2.3 GPU

Le GPU (Graphics Processing Unit) est un circuit intégré présent sur les cartes graphiques qui présente une architecture parallèle. Il suit le paradigme Single Instruction Multiple Data (SIMD) c'est-à-dire qu'il permet d'exécuter une même instruction sur plusieurs données. Son rôle premier est d'accélérer l'exécution des tâches graphiques (à base de calcul vectoriel). Les GPU étant initialement prévus pour les applications graphiques (jeux vidéo), ils bénéficient de débouchés commerciaux importants, ce qui en fait une ressource matérielle peu onéreuse.

Cependant, les applications graphiques ne sont pas les seules à pouvoir bénéficier de cette architecture matérielle. Il est possible d'utiliser le GPU pour exécuter les applications parallèles. C'est le GPGPU (General-purpose computing on graphics processing units). Il permet d'obtenir une grande puissance de calcul en utilisant un GPU ou une combinaison de GPU.

Le GPU possède une mémoire propre et est relié au processeur et à la mémoire par l'intermédiaire du bus PCI. Ce bus a des performances moins élevées que le bus mémoire. Le GPU est composé d'une grande quantité d'unités de calcul dont le jeu d'instruction est plus simple que celui d'un processeur classique. Cette architecture est orientée vers l'exécution d'instructions en parallèle sur un ensemble de données.

Exécuter des applications sur une telle architecture nécessite de prendre en compte l'ensemble de ces nouveaux paramètres. Il faut pouvoir spécifier quelles sont les instructions à exécuter sur le processeur central, celles qui peuvent être parallélisées sur le GPU et les transferts de données entre les mémoires du GPU et du processeur. Cela demande encore une fois de contrôler très précisément l'exécution de l'application.

2.2.4 Grappes

Une grappe est un ensemble d'ordinateurs indépendants qui apparaît à l'utilisateur comme un seul système. Elle est constituée d'un ensemble de serveurs ou de stations de travail assez homogène – chaque ordinateur d'une grappe est appelé noeud. Ils sont interconnectés par un réseau local (Local Area Network, LAN) reposant sur la technologie Gigabit Ethernet. Ce réseau

est parfois doublé d'un second plus performant basé sur des technologies comme Quadrics, Myrinet ou InfiniBand. Dans ce cas le premier est utilisé pour les tâches communes (telles que l'administration).

Une grappe nécessite un investissement moins important qu'un serveur multiprocesseur pour un même nombre de coeurs. Il a de plus l'avantage d'être extensible. L'inconvénient majeur est que les communications entre les noeuds d'une grappe sont beaucoup moins performantes qu'au sein d'un ordinateur multi-processeur.

Afin d'utiliser pleinement les capacités d'une telle architecture il est nécessaire d'adapter les applications à la topologie réseau de la grappe. Suivant les topologies réseau utilisées les communications entres certains noeuds peuvent varier énormément.

Une grappe peut être partagée entre plusieurs utilisateurs notamment grâce aux gestionnaires de tâches. UNIX possède son gestionnaire de tâche `at` mais il en existe d'autre comme OAR [53], PBS [35] ou LoadLeveler [77]. Ces gestionnaires de ressources permettent de réserver un ensemble de noeuds pour une période donnée. Une grappe peut aussi être gérée par un système d'exploitation à image unique (Single Image System, SSI) comme Kerrighed [84] ou Mosix [33] qui permet de voir la grappe comme un seul serveur fortement NUMA.

2.2.5 Supercalculateurs

Les supercalculateurs sont conçus pour exécuter des tâches nécessitant une très grande puissance de calcul comme, par exemple, les simulations numériques de météorologie, climat, cosmologie... Ils sont très onéreux et demandent généralement une infrastructure spécifique.

Top500 Le Top500 [23] propose un classement des 500 plus puissants supercalculateurs. Les performances de ces supercalculateurs sont évaluées grâce au test de performance Linpack (http ://www.netlib.org/linpack/hpl/) et le classement s'effectue suivant le nombre d'opérations à virgule flottante par seconde (FLoating point Operations Per Second, FLOPS).

Sequoia Le 14 juin 2012, le plus puissant supercalculateur du top500 était Sequoia (un supercalculateur de la série IBM Blue Gene/Q) avec une puissance théorique maximale de 20132.7 pétaFLOPS. Il possède 98304 noeud de calcul, 1572864 coeurs et 1.6Po de mémoire. Ses processeurs sont des Power Architecture à 16 ou 8 noeuds. Son système d'exploitation est Linux.

K Computer Le K Computeur est actuellement le second plus puissant supercalculateur du Top500 avec une puissance théorique maximale de 11.2804 pétaFLOPS. Il possède 88128 processeurs de 8 coeurs chacun, soit un total de 705024 coeurs. Son système d'exploitation est Linux. Le K computer possède un réseau toroïdal et utilise une version optimisée pour cette topologie de la librairie OpenMPI.

Tianhe-IA Tianhe-IA est un supercalculateur qui utilise l'architecture GPGPU. Il est équipé de 14336 processeurs Xeon et de 7168 GPU Tesla. Sa puissance théorique est de 4,701 pétaFLOPS. Sur chaque carte deux processeurs Xéon sont associés à un GPU Tesla. Ces processeurs sont connectés par un réseau propriétaire spécialement conçu pour ce supercalculateur.

Tera 100 Tera-100 est la première machine française du classement Top500 qui est située à la 17ème place. C'est un supercalculateur fabriqué par Bull. Il est composé de 4370 noeuds et rassemble au total 138368 coeurs Xeon. Son système d'exploitation est linux. Sa puissance théorique est de 1254.55 pétaFLOPS. Et il utilise un réseau d'interconnexion InfiniBand sur une topologie constituée d'une grappe de *flat tree*.

Ces supercalculateurs nécessitent des investissements très importants. Néanmoins leur durée de vie est inférieure aux codes qu'ils exécutent. Comme le montrent les quelques exemples issus du Top500, l'architecture d'un supercalculateur peut varier énormément de l'un à l'autre. Tous les paramètres changent : les processeurs, la topologie réseau, la présence ou non de GPU. Cette très forte hétérogénéité implique un effort de portage des applications. Il est ainsi nécessaire d'adapter les applications à chacun de ces supercalculateur.

2.2.6 Grilles

2.2.6.1 Computing grid

Une grille de calcul est un type d'architecture matérielle qui permet aux institutions de mutualiser leurs ressources matérielles pour résoudre des problèmes de plus grande taille. Cela permet aussi aux institutions ayant des moyens financiers limités d'avoir accès à une grande puissance de calcul pour un moindre coût.

Une grille de calcul fédère les ressources de calcul des différentes institutions y prenant part. Elle est donc souvent hétérogène. En effet, les ressources partagées ne sont pas du même type car acquises et maintenues par les différentes institutions de la grille. Il n'y a par ailleurs pas de politique centralisée de gestion de ces ressources. Une grille peut ainsi être constituée de grappes de serveurs, de supercalculateurs ou de simples stations de travail. Ces ressources de calcul varient par le nombre et l'architecture des noeuds, les réseaux les interconnectant et leurs capacités de stockage.

Une première définition de la grille de calcul à été donnée dans les années 90 par Foster et Kesselman [69] :

> "A computational grid is a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities."

Ces deux auteurs ont ensuite fait évoluer cette définition [70] :

> "Grid computing is concerned with coordinated resource sharing and problem solving in dynamic, multi-institutional virtual orgranizations "

Dans cette dernière version les auteurs mettent l'accent sur le caractère partagé de cette architecture matérielle. Finalement, Ian Foster propose une définition de la grille en trois points [68] : Une grille de calcul est un système qui :

- coordonne des ressources non soumises à un contrôle central,
- en utilisant des protocoles et interfaces standardisés et ouverts
- pour offrir des services non triviaux. C'est-à-dire dont l'utilité est supérieure à la somme des utilités des sous-services la constituant.

La suite de cette sous-section présente différents exemples de grilles.

EGI, European Grid Infrastructure [19] est un projet Européen dont l'objectif est de fournir des ressources de calcul aux chercheurs européens (industriels ou scientifiques) issus de divers champs de recherche. Cette grille est répartie entre presque 350 centres, rassemblant plus de 300000 coeurs et totalisant une capacité de stockage de plus de 100 Po, le tout reposant sur le réseau GÉANT[20] qui offre jusqu'à 10Go/s entre certains sites.

XSEDE Extreme Science and Engineering Discovery Environment (XSEDE) [98] est la suite du projet TeraGrid [22]. Il est financé par le National Science Foundation (NSF). XSEDE est une grille répartie sur le territoire américain qui est constituée de grappes de PCs, de machines

vectorielles, de supercalculateurs. Les sites sont interconnectés par une réseau à haut débit de 40 Gbit/s. La puissance de calcul est de plus 2 PetaFLOPS et la capacité de stockage de 30 Po.

Plate-forme expérimentale Grid'5000 Aladdin Grid'5000 [97] est une plate-forme expérimentale disponible pour la recherche sur les systèmes parallèles et/ou distribués. À ce jour Grid'5000 est composée de grappes disponibles sur 10 sites : Lille, Rennes, Orsay, Nancy, Bordeaux, Lyon, Grenoble, Toulouse, Reims et Sophia-Antipolis. Les grappes sont constituées soit d'AMD (Opteron), soit d'Intel (Xeon). Grid'5000 rassemble en juin 2012 de 1594 noeuds soit 3064 processeurs ou 8700 coeurs. Les différents sites de la plate-forme sont reliés par le réseau Renater[99] qui offre un débit pouvant aller jusqu'à 10Go/s entre la plupart des sites. Les réseaux des grappes constituant Grid'5000 sont de type Gigabit Ethernet, Myrinet, ou InfiniBand.

Chaque grappe est gérée par le gestionnaire de ressource OAR. Il est possible d'utiliser le système d'exploitation installé sur les noeuds (debian linux) ou de déployer son propre système d'exploitation.

2.2.6.2 Desktop grid

La desktop grid est une autre forme de grille[81]. Elle propose d'utiliser les ressources inutilisées des ordinateurs personnels connectés à internet. En effet ces ordinateurs n'utilisent qu'environ 10% de leurs ressources de calcul lorsqu'ils sont utilisés et pratiquement 0% lorsqu'ils sont en veille.

World Community Grid World Community Grid est un exemple de ce genre de grille. Elle est constitué de 400000 membres. Un volontaire désirant participer à cette grille doit installer un logiciel mettant sont ordinateur à disposition des utilisateurs. Ce logiciel est paramétrable. Il permet au volontaire de spécifier le taux de partage de son ordinateur. C'est à dire le taux maximal d'utilisation du processeur, la taille de la mémoire partagée ainsi que les plages horaires pendant lesquelles le partage est effectif. Le volontaire peut aussi arrêter l'application à tout moment.

Ce type de ressource peut rassembler des centaines de milliers d'ordinateurs et présente une grande hétérogénéité, les ordinateurs des volontaires ayant des caractéristiques matérielles très dissemblables. Le taux de partage d'une ressource, la bande passante et les latences des ordinateurs volontaires ainsi que la volatilité des ressources (un volontaire peut éteindre son ordinateur à tout moment) sont aussi des paramètres à prendre en compte lors du portage d'une application vers ce genre de ressource.

Des environnements de programmation comme XtremWeb [54] ou BOINC [4] (utilisé par World Community Grid) facilitent ce genre de développement. Cette configuration matérielle convient particulièrement bien aux applications de type *parameter sweep*. Cependant, la pertinence de ce type de plate-forme pour d'autres types d'application (par exemple mapreduce) est à l'étude. Notamment, avec BitDew[65]

2.2.7 Cloud

Le cloud computing propose à l'utilisateur de ne payer que les ressources qu'il consomme effectivement. L'utilisateur n'a plus besoin d'avoir un ensemble de ressources toujours disponibles et cela même lorsqu'il n'a aucune tâche à exécuter. Le cloud lui propose un nombre de ressources théoriquement infini – en pratique assez limité. Lorsque l'utilisateur a besoin de ressources de calcul, le cloud lui fournit la quantité de ressources dont il a besoin. Une fois le calcul terminé,

les ressources sont libérées. L'utilisateur est facturé relativement au nombre d'heures utilisées par serveur. Le prix des ressources varie avec le taux d'utilisation du cloud. Il est intégré, ainsi que la demande de calcul, dans le choix du nombre de ressources à utiliser.

Les clouds peuvent être classifiés en différents niveaux de services :

- SaaS (Software as a Service) : Ce niveau propose d'accéder à des services à la demande. Ces services sont accessible via une interface web ou une API. Un exemple est le Microsoft Online Services qui propose Exchange.
- PaaS (Platform as a Service) : Dans cette forme de cloud, l'utilisateur a en charge le développement ou le paramétrage des applications. Les ressources exécutant ces applications sont, elles, maintenues par le cloud. Des exemples de ce type de cloud sont Google App Engine et Microsoft Azure.
- IaaS (Infrastructure as a Service) : dans ce niveau de service les utilisateurs ont accès a un environnement d'exécution par le biais de machines virtuelles. Ce type de service est proposé par Amazon EC2[18] par exemple.

Dans le contexte du cloud computing, le nombre de ressources disponibles est inconnu. Concevoir une application pour cette ressource implique donc de prendre en compte la souplesse de celle-ci.

HPC@Cloud Amazon EC2 offre aussi des ressources compatibles avec le HPC. Ainsi, l'utilisateur peut choisir ses ressources dans une grappe de calcul ou de GPU (bande passante élevée, une faible latence réseau et des capacités de calcul élevées).

L'utilisation du cloud pour le HPC permet par ailleurs le chiffrage du coût d'utilisation des ressources. Les expériences récentes [24] établissent un prix de 5000$ par heure pour 50000 noeuds de calcul.

FutureGrid FutureGrid[21] est un projet américain financé par la National Science Foundation (NSF) dont le but est de fournir une plate-forme pour la recherche dans le domaine des grilles de calcul et du *cloud computing*. À cette fin, le projet FutureGrid rassemble plus de 5600 noeuds répartis entre six sites. FutureGrid est intégré dans TeraGrid[22]. Ses ressources sont accessibles à travers des outils comme Nimbus[79], OpenNebula[13], Eucalyptus[10].

2.2.8 Discussion

Cette section a présenté un aperçu des différentes ressources disponibles pour les applications scientifiques. Deux aspects se dégagent de cette présentation. Premièrement, le caractère hétérogène de leur ensemble, deuxièmement leur évolution continue.

Hétérogénéité des ressources L'ensemble des ressources matérielles décrit dans cette section présente des caractéristiques très hétérogènes. Les processeurs, la hiérarchie de la mémoire et la topologie du réseau est différente d'une architecture à une autre. Concevoir une application ayant une performance optimale pour ces architectures nécessite d'adapter ses différentes parties à chacune d'entre elles. Les communications, le parallélisme, le partage de données doivent être adaptés à chaque architecture.

Évolution rapide Ces ressources évoluent très rapidement, bien plus rapidement que les applications qu'elles exécutent. La puissance de calcul augmente régulièrement. Mais les autres paramètres évoluent aussi. Ainsi, le nombre de processeurs par ordinateur, l'architecture des processeurs, la vitesse des réseaux et les topologies proposés évoluent. De plus, de nouvelles

architectures matérielles apparaissent régulièrement. Celles-ci présentent des caractéristiques nouvelles qui impliquent un effort de portage.

2.3 Paradigmes de programmation d'applications scientifiques

2.3.1 Introduction

La section précédente a présentée un aperçu des principaux types de ressources matérielles disponibles. Concevoir des applications destinées à de telles ressources peut se faire de différentes manières. Cette section présente les paradigmes de programmation les plus utilisés. Ils ont été classifiés en deux catégories : mémoire partagée et mémoire distribuée.

Dans un modèle de programmation à mémoire partagée, les fils d'exécutions accèdent à un même espace mémoire qui leur permet d'échanger des données au cours de l'exécution. Dans un modèle à mémoire distribuée, chaque fil d'exécution possède son propre espace d'adressage. Le partage de données entres fils d'exécutions d'un modèle à mémoire distribué se fait par envoi explicite de données entre les différents fils d'exécutions.

2.3.2 Mémoire partagée

La mémoire partagée peut être une mémoire physique partagée comme dans le cas d'une machine multiprocesseurs. Elle peut également être constituée de mémoire distribuées. Dans ce cas, l'utilisation d'un mécanisme matériel ou logiciel de Distributed Shared Memory (DSM) offre l'illusion d'un espace d'adressage commun. Les modèles de programmation associés à la mémoire partagée sont présentés dans la sous-section suivante.

2.3.2.1 Multithread

Le premier paradigme de programmation étudié est le modèle multithread, c'est-à-dire un modèle de programmation qui permet à une application d'avoir plusieurs fils d'exécution (*thread*) qui interagissent via une mémoire commune.

2.3.2.2 Pthread

Pthread (POSIX Threads) est un standard qui définit une Api pour la création et la gestion des threads. Son implémentation est disponible sur les systèmes d'exploitation compatibles POSIX mais il existe aussi des implémentations pour Windows ou DOS.

Avec l'Api Pthread il est possible de créer et joindre les threads. Des zones d'exclusions mutuelles entre threads peuvent être définies grâce aux mutex. Finalement la synchronisation des threads peut se faire grâce à des verrous, des sémaphores ou des barrières. De plus, ce modèle de programmation permet grâce a des extensions comme la bibliothèque NUMActl de contrôler le placement des threads sur les coeurs et des données sur des zones mémoire.

L'ensemble des fonctions proposées par le standard Pthread et ses extensions permet de gérer très finement l'exécution des threads. Cependant, le programmeur est obligé de se préoccuper de détails d'exécution de très bas niveau. Ceci requiert une connaissance de l'architecture matérielle et une importante connaissance de l'architecture du logiciel à paralléliser.

2.3.2.3 OpenMP

OpenMP (Open Multi-Processing) est un langage à base de directives défini pour C, C++ et fortran, qui permet de spécifier les portions de code à paralléliser. C'est un modèle de

programmation de plus haut niveau que Pthread, l'utilisateur n'a plus à se soucier de la création et de la gestion des threads. Ils sont automatiquement générés par OpenMP suivant les directives fournies par le développeur et un ensemble de variables d'environnement. Le développement d'applications multi-threads avec OpenMP est plus simple et plus rapide qu'avec Pthread, mais permet aussi moins de contrôle sur le parallélisme de l'application. En effet, c'est l'intergiciel de l'implémentation d'OpenMP utilisée qui contrôle la création et le nombre de threads.

2.3.2.4 GPGPU

L'utilisation des processeurs GPU nécessite un modèle de programmation particulier. Il est en effet nécessaire de gérer la répartition des tâches entre le CPU et le GPU ainsi que la parallélisation des tâches sur le GPU.

CUDA CUDA (Compute Unified Device Architecture) est une technologie développée par NVIDIA. Elle permet de programmer un GPU en langage C et plus récemment en C++.

CUDA propose une boite à outils comprenant un compilateur, des librairies et des outils d'optimisation et de débogage. L'Api de CUDA permet de spécifier les instructions du programme à exécuter sur le GPU. C'est CUDA qui se charge automatiquement de créer les threads sur le GPU et de contrôler leurs exécutions.

Porter une application pour l'exécuter sur GPU en utilisant CUDA implique d'utiliser tous ces outils. Il faut initialiser l'environnement CUDA, distinguer les portions de code parallélisables et compiler l'application avec le compilateur CUDA.

OpenCL OpenCL (Open Computing Language) est un standard proposé par le Khronos Group qui est composé d'un langage de programmation proche du C et d'une Api. Son objectif est de proposer un modèle de programmation pour les GPU et les architectures multi-coeurs. Des implémentations de ce standard sont disponibles pour les architectures courantes (Intel, AMD, NVIDIA, ARM). Son modèle de programmation est très proche de celui de CUDA.

OpenACC OpenACC est un autre standard de programmation développé pour la programmation parallèle sur CPU-GPU. Il a été développé par CAPS, Cray, NVIDIA et PGI. Il s'utilise d'une manière similaire à OpenMP. Des directives sont placées dans le code de l'application pour spécifier les portions qui seront exécutées sur le GPU. Les langages pour lesquels cette technologie est proposée sont le C, C++ et Fortran.

2.3.3 Mémoire Distribuée

Les ressources distribuées permettent l'exécution simultanée dans des espaces mémoires différents. Le partage de données entre exécutables se fait donc explicitement soit par passage de message, soit par appel de méthode distante.

2.3.3.1 Passage de messages

Le modèle de programmation par passage de messages est un des plus utilisés dans le calcul scientifique. Il permet d'écrire des applications devant s'exécuter sur des systèmes à mémoire distribuée. Les ressources sont présentées de manière abstraites : c'est le modèle de programmation qui prend en charge l'optimisation des communications entre les processus de l'application. Le développeur ne doit se soucier que de la parallélisation et des communications entre les tâches

dans son application. Les deux environnements de programmation par passage de message les plus courants sont MPI [66] (Message-Passing Interface) et PVM[72] (Parallel Virtual Machine).

MPI MPI permet de définir des applications constituées d'un seul programme (SPMD) et des applications contenants différents exécutables comme dans le cas du couplage de code (OASIS, ARPÈGE...). MPI propose une API dont deux versions sont principalement utilisées. Une troisième version de cette API est en cours de finalisation.

MPI-1 La première version de l'API MPI [66] proposée par le MPI forum est MPI-1. Cette version propose un ensemble de concepts dont voici les principaux.

- Le **communicateur** permet de grouper les processus MPI d'une session. Les processus MPI d'une application ne peuvent communiquer qu'avec ceux ayant le même communicateur.
- Les **communications point à point** permettent à deux processus MPI de communiquer directement. La famille de fonctions `MPI_send` et `MPI_recv` permettent ces communications. Elles prennent en paramètre le rang de processus MPI source ou destination.
- MPI-1 propose aussi un ensemble de fonctions pour les communications collectives. Par exemple, `MPI_Bcast` envoie un message à tous les processus du communicateur et `MPI_reduce` attend les données de tous les processus MPI et effectue une opération de réduction sur ces données.
- Cette version de MPI propose également des types de données dérivés. Ils permettent de définir des vecteurs, des zones de mémoire contiguës ou des structures qui peuvent ensuite être envoyés entre processus.

MPI-2 Cette seconde version de l'API MPI [67] offre principalement la possibilité d'ajouter des processus MPI à une application en exécution ainsi que les communications *one-sided*. Les processus ajoutés sont soit issus de l'application MPI en exécution en utilisant la fonction `MPI_Comm_spawn`, soit en rattachant des processus créés à l'extérieur de l'application avec `MPI_Comm_acceppt/connect/join`.

Dans MPI-1, le passage de messages entre deux processus se fait par rendez-vous. Le processus qui envoie la donnée utilise `MPI_Send` et celui qui reçoit la donnée utilise `MPI_Recv`. Les communication *one-sided* ajoute la possibilité de communication sans rendez-vous. Un processus MPI peut ainsi accéder à la mémoire d'un autre processus MPI grâce notamment aux primitives `MPI_Get` et `MPI_Put`.

MPI-3 Les perspectives pour la prochaine version de l'API MPI sont une meilleure prise en charge des architectures distribuées multi-coeurs. Actuellement le meilleur compromis pour optimiser l'exécution d'une application sur une architecture distribuée de processeurs multi-coeurs est de coupler MPI et OpenMP. MPI est utilisé pour le parallélisme entre processeurs et OpenMP entre coeurs d'un même processeur en utilisant la mémoire partagée. De plus MPI-3 proposera des primitives pour les opérations collectives asynchrones.

Le placement des processus MPI d'une application est un élément crucial pour optimiser ses performances. Cependant, adapter une application MPI aux ressources est une tâche complexe [74].

PVM PVM permet de considérer un ensemble hétérogène de ressources comme une machine parallèle virtuelle. Cet environnement de programmation permet les opérations collectives (barrière, broadcast, reduce, gather, scatter) et les communications de point à point. Contrairement à MPI, PVM ne permet pas de définir de ses propres types de données.

2.3.3.2 Appel de procédure distante

L'appel de procédure distante (Remote Procecdure Call, RPC) [44] est un modèle de programmation qui permet au développeur d'appeler des méthodes s'exécutant sur un hôte distant. L'appel d'une procédure distante se fait de manière similaire à un appel de procédure locale. Il est néanmoins nécessaire d'adapter les arguments de la fonction. En effet, il est impossible de passer un pointeur en paramètre à une procédure distante. Par ailleurs, il existe plusieurs types d'appel de procédure distante : synchrone ou asynchrone.

Le standard RPC est l'ONC-RPC développé par Sun Microsystems[92]. La conception d'une application utilisant l'appel de procédure distante demande l'utilisation d'un langage de définition d'interfaces (Interface Definition Langage, IDL). Ce langage permet de définir l'interface de communication entre les différentes parties de l'application. La partie client et la partie serveur utilisent l'IDL pour définir quel sont les fonctions et les arguments de celles-ci. L'IDL définit également pour chaque argument d'une fonction si celui-ci est passé par référence ou par valeur. L'IDL est ensuite utilisé pour générer les souches des client et serveur de l'application. La construction de l'application se fait en ajoutant la partie fonctionnelle à ces souches. Dans ce modèle, chaque appel distant demande d'expliciter la destination.

GridRPC GridRPC [91] a été proposé par le groupe de travail GridRPC de l'OGF (Open Grid Forum). Il propose une Api pour la conception d'applications reposant sur l'appel de procédure distante devant s'exécuter sur les grilles de calcul. La principale différence par rapport au RPC classique est qu'il n'est pas obligatoire d'expliciter la destination : un intergiciel implémentant ce standard, comme par exemple DIET[55], peut choisir le meilleur serveur pour exécuter la procédure et se charger de la transmission des données entre celui-ci et le client.

2.3.3.3 Appel de méthode distante

L'appel de méthode distante (Remote Method Invocation, RMI) applique les principes du RPC à la programmation objet. Il ne s'agit plus de faire référence à des procédures distantes mais à des objets distants. Une application construite sur le modèle d'appel de méthode distante est une application constituée d'objets distants faisant référence les uns aux autres. La conception d'une application utilisant l'appel de méthode distante suit le même principe que celui pour l'appel de procédure distante.

Corba Corba [87] (Common Object Request Broker Architecture) est un standard défini par l'OMG [100]. Ce standard spécifie un intergiciel permettant la conception d'applications utilisant le paradigme d'appel de méthode distante. Un des avantages de Corba est qu'il permet de gérer l'hétérogénéité des machines, des langages de programmation, des implémentations et des réseaux. Corba repose sur le concept d'ORB (Object Request Broker) qui permet la communication entre les différents objets distribués de l'application. Le protocole de communication entre les ORBs fait partie de la norme Corba. Les objets distribués d'une application sont en général décrits grâce à un IDL qui est utilisé pour générer les souches des objets de l'application. Le code de l'application est finalement complété en utilisant les objets distants comme s'ils étaient locaux.

2.3.3.4 Appel de service

L'appel de service est un modèle de programmation proche de l'appel de méthode distante. En effet, celui-ci propose de construire une application à partir de services distants. Mais au lieu de faire un appel de méthode, il procède par passage de document.

Les *Web Services* [45] sont un ensemble de spécifications éditées par le W3C [101]. Les communications entre services se fait par passage de documents. Les documents supportés pas un service sont décrits grâce au langage WSDL [57] basé sur XML [49]. L'échange des documents entres services repose principalement sur le protocole standard d'échange de message SOAP [48] (Simple Object Access Protocol). Par ailleurs, les services sont disponibles grâce à des serveurs de nommage du type UDDI [36]. Les *Web Services* permettent une indépendance vis-à-vis des langages de programmation et des plate-formes.

2.3.4 Discussion

Les différents modèles de programmation présentés dans cette section offrent tous un niveau d'abstraction masquant les aspects techniques. Ainsi, le partage d'un espace d'adressage (notamment à partir d'espaces distribués), les protocoles de communication, les communications collectives, etc, sont des détails pris en charge par les modèles de programmation. Néanmoins, ces modèles différent par le niveau d'abstraction offert. Par exemple, OpenMP est de plus haut niveau que Pthread. Augmenter le niveau d'abstraction facilite la conception d'applications mais réduit les possibilités de contrôle. Le choix du modèle de programmation dépend donc des objectifs visés par le développeur.

2.4 Analyse

2.4.1 Ressources hétérogènes

Les ressources présentées dans la première section sont très hétérogènes. Chacune pouvant être utilisée avec différents modèles de programmation. De plus, certaines ressources présentent une forte hétérogénéité intrinsèque. Les supercalculateurs peuvent combiner des multi-coeurs avec GPU. Les grilles assemblent des grappes, des supercalculateurs, des ordinateurs personnels, etc. Concevoir une application pour ces ressources implique donc l'usage simultané de plusieurs modèles de programmation. L'application devant s'adapter lors de son exécution.

Difficulté de programmation Cette conception demande donc une connaissance de tous les paradigmes. Elle demande aussi de concevoir l'application de telle sorte qu'un des paradigmes puisse facilement remplacer un autre afin d'éviter d'avoir une version de l'application lié à chaque paradigme. Cette conception devient donc très complexe.

2.4.2 Évolution rapide des ressources

Effort constant de portage L'évolution des ressources est plus rapide que celle des codes de calcul. Les applications doivent donc être régulièrement portées vers les nouvelles architectures. C'est une tâche complexe qui prend souvent plusieurs mois voir années. Par exemple, à peine l'adaptation d'applications pour des architectures à base d'un ensemble de noeuds multi-coeur – à base de MPI et OpenMP– a-t-il été terminé, que la question se pose de les adapter pour prendre en compte des architectures avec des GPGPU. Ces portages étant très coûteux, seul un petit ensemble d'applications peut suivre cette évolution forcée.

Nombre de ressources inconnues et variable Dans certaines ressources comme le cloud ou la desktop grid le nombre de ressources à disposition est inconnu et peut varier au cours de l'exécution de l'application. Prendre en compte se paramètre demande d'abstraire les ressources matérielles.

Paramètre énergie Comme l'atteste le Green 500 [25], qui présente les 500 plus puissantes machines classées suivant l'efficience de leur gestion d'énergie, un nouveau paramètre est de plus en plus pris en compte. Il s'agit de l'efficacité énergétique. Les supercalculateurs sont de plus en plus puissants mais consomment de plus en plus d'énergie. L'énergie devient un facteur limitant dans les conceptions de machines de calcul. Ainsi, l'objectif majeur pour la machine exaflopique est de consommer au maximum environ 20 MW. De plus, le coût de l'énergie augmentant, il devient important de la minimiser. L'énergie consommée est donc un nouveau paramètre à intégrer lors du développement d'une application. Le modèle de programmation, par une meilleure abstraction des ressources peut faciliter la prise en compte de ce paramètre.

2.5 Conclusion

Dans ce chapitre nous avons passé en revue les principales ressources de calcul disponibles ainsi que différents paradigmes de programmation pour les applications scientifiques HPC. L'analyse nous a montré que l'hétérogénéité et l'évolution rapide de ces ressources confrontent le développeur à une grande complexité de programmation. En effet, il doit maîtriser toutes les architectures et tous les paradigmes pour permettre à son application de s'adapter à toutes les situations.

De plus, les applications suivent l'évolution des ressources. Elles sont elles-mêmes de plus en plus complexes. L'augmentation de puissance de calcul permet d'envisager des simuler des problèmes plus complexes, notamment grâce au couplage de code. Cette complexité peut-être dépassée en utilisant des modèles de programmation permettant d'abstraire la conception d'applications de leur ressources.

Le chapitre suivant présente des modèles de programmation de plus haut niveau.

Chapitre 3

Modèles de composition

Le chapitre précédent a montré que l'hétérogénéité des ressources et la complexité des applications induisent une grande complexité de programmation. Les modèles de programmation par composition sont des modèles de plus haut niveau que ceux présenté dans le chapitre précédent. Ils ont pour objectif d'apporter une solution à cette complexité de programmation.

Ce chapitre présente quelques modèles de programmation utilisant la composition spatiale ou la composition temporelle. Les compositions spatiales et temporelles sont deux approches orthogonales de la composition. La composition spatiale permet de définir la structure d'une application à base de composants logiciels indépendamment de tout aspect temporel. Ces modèles, sont appelés modèles de composants logiciels. Ils sont présentés dans la première partie de ce chapitre. La composition temporelle implique des liens entre tâches conditionnant leur ordre d'exécution. Les modèles de composition temporelle sont appelés modèles de *workflow*. Ils sont présentés dans la seconde partie de ce chapitre.

Enfin, la troisième partie présente les modèles combinant les deux types de composition : spatiale et temporelle.

3.1 Composition spatiale : Modèles de composants

Cette section est consacrée à la composition spatiale. Après quelques définitions relatives aux modèles de composants logiciels, plusieurs exemples de modèles sont présentés.

3.1.1 Définitions

Composant logiciel Le terme de composant est utilisé dans plusieurs domaines de recherche. Sa définition varie beaucoup de l'un à l'autre. La définition utilisée ici doit donc être précisée. Nous utilisons celle proposée par Clemens Szyperski [93] :

> "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 composition by third parties."

Un composant logiciel est donc une unité de code réutilisable, composable et déployable qui est vue comme une boite noire. Un composant logiciel définit des ports qui lui permettent d'interagir avec d'autres composants et l'environnement. L'interaction des composants, suit le paradigme fournit-utilise. Il est principalement basé sur les modèles de communications suivants :

- Appel de méthode
- Passage de message

- Passage de document
- Événements

Port Les ports sont les points d'interactions d'un composant logiciel. Ils sont en général typés par une interface. C'est grâce à eux qu'un composant logiciel peut exprimer quelles sont les interfaces qu'il fournit (c'est-à-dire implémente) ou utilise.

Assemblage La constitution d'une application se fait par assemblage de composants. C'est-à-dire en connectant les ports fournit/utilise des ces composants. Cet assemblage peut être décrit par un langage de description d'application (Architecture Description Language, ADL). Celui-ci spécifie les instances de composants de l'application et les connexions de ports à effectuer. L'assemblage peut aussi se faire dynamiquement au cours de l'exécution de l'application. Dans ce cas les composants utilisent des interfaces d'assemblage fournies par la plate-forme implémentant le modèle de composants. Par ailleurs, l'assemblage peut être hiérarchique. Ainsi, certains composants peuvent être constitués d'un assemblage d'autres composants. Certains modèles de composants logiciels permettent aussi définir des attributs permettant de configurer les propriétés d'un composant.

Après ces quelques définitions, la suite de cette section est consacrée à la description de plusieurs modèles de composants logiciels.

3.1.2 CCM

Le CORBA Component Model (CCM) est défini par la spécification version 4.0 de CORBA [17]. Ce modèle spécifie des composants distribués et hétérogènes, c'est à dire indépendants des langages, des systèmes d'exploitations et des fournisseurs d'intergiciels. À cette fin, les fonctions métier et gestion d'un composant sont donc séparées. Par ailleurs, CCM spécifie le cycle de vie complet des composants.

Composants Un composant CCM peut avoir différents types de ports :

- Les ports **facette** et **réceptacle** fournissent ou utilisent une interface. Cette interface est celle d'un objet CORBA.
- Les ports **sources** et **puits** permettent le transfert asynchrone de données entre composants.

Un composant CCM peut aussi définir des propriétés configurables appelées *attributs*. L'interface globale d'un composant est composée de ses ports et attributs. Elle est définie dans le langage IDL3 (extension du langage de description d'interfaces CORBA).

CCM spécifie aussi l'implémentation d'un composant. C'est-à-dire les aspects non fonctionnels (transaction, persistance, sécurité). Ainsi, la partie fonctionnelle d'un composant est placée dans un **exécuteur**, les instances d'un composant sont crées par une **fabrique** et sont gérées par une **maison**. Ces aspects non fonctionnels du composant sont générés automatiquement à partir du Common Implementation Definition Language (CIDL).

Assemblage CCM fournit un langage d'assemblage au format XML. Ce langage permet de définir les instances de composants et les connexions des leurs ports. Il permet aussi de fixer des contraintes sur la localisation des composants. Depuis la version 4.0, il est possible de définir des composants composites, c'est à dire dont l'implémentation est un assemblage d'autres composants.

Une limitation importante du modèle CCM est qu'il ne permet pas de définir des composants parallèles.

3.1.3 FRACTAL

Fractal [50] est un modèle de composants logiciels développé dans le cadre du consortium OW2 [12] par France Telecom R&D et l'INRIA. Il propose une spécification et un ensemble d'implémentations de référence dans différents langages (Java, C, C++, .NET, SmallTalk, Python).

Le modèle propose les concepts suivants :
- Composants composites
- Composants partagés
- Introspection
- Configuration et reconfiguration

Objectif Le modèle de composants logiciels Fractal a pour objectif de pouvoir s'appliquer aux systèmes de différentes tailles, de l'embarqué au système d'information. C'est pour cette raison que dans Fractal tout est optionnel. Fractal consiste en un ensemble de concepts et une Api que les composants peuvent implémenter suivant leurs besoins. Le modèle de composants logiciels Fractal est structuré en niveaux de contrôles. Au niveau de contrôle le plus faible un composant est juste un objet qui implémente une interface. Le niveau suivant ajoute l'introspection. Un composant Fractal peut ainsi fournir une interface standard qui permet de découvrir toutes ses interfaces extérieures. Ensuite vient le niveau configuration dans lequel un composant Fractal propose une interface de contrôle qui permet l'introspection et la modification de son contenu.

Concepts Un composant est constitué de deux parties : le contenu et la membrane. Le contenu d'un composant Fractal est soit constitué par son implémentation dans le cas d'un composant primitif, soit par un ensemble d'autres composants dans le cas d'un composant composite. La membrane permet de séparer l'extérieur de l'intérieur d'un composant. C'est également elle qui expose les ports du composant. Ceux-ci peuvent être internes (dans le cas d'un composant composite) ou externes. Les ports internes d'un composant composite permettent de relier les ports externes des composants dont l'assemblage le constitue.

Dans le modèle Fractal, il est possible de spécifier si la présence d'une interface est essentielle à l'exécution d'un composant. Il est aussi possible de spécifier le nombre d'interfaces d'un certain type qu'un composant possède.

Par ailleurs, Fractal impose au concepteur de prendre en compte les aspects fonctionnels et non fonctionnels d'un composant. Cela à l'avantage de rendre le côté non fonctionnel extensible mais au prix d'une augmentation de la complexité de programmation. Fractal ne propose pas de langage de description d'architecture mais propose une Api qui permet l'instanciation des composants à l'exécution.

3.1.4 GCM

Gcm [34] est une extension du modèle de composants logiciels Fractal destinée à un environnement de type grille de calcul. La première extension consiste à proposer un Adl permettant lier les composants à des ressources abstraites. L'association à des ressources réelles ne s'effectue qu'au moment de déployer l'application. Le *framework* et son ordonnanceur choisissent le placement des composants en fonction des ressources disponibles et des informations contenues dans l'Adl.

Gcm apporte aussi une solution au problème des communications collectives. Gcm ajoute les interfaces de type *multicast* et *gathercast* au modèle Fractal. Ainsi les opérations *broadcast* et *scatter* sont implémentées dans la membrane des composants. Ces opérations et l'extension de

l'interface de contrôle permettent à GCM de proposer une solution au couplage de composants parallèles. Ainsi, lors de la connexion de deux composants parallèles, l'interface de contrôle est utilisée pour partager le nombre de composants, la taille et le type des données, le schéma de partage, etc. C'est cette interface qui établit alors le lien direct entre chaque sous-composant des deux composants parallèles.

3.1.5 CCA

Le Common Component Architecture [39, 32] (CCA) est modèle de composants logiciels proposé par le CCA Forum qui regroupe principalement des chercheurs de laboratoires nationaux américains. Ce modèle a été conçu pour le calcul haute performance.

La spécification CCA, comme un composant CCA, est écrite en Scientific Interface Définition Language (Sidl). Elle définit l'interface du *framework* d'exécution des composants CCA. Cette interface permet de créer, connecter et gérer les composants CCA.

Composants Un composant CCA peut fournir (port `provides`) ou utiliser (port `uses`) une interface. Il définit son interface globale (l'ensemble de ses ports) en utilisant le Sidl qui est ensuite converti en souche du composant ne nécessitant plus que l'ajout des parties fonctionnelles. La compatibilité entre les différents langages de programmation est obtenue grâce à l'outil Babel [80].

ADL CCA ne possède pas d'ADL. L'assemblage se fait dynamiquement. Pour cela, chaque composant CCA utilise une interface implémentée par le *framework*. Cette interface est appelée `Services`. Elle permet aux composants d'exposer un port `provides` et le rendant accessible aux autres composants. Elle permet aussi d'obtenir les ports `provides` correspondants aux ports `uses` des composants. Cette absence d'ADL permet de créer dynamiquement de nouveaux ports mais implique une connaissance des mécanismes internes aux composants. Les composants ne sont donc plus des boites noires et la structure de l'application à base de composants CCA n'est plus explicite.

Parallélisme Le modèle de composants logiciels CCA a été défini orthogonalement au concept de parallélisme. Un composant CCA est défini de la même manière qu'il soit parallèle ou séquentiel. Le parallélisme est à la charge du développeur (en utilisant MPI ou PVM par exemple). Le *framework* SCIRun2[75] étend les spécifications CCA en permettant de spécifier dans l'interface des composants des ports `uses` et `provides` parallèles.

3.1.6 HLCM

High Level Component Model (HLCM) [42, 43] est un modèle de composants logiciels hiérarchique générique dont l'exécution des composants primitifs repose sur un autre modèle de composant. Il abstrait les relations entre composants en adaptant le concept connecteur[52, 83] à un modèle de composants logiciels hiérarchique. Ainsi, il permet de définir des interactions de composants plus complexes que les connexions de ports `uses`/`provides` des précédents modèles de composants logiciels. HLCM utilise la transformation de modèle pour générer un assemblage concret dans un modèle de composants logiciels existant à partir d'un assemblage abstrait de composants et connecteurs HLCM.

Types de composants Un composant primitif HLCM dépend du modèle de composants logiciels cible. Si, par exemple, le modèle cible est CCM, les composants en auront toutes les

caractéristiques. De plus, HLCM est hiérarchique, c'est à dire qu'il supporte le concept de composant composite. Un composant composite est un assemblage d'instances de composants et de connexions.

Connecteurs Les connecteurs définissent un type d'interaction. Ils contiennent un ensemble de rôles qui ont un nom et une multiplicité (simple ou multiple). Dans les connexions, les rôles sont remplis par des ports, et les rôles multiples par plusieurs ports.

Connexions Les connexions sont des instances de *connecteurs*. Un connecteur est implémenté par un générateur. Comme un composant primitif peut avoir différentes implémentations, un connecteur peur avoir plusieurs générateurs.

De plus, ces générateurs peuvent être primitifs ou composites. Les générateurs primitifs correspondent aux interactions du modèle de composants logiciels sous-jacent (ports `uses` et `provides` de CCM par exemple). Les générateurs composites génèrent un assemblage de composants et de connecteurs.

Implémentations des composants Une application HLCM est donc constituée d'un ensemble de composants, de connecteurs, de générateurs et par le modèle de composants logiciels sous-jacent. La génération de l'application dans le modèle de composants logiciels sous-jacent consiste à remplacer récursivement tous les composites (composants ou générateurs) par leur contenu.

Low Level Component Low Level Component L^2C est un modèle de composants logiciels bas niveau qui a été développé conjointement à HLCM dans l'objectif de réduire au minimum les pénalités de performances d'un modèle de composants logiciels primitif en C++. Les composants L^2C supportent les ports de type `uses/provides` pour les compositions par interfaces C++ et CORBA. Pour les interactions basées sur MPI le modèle propose une primitive pour la définition du communicateur.

Les composants L^2C définissent des points d'entrée et de sortie. Les points d'entrée peuvent être utilisés pour configurer le composant et les points de sortie pour récupérer des informations en vue de la configuration d'autres composants. Par exemple, lors d'un appel de méthode entre composants, les points de configuration permettent de transmettre le pointeur de la fonction au composant qui en a besoin.

L^2C possède un langage d'assemblage (sous format XML) et une API. Une application peut donc soit être décrite avec ce langage soit avec l'API, en connectant les points de configuration des composants.

Lors du lancement de l'application, la partie *runtime* de L^2C instancie les composants, utilise les points de configuration pour configurer tous les composants et finalement leur transfert le contrôle. Ainsi, lors de l'exécution, il ne reste aucun code L^2C entre les composants. Le surcoût est donc celui d'un appel de méthode virtuelle pour un appel de méthode C++, de l'utilisation d'une variable pour le communicateur MPI et nul dans le cas d'un appel de méthode CORBA.

3.1.7 Analyse

Mode de communication Certains modèles de composants logiciels comme CCM et CCA reposent sur l'appel de méthode distante. Celui-ci permet une abstraction de l'hétérogénéité des ressources mais au prix d'une perte de performances.

Hiérarchie Les modèles FRACTAL et HLCM proposent le concept de composant composite, ce qui permet de simplifier la structuration d'une application.

Langage d'assemblage CCM, FRACTAL et HLCM possèdent un langage d'assemblage. La présence de celui-ci permet d'expliciter la structure d'une l'application, ce qui facilite la conception et la réutilisation de celle-ci. Cela permet aussi de prévoir les ressources nécessaires lors du déploiement de l'application.

CCA ne possède pas d'ADL. La structure des applications construites avec ce modèle de composants logiciels est donc moins facile à comprendre. Cela demande une connaissance du fonctionnement interne des composants et complexifie l'ordonnancement de l'exécution de l'application. Cependant, ceci autorise CCA à adopter un comportement très dynamique. Par ailleurs, l'aspect temporel est absent des ces langages d'assemblage. Même si la structure de l'application est rendue explicite par l'ADL, il ne fournit aucune information sur l'ordre d'exécution des composants. Ces modèles ne permettent donc pas d'optimiser le déploiement et l'usage des ressources d'une application, contrairement aux modèles de flux de travail.

La section suivante présente quelques modèles de composition temporelle.

3.2 Composition temporelle : Modèles de flux de travail

Les modèles de flux de travail (*workflow*) ont pour but d'automatiser l'exécution de différentes tâches d'une application en prenant en compte les dépendances temporelles entre elles. Cette section a pour but de présenter les caractéristiques de quelques modèles de flux de travail existants.

Après quelques définitions relatives aux modèles de flux de travail, plusieurs exemples de modèles sont présentés.

3.2.1 Définitions

Un modèle de flux de travail permet de définir l'exécution coordonnée d'un ensemble de tâches. Ce flux de travail est exécuté par un moteur de flux de travail.

Tâche Une tâche est l'unité de base de composition d'un flux de travail. Elle est vue comme une boîte noire et est indépendante des autres tâches. Elle peut donc être réutilisée.

Ports Les ports permettent de définir le flux de données entre les tâches. Ils peuvent être d'entrée ou de sortie. Un port d'entrée apporte une donnée nécessaire à une tâche. L'arrivée de cette donnée déclenche l'exécution de cette tâche. Le port de sortie permet de transmettre une donnée produite par l'exécution d'une tâche. Les ports sont généralement typés (entier, objet, nombre flottant, etc).

Composition La composition de tâches consiste à établir le flot de données et le flot de contrôle entre les tâches. Le flot de données est établi en connectant les ports d'entrée et les ports de sortie des différentes tâches. Le flot de contrôle spécifie l'ordre d'exécution des tâches. Il peut comprendre la séquence, le branchement conditionnel, les boucles, les sections parallèles ou être induit par les flots de données. D'autre part, l'assemblage peut être hiérarchique. Une tâche peut être constituée par un sous-flux de travail.

La suite de cette section présente quelques exemples de modèles de flux de travail.

3.2.2 KEPLER

KEPLER [30] est un logiciel développé initialement par l'université de Berkely (Californie) qui permet de concevoir des *workflow* scientifiques. Il est défini au dessus de l'environnement de conception PTOLEMY II[61]. La construction d'un *workflow* dans KEPLER se fait grâce à une interface graphique. Elle permet de connecter les ports d'entrée et de sortie des tâches. Les *workflow* ainsi définis sont sauvegardés au format `XML`. Les *workflow* de KEPLER peuvent être composés de trois entités (*directors*, *actors* et *parameters*) et de ports connectés.

Actors Dans le modèle KEPLER, chaque tâche est appelée composant ou *actor*. Chaque *actor* peut avoir des ports d'entrée ou sortie qui peuvent être multiples. Ces ports sont connectés par un lien appelé *channel*. KEPLER supporte le concept de *workflow* hiérarchique : un *actor* peut être composite, c'est à dire qu'il peut contenir un sous-*workflow* d'*actors*. KEPLER propose un ensemble d'*actors* prêt à l'emploi (mathématiques, statistiques, traitement du signal, entrée-sortie, affichage, etc.). Il est aussi possible de créer des composants exécutant un script (langage R ou Matlab) ou de créer des composants à partir de programmes existants.

Director Un *director* contrôle l'exécution d'un *worlflow*. Par exemple, le *SDF Director* (*Synchronous Dataflow Director*) autorise l'exécution d'un composant à la fois, le *PN Director* (*Process Network Director*) autorise l'exécution de composants en parallèle, etc. KEPLER propose un ensemble de *director* mais d'autres sont accessibles à travers Ptolemy II. C'est le *director* qui permet de définir le flot de contrôle du *workflow*. Celui-ci est découplé du flux de données. L'objectif de cette stratégie est de pouvoir appliquer différents flots de contrôle à un même flux de données.

Paramètres Les paramètres sont des valeurs attachées globalement à un *workflow*, a un *director* ou à un *actor*. Ces paramètres peuvent aussi être fixés par les ports de sortie des actors.

3.2.3 Triana

Triana [94] est un modèle de *workflow* développé à l'université de Cardiff. Il propose une interface graphique pour la conception de *workflow*. Il propose, comme KEPLER, un ensemble de tâches prédéfinies en mathématique, traitement du signal, etc. Il a été étendu avec des fonctionnalités destinés au calcul distribué. Triana permet la construction de *workflow* de type non-DAG hiérarchiques. Ainsi, il propose les structures de contrôle comme la séquence, le branchement conditionnel et l'exécution parallèle. Les structures de contrôle de Triana sont considérés comme des tâches. Elles sont donc modifiables et il est possible d'en créer de nouvelles. Les tâches locales ont en entrée et sortie des données qui sont des objets Java.

3.2.4 AGWL

Abstract Grid Workflow (AGWL) [63] est un langage de *workflow* destiné à un environnement de grilles de calcul. Il s'exécute dans l'environnement Askalon [62].

Ce langage, au format XML, est généré grâce à une interface graphique. Son objectif est d'abstraire la définition du *workflow* des ressources et des technologies utilisées pour implémenter les tâches. C'est le moteur de *workflow* qui a pour rôle de transformer le *workflow* abstrait AGWL en un *workflow* concret, c'est à dire en associant les tâches aux ressources.

Ce langage de *workflow* propose un flot de contrôle comprenant les structures de contrôle if, switch, for, while, parallelFor, Foreach. De plus il supporte les *workflow* hiérarchiques.

3.2.5 MOTEUR

hoMe-made OpTimisEd scUfl enactoR (MOTEUR) [73] est un environnement d'exécution développé en Java sous licence CeCILL dont l'objectif est de permettre de définir des workflows expressifs et de les exécuter efficacement sur des grilles de calcul.

Un *workflow* destiné à MOTEUR est décrit dans le langage Simple Conceptual Unified Flow Language (SCUFL) et sauvé au format XML. Le langage SCUFL permet de définir des tâches (*processor*) avec des ports d'entrée et sortie permettant de spécifier le flot de données. Un second type de tâche (*coordination constraint*) ne transmet aucune donnée. Son rôle est de permettre la transmission d'un flot de contrôle. SCUFL permet également de spécifier le nombre d'itérations qu'un *processor* doit exécuter. Par ailleurs, les structures de contrôle peuvent être définies en utilisant le *processor fail-if-false/true*.

Le *workflow*, peut être développé en utilisant une interface graphique proposée par MOTEUR. Son exécution est centralisée et repose sur l'utilisation de l'intergiciel gLite.

3.2.6 ASSIST

ASSIST [28] propose un environnement de programmation parallèle. Il est basé sur un langage de coordination (ASSISTcl) supportant des squelettes algorithmiques, un ensemble d'outils de compilation et d'exécution et des librairies.

Une application ASSIST est composée de plusieurs sections. La première décrit un graphe de modules dans le langage ASSISTcl (proche du C et du Pascal). Ces modules sont connectés par des flux de données typés. La seconde contient le code des modules qui peuvent être séquentiels ou parallèles. Les modules séquentiels sont écrit en C, C++ ou FORTRAN 77. Les modules parallèles, appelés *parmod*, sont utilisés pour exécuter parallèlement des modules séquentiels. Le comportement d'un module parallèle peut être spécialisé suivant certains schémas (fermes, pipelines, etc). Il permet de définir un ensemble de processeurs virtuels, de leur assigner des tâches, de gérer l'accès concurrent aux variables d'état, de gérer les flux d'entrée et de sortie, et d'interagir avec l'environnement extérieur par l'intermédiaire d'appels de méthodes (par exemple Corba). La programmation d'un module parallèle se fait avec le langage ASSISTcl et ses primitives.

Le compilateur ASSIST convertit le graphe de modules en méta données (en XML) et les codes des modules en exécutables. Les modules séquentiels deviennent des processus séquentiels et les modules parallèles deviennent un réseau de processus. À l'exécution, ce réseau de processus est géré par un Module Adaptation Manager (MAM). Celui-ci contrôle le nombre de processus du réseau, sa configuration, etc. en fonction des paramètres du programme ASSISTcl.

3.2.7 Analyse

Les modèles de *workflow* présentés dans la section précédente se distinguent sur plusieurs points.

Flot de contrôle Le flot de contrôle s'exprime de façon plus ou moins explicite d'un modèle à l'autre. Ainsi dans Kepler il est découplé du flot de données, dans Triana il est intégré comme tâche alors que dans AWGL le flot de contrôle est explicite. Plus le flot de contrôle est explicite, plus le moteur de *workflow* peut optimiser son exécution.

Abstraction des ressources Tous les modèles de *workflow* présentés permettent de définir des *workflow* abstraits, c'est à dire découplés des ressources matérielles qui vont les exécuter.

Application parallèles Le cas des applications composées de tâches parallèles reste un problème. En effet, aucun des modèles présentés dans cette section n'offre de solution pour gérer la communication et la redistribution de données entre plusieurs tâches parallèles.

3.3 Composition spatiale et temporelle

3.3.1 Introduction

Les deux sections précédentes ont présenté des modèles proposant un seul type de composition. La première, reposant sur la composition spatiale, est plus adapté au couplage "fort". La seconde, reposant sur la composition temporelle, permet d'optimiser l'usage des ressources et le déploiement d'une application. Afin de capturer l'ensemble de la structure d'une application (spatiale et temporelle) et de bénéficier ainsi des avantages de ces deux approches, plusieurs modèles combinent ces deux modes de composition. La section suivante présente quelques exemples de ces modèles.

3.3.2 ICENI

IMperial College e-Science Networked Infrastructure (ICENI)[71] est un modèle de composants logiciels destiné aux grilles de calcul. Il a pour objectif de permettre l'ordonnancement des composants. Pour cela, le modèle de composants logiciels permet d'associer des méta-données à chaque composant. Ces méta-données comprennent une description sous forme de *workflow* du comportement interne du composant. Elles permettent de choisir entre différentes implémentations d'un composant. Ce choix est fait au déploiement de l'application par le framework. Celui-ci utilise les méta-données de tous les composants pour optimiser le placement des composants et choisir leur implémentation.

Ce modèle suppose une connaissance du fonctionnement interne des composants. Le développeur doit connaître le comportement du composant et le décrire dans les méta-données. Les composants ne sont donc plus vu comme des boites noires. De plus, le *workflow* décrivant le comportement des composants est contenu dans ses méta-données. La structure temporelle d'une application avec ce genre de composant n'est donc pas rendue explicite.

3.3.3 ULCM

Unified Lego Component Model (Ulcm) [31] est un modèle de composants logiciels abstrait qui étend les modèles de composants logiciels classiques avec de nouvelles fonctionnalités. Il est basés sur Spatio-Temporal Component Model (Stcm) [47], un modèle de composants logiciels dont l'objectif est d'unifier les aspects spatiaux des modèles de composants logiciels et les aspects temporels de la composition dans les modèles de *workflow*. Ulcm apporte aussi des solutions aux problèmes étudiées dans l'ANR Lego : le partage de données et le support du paradigme master-worker.

Ce modèle est basé sur les concepts standards de composants primitif et composite, port, instance et connexion. Afin de permettre d'étudier de nouvelles relations entre composants, les spécifications de Ulcm ne sont pas liées à un modèle de composants logiciels particulier.

Composants Dans Ulcm un composant est un type défini grâce au mot clé `component`. Ce type contient deux parties : la liste des ports et une description de son contenu. Ulcm supporte plusieurs types de ports : les ports spatiaux (qui permettent de fournir ou utiliser une interface) et les ports temporels (qui permettent de transmettre une donnée et un flux de contrôle d'un composant à un autre).

Ulcm supporte deux types de composant : primitif ou composite. Un composant primitif est défini en Ulcm grâce au mot clé `primitive`. Il permet de spécifier l'implémentation du composant. Dans le modèle Ulcm les composants sont des boites noires. Leur contenu n'est donc pas accessible depuis l'extérieur. De plus les composants Ulcm sont abstraits. En effet, aucune contrainte n'est mise sur les composants primitifs. Ils peuvent être issus d'autres modèles comme Ccm, Cca, Sca ou être une classe Java.

Un composant composite est un composant dont l'implémentation est un assemblage d'autres composants (primitifs ou composites). La structure du composite peut être étendue avec une description temporelle.

Assemblage Ulcm repose sur un langage d'assemblage permettant de décrire l'architecture d'une application à base de composants d'un point de vue temporel et spatial. L'assemblage peut contenir trois sortes d'informations :

- la définition de nouveaux types de composants,
- la déclaration d'instances de composants
- et la configuration de ports.

Dans Ulcm un composant peut fournir (serveur) ou utiliser (client) un service.

Composition temporelle Ulcm définit également deux ports temporels : *input* et *output* attachés chacun à un type de donnée. Les composants-tâches doivent implémenter une opération *task* qui est exécutée lorsque ses dépendances temporelles le permettent.

Ulcm fournit des structures de contrôle pour la création et la destruction d'instances de composants, l'exécution de tâches, la connexion et déconnexion de ports, l'initialisation d'attributs : séquence, section parallèle, condition, boucle séquentielle, boucle parallèle et boucle conditionnelle.

Modèle transparent d'accès aux données Les modèles de composants logiciels courants ne supportent que l'appel de méthode ou le passage de message. Ils impliquent des opérations de communications entre composants. Ils ne permettent donc pas de partager facilement des données entre composants. Ulcm permet la définition de ports de données. Un port de donnée permet de lier logiquement une donnée partagée à un composant. Il est ainsi possible d'exprimer qu'un composant fournit une donnée partagée ou a besoin d'accéder à une donnée partagée. Les interfaces de ces ports spécifient précisément les interactions entre composants.

Paradigme maître-travailleur Ulcm introduit le concept de *collection* pour proposer l'abstraction du paradigme maître-travailleur. Une *collection* permet de définir de manière abstraite un ensemble de composants connectés à d'autres composants. Le nombre de composants de cet ensemble n'est connu que lors du déploiement.

Premièrement on définit une architecture abstraite dans laquelle on spécifie une *collection* de composants travailleurs. Quand l'environnement de déploiement est connu le nombre de travailleurs est fixé et la politique de transport de la requête est choisie.

3.3.4 STKM

Le modèle STKM [27, 26] a été crée dans le but d'unifier les modèles de composants logiciels avec les modèles de *workflow* et les squelettes algorithmiques. Il est basé sur les concepts de Stcm. Ses composants implémentent donc aussi des tâches et ont des ports spatiaux et temporels. La composition d'une application avec le modèle STKM se fait avec un Adl qui propose des structures de contrôle (séquences, branches, boucles) et des squelettes algorithmiques (pipe,

ferme, réplication, etc) qui peuvent être assemblés avec des composants ou d'autres squelettes. STKM est un modèle hiérarchique. En effet, les squelettes peuvent être eux-mêmes constitués d'assemblages de squelettes.

3.3.5 Analyse

Les modèles à composition spatiale et temporelle permettent de mieux expliciter la structure d'une application. En effet, d'une part la composition des codes constituants l'application est directement accessible. D'autre part, dans un modèle comme ULCM l'algorithme de l'application est lui aussi visible. Ainsi, il devient plus aisé de modifier une partie de l'assemblage et l'algorithme de l'application.

Le modèle de composants logiciels ULCM présente l'avantage de s'abstraire du modèle de composants logiciels concret sur lequel il repose. Par contre, ce modèle ne possède pas le concept de connecteur. Il ne permet donc pas d'exprimer des compositions complexes comme peut le faire HLCM.

3.4 Conclusion

Cette section a présenté un aperçu des différents modèles de composition. Les modèles de composants logiciels, les modèles de *workflow* puis les modèles de composition spatiale et temporelle. Ces modèles proposent des solutions intéressantes au problème de la complexité croissante de la conception et de la maintenance des applications scientifiques. En effet, chaque composant exprime clairement ses interactions. Contrairement au modèles présentés dans le chapitre précédent, la construction d'une application peut se faire par assemblage de codes pré-existants. Les composants pouvant être développés par des spécialistes de chaque domaine.

Néanmoins, les modèles de composants existants ne permettent pas de concevoir efficacement certains types d'applications parallèles comme la décomposition de domaine. En effet, aucun des modèles présentés n'offre le niveau d'abstraction nécessaire à la conception de ce type d'application.

Bien que simplifiant la conception d'applications scientifiques, les modèles de composants ne sont généralement utilisés que par les spécialistes en architecture logicielle. Il est en effet nécessaire au développeur d'un composant de connaître les concepts de composant, ADL, port, framework, etc. C'est pour cette raison que d'autres modèles de programmation spécialisés existent. Ces modèles sont généralement focalisés sur la conception d'un type particulier d'applications. Ce qui leur permet d'offrir un environnement de développement simplifié. Ils ne permettent donc pas, contrairement aux modèles de composition de ce chapitre, de concevoir tout types d'applications. Le chapitre suivant présente quelques uns de ces modèles de programmation.

Chapitre 4

Environnements de développement

Le chapitre précédent a présenté les modèles de programmation par composition. Ces modèles ont pour objectif de faciliter la conception d'application scientifiques de manière générale. Ils ne sont en effet pas liés à un type d'application particulier.

Les modèles de programmation présentés dans ce chapitre sont moins généralistes. Ils se focalisent sur certains domaines de recherche. Ils permettent ainsi de simplifier et d'optimiser la conception de ces applications en proposant des configurations adaptées. Ils reposent souvent sur un logiciel, le coupleur, qui offre un niveau de programmation de plus haut niveau que les modèles par composition.

Un logiciel de couplage de code permet la construction d'une applications à partir de codes de calcul. Il doit permettre l'exécution et l'inter-connexion de programmes qui n'ont généralement pas été conçu pour cela. Son rôle est de faciliter les échanges de données, le traitement de données intermédiaires, les interpolations de champs, la redistribution parallèle, etc. Le coupleur est en général dédié à un type d'application particulier. Ainsi, il supporte parfois les concepts de maillage, interpolation, intégration, etc. Mais il peut également être plus généraliste.

Cette section présente quelques environnements de couplage de code. Cet ensemble est représentatif de la diversité des coupleurs existant. Certains sont plus spécialisés. Ainsi, les deux premiers présentés (ESMF et Cactus) sont plus spécialisés que les deux suivants (Palm et SALOMÉ). L'objectif de ce chapitre est d'étudier ces environnements dans le contexte des modèles de composition afin de cerner les concepts nécessaires au support des applications scientifiques.

4.1 ESMF

Earth System Modeling Framework (ESMF) est un projet sous licence GPL développé le NASA's Earth Science Technology Office/ Computational Technology. ESMF [9] est un environnement de programmation destiné au développement d'applications de simulation climatologiques ou météorologiques par couplage de code. Pour cela, il offre une structure flexible permettant le couplage de différents modèles.

L'environnement ESMF [96] repose deux types de composant : les composants grille (`Gridded Components`) et les composants de couplage (`Coupler Components`). Ces deux types de composants sont implémentés par l'utilisateur en fortran. Les `Gridded Components` s'occupent de la partie fonctionnelle du modèle. Ces composants doivent périodiquement transférer des flux aux interfaces des domaines de simulation. Ils peuvent ainsi avoir besoin de redistribuer les données, faire des transformations, des moyennes, des conversions d'unités, etc. Les `Coupler Components` permettent les opérations de transformation et de transfert de données. Ils sont implémentés en utilisant les classes du canevas ESMF. Ces classes comprennent des méthodes pour l'avancement temporel, la redistribution de données, les calculs d'interpolation et d'autre fonctions

communes.

Les interfaces de ces composants sont standardisées. Les données sont toujours échangées entre composants en utilisant une structure appelée `State`. Cette structure peut stocker des champs, des ensembles de champs, des tableaux et d'autres `States`.

Les composants peuvent aussi être imbriqués. Ainsi un `Gridded Components` peut être un couplage de composants. La structure de l'application est donc hiérarchique.

Chaque composant – et donc l'application composée par cette hiérarchie de composants – s'exécute en en trois phases : `initialize`, `run` et `finalize`. Elles correspondent chacune à une méthode que le composant doit implémenter.

Les `Gridded Components` ne stockent aucune information à propos des autres composants avec lesquels ils interagissent. Ce type d'information est passé dans la liste des arguments de ses méthodes `initialize`, `run` et `finalize`. L'information qui est passée dans cette liste peut être une structure `State` d'un autre composant grille ou un pointeur de fonction qui accomplit un calcul ou des communications sur une structure `State`. Ces pointeurs de fonction sont appelés `Transforms` et sont créés par les composants de couplage. Ils sont appelés par les composants de couplage auxquels ils sont passés.

Utiliser ESMF se fait en cinq étapes :

- Préparer les codes, c'est-à-dire décider quels seront les éléments de l'application qui vont devenir des `Gridded Components`. Une fois les `Gridded Components` identifiés ils doivent être découpés en trois étapes : initialisation, run et finalisation. Chacune de ces étapes doit être implémentée par une sous-routine.
- Adapter les structures de données, c'est-à-dire englober les structures de données du code dans celle offertes par ESMF pour se conformer aux interfaces ESMF.
- Attacher les méthodes d'initialisation, run et finalisation aux composants.
- Ordonnancer, synchroniser et envoyer les données entre composants : Écrire des coupleurs en utilisant la redistribution, multiplication de matrices creuses, remaillage, transformations d'ESMF.
- Exécuter l'application : Lancer les composants avec le driver ESMF.

ESMF est basé sur le modèle de composant Cca présenté dans le chapitre précédent. Il simplifie ainsi la programmation par composants, mais ne s'applique qu'aux applications dont la forme se découpe en trois parties (initialisation, run et finalisation).

4.2 CACTUS

Cactus [5, 59] est un environnement de conception pour ingénieurs et scientifiques développé sous licence *GPL* par l'*Albert Einstein Institute* [1] à Potsdam et le *Center for Computation & Technology* [6] de l'université de Louisiane. Cactus est destiné à diverses architectures (Stations de travail, grappes, super-ordinateurs).

Cactus tire son nom de son architecture basée sur un coeur central (la chair) qui connecte les modules d'applications (les épines) grâce à une interface extensible. La chair ne fournit que quelques fonctionnalités basiques et les épines fournissent des fonctionnalités plus spécifiques comme la supervision de certaines parties des simulations, les E/S, des modèles spécifiques et des solveurs numériques. Les épines sont connectées au canevas Cactus, elles sont interchangeables et implémentent une fonctionnalité particulière. Plusieurs épines sont à disposition, certaines fournissent ainsi la possibilité de superviser la simulation par l'intermédiaire d'une interface Web.

Les épines communiquent avec le canevas en utilisant l'Api de la chair et plus rarement les Api des autres épines. Elles sont écrites en C, C++ ou fortran. Pour supporter ces différents langages et être portable des types génériques sont utilisés pour les réels et les entiers.

Les épines sont décrites dans trois fichiers écrits en Cactus Configuration Language (CCL). Ces fichiers décrivent les paramètres d'une épine, la façon dont cette épine est ordonnancée et l'interface qu'elle expose à la chair et aux autres épines.

Le fichier `interface.ccl` décrit l'interface d'une épine, c'est-à-dire, les structures de données et les sous-routines qu'une épine met à disposition des autres épines.

Le fichier `schedule.ccl` permet de définir l'ordre d'exécution des sous-routines d'une épine. C'est dans ce fichier que sont spécifiés quand les sous-routines doivent être exécutées par la chair. Chaque sous-routine est associée à un moment particulier de la boucle générale d'itération de l'application. Ainsi, Cactus propose différentes boites dans lesquelles placer les sous-routines à exécuter. Chaque épine doit donc être découpée en différentes phases (initialisation, évolution, et finalisation par exemple) et ses sous-routines placées dans les boites correspondantes.

Enfin, le fichier `param.ccl` contient la description des paramètres d'entrée d'une épine. Ce fichier liste les noms des paramètres, leurs types, la fourchette des valeurs permises et leurs significations. Lors de l'exécution, les informations de ce fichier sont comparées aux valeurs d'entrée et leurs valeurs vérifiées. Les paramètres dans ce fichier peuvent aussi être rendus modifiables. Ainsi, ils peuvent être ajustés durant l'exécution.

Cactus est basé sur une seule boucle d'itération temporelle (définie dans le fichier `schedule.ccl`). L'application construite doit donc correspondre à cette forme.

Le modèle de programmation proposé par Cactus repose sur le principe de composition spatiale qui est présenté dans le chapitre précédent. La composition temporelle est présente, mais limitée. Il est en effet possible de jouer sur l'ordre d'exécution des épines en les plaçant différemment dans la boucle d'itération temporelle, mais le modèle de programmation de Cactus ne propose pas de langage de *workflow*.

4.3 PALM

Le *Projet d'Assimilation par Logiciel Multi-Méthodes* [15] est un outil développé sous licence libre au CERFACS. Il a été initialement conçu pour intégrer des modèles couplés océan-atmosphère.

Ce coupleur est moins spécialisé que les modèles ESMF ou Cactus présentés précédemment. En effet, les composants de ce modèle de programmation peuvent être couplés par l'intermédiaire d'un algorithme plus complexe. Les composants peuvent être exécutés en séquences qui sont à leur tour placées dans des branches pouvant s'exécuter en parallèle.

Une application Palm est donc un ensemble de branches contenant des composants (appelés unités fonctionnelles). L'échange de données entre les composants se fait grâce aux primitives `PALM_Put` et `PALM_Get`. L'échange d'informations entre les composants tient compte de la volatilité éventuelle de ceux-ci. Ainsi, un programme ayant terminé son exécution, les données partagées par la primitive `PALM_Put` demeurent accessibles aux autres composants.

Les composants de Palm ne reposent pas sur le mécanisme `uses`/`provides` présentée dans le chapitre précédent. Les communications entre composants se font par l'intermédiaire du coupleur Palm, il n'y a pas de connexions directes entre composants. Le modèle de programmation proposé par Palm est basé sur la composition spatiale et temporelle.

4.4 SALOMÉ

SALOMÉ [16, 90] est une plate-forme logicielle développée par EDF et le CEA, distribuée sous la licence GNU LGPL. Elle a pour objectif de permettre de gérer l'ensemble du processus de simulation numérique. À cette fin, elle propose un environnement de développement et d'étude

d'applications de couplage de codes. Cette plate-forme est principalement développée en C++ et Python.

4.4.1 Architecture Générale

La plate-forme SALOMÉ est constituée d'un ensemble de modules. Le module principal est le KERNEL. Ce module est la base de la plate-forme. Il implémente un modèle de composant, un modèle d'échange de données et propose des services pour le déploiement d'applications.

La plate-forme SALOMÉ propose d'autres modules. Un module IHM permet aux autres modules d'utiliser l'interface graphique de SALOMÉ. Un module appelé YACS permet de définir et superviser l'exécution d'une application. D'autres modules permettent de créer des géométries, de les mailler, de post-traiter les résultats des simulations numériques. Enfin, il existe d'autres modules qui intègrent des codes de calcul qui seront ensuite liés grâce au module YACS.

Cette plate-forme est extensible. C'est-à-dire qu'il est possible de concevoir d'autres modules contenant, par exemple, d'autres algorithmes de maillage, solveurs, post-traitements, etc. Ainsi, il est possible de créer une plate-forme dédié à un domaine de simulation particulier en sélectionnant uniquement les modules nécessaires.

4.4.2 Modèle de programmation

Le modèle de programmation de la plate-forme SALOMÉ est la combinaison d'un langage de *workflow* et d'un modèle de composants. Une application SALOMÉ est un assemblage de composants implémentant des codes numériques. L'exécution des services offerts par ces composants est orchestré par le langage de *workflow* proposé par YACS.

4.4.2.1 Modèle de composants

Le modèle de composants de la plate-forme SALOMÉ est implémenté par le module KERNEL. Un composant SALOMÉ est une boite noire qui interagit avec l'environnement par l'intermédiaire de ports fournissant ou utilisant des interfaces.

Ce modèle de composants est constitué de trois couches.

- La première, l'objet SALOMÉ, est une extension du modèle d'objet CORBA. Ainsi, un objet SALOMÉ est un objet CORBA avec une interface spécifique permettant de gérer son cycle de vie et son intégration dans la plate-forme. Chaque objet SALOMÉ possède une fabrique qui est embarquée dans une librairie dynamique. Cette librairie est chargée dans un *conteneur* qui joue le rôle d'un serveur d'objets SALOMÉ. Lors de l'exécution de l'application le framework utilise l'interface de ces *conteneurs* pour charger, créer et détruire les objets SALOMÉ.
- La seconde couche ajoute la notion de *ports* aux objets SALOMÉ. Un port est une interface CORBA qu'un objet utilise ou fournit (ports `uses` et `provides`). Cette couche est appelée **Dynamic Software Component** (**DSC**). Elle étend le concept d'objet SALOMÉ avec une nouvelle interface divisée en deux parties. La première permet d'ajouter, effacer ou obtenir les ports d'un composant DSC. La seconde permet la connexion des différents ports. L'extension DSC permet à chaque composant d'ajouter et d'enlever dynamiquement des ports `uses` et `provides`.
- La troisième couche introduit la notion de *service*. Un service est une fonctionnalité exposée par un code. C'est donc une méthode d'un objet SALOMÉ. La construction d'une application se fait par le couplage de ces services.

4.4.2.2 Modèle de *workflow*

Le couplage des services offerts par les composants se fait grâce au module YACS. Celui-ci offre un langage permettant de définir un *workflow* (appelé schémas de calcul dans le contexte de YACS). Les tâches de ce schéma de calcul sont appelées noeuds de calcul. Il y a deux types de noeuds de calcul : élémentaire et composé. Un noeud de calcul élémentaire est implémenté par un service d'un composant DSC ou par un script python directement exécuté dans l'environnement YACS. Un noeud de calcul composé donne accès aux structures de contrôle (block, for, while, foreach, etc).

Les noeuds de calcul ont des ports d'entrée et de sortie. C'est en connectant ces ports que le schéma de calcul est construit. Ces ports sont de trois types.

- Les ports de **contrôle** définissent une dépendance temporelle entre les noeuds. Un noeud dont le port de contrôle d'entrée est connecté au port de contrôle sortie d'un autre noeud sera exécuté après la terminaison de ce dernier. Chaque noeud de calcul (élémentaire ou composé) possède ce type de port.
- Les ports **dataflow** sont utilisés pour transmettre des données d'un noeud à un autre. La transmission des données s'effectue à la fin de l'exécution du premier noeud (celui dont le port de sortie est connecté à un port d'entrée du second noeud). Un port dataflow possède un nom, une direction (entrée ou sortie) et un type. Le noeud de calcul étant implémenté par le service d'un composant, les ports dataflow d'un noeud correspondent aux paramètres de la méthode implémentant le service.
- La dernière catégorie de type de port est le port **datastream**. Il permet aux noeuds d'échanger des données pendant leur exécution simultanée. Comme le port dataflow, il a un nom, une direction et un type de donnée. Les ports datastream correspondent aux ports des composants DSC implémentant le service du noeud de calcul. Un port d'entrée d'un noeud de calcul correspond à un port `provides` d'un composant DSC et un port de sortie à un port `uses`.

CALCIUM est un coupleur du CEA permettant le couplage de codes FORTRAN, C ou C++. Une implémentation de ce coupleur est disponible dans SALOMÉ depuis la version 4. Le couplage de codes CALCIUM est donc possible en appelant les primitives CALCIUM depuis les composants SALOMÉ. Pour cela, les codes de l'application initialement prévus pour fonctionner avec CALCIUM sont encapsulés dans des composants SALOMÉ et les services de ces composants sont connectés par des ports *datastream*.

4.4.2.3 Schéma de calcul

Le schéma de calcul peut être conçu grâce à l'interface graphique du module YACS. Ce schéma de calcul est ensuite sauvé au format XML. Cependant, un schéma de calcul peut aussi être directement écrit en XML ou en utilisant l'interface python de YACS. Toute l'API C++ de YACS est également accessible depuis Python. Il est donc possible de créer le schéma de calcul dans un script python ou directement dans une console python lancée dans l'environnement YACS.

Le module YACS permet l'exécution d'un schéma de calcul. Pour cela, il le charge en mémoire et crée un graphe de noeuds correspondant. Ensuite YACS itère sur ce graphe et recherche un noeud prêt à être exécuté. Lorsqu'il en trouve un, il le charge dans un conteneur distant, connecte ses ports et l'initialise. Finalement, il l'exécute dans un nouveau thread du moteur de *workflow*.

4.5 Analyse

Les coupleurs ESMF, Cactus sont très liés à un domaine de simulation particulier. Ils facilitent beaucoup le travail d'intégration d'applications dont l'architecture correspond à un domaine d'application précis mais ils sont difficilement utilisables avec d'autres types d'applications.

Les coupleurs Palm et SALOMÉ apportent plus de liberté mais demandent aussi plus de connaissances de la part des développeurs. Ainsi, un développeur voulant intégrer son application dans SALOMÉ doit être familiarisé avec la notion de composant, alors que Cactus ne lui demande pas cet effort.

Les deux premiers environnements présentés dans ce chapitre sont principalement basés sur la composition spatiale. Ainsi ESMF et Cactus ne permettent pas de définir de vrais *worflow* même s'ils permettent de jouer sur l'ordre d'exécution au sein de leur boucle d'itération.

Palm et SALOMÉ utilisent la composition temporelle. Palm permet de définir différents branches d'exécution et SALOMÉ propose un langage de *workflow* avec son module YACS.

4.6 Conclusion

Cette section a présenté différents environnements de couplage de code. Contrairement aux modèles de composition présentés dans le chapitre précédent (qui se veulent généralistes), les environnements de couplages sont conçus pour des domaines de simulations plus ou moins spécifiques. Néanmoins, certains de ces environnements tendent à devenir plus généralistes. La plate-forme SALOMÉ, dédiée au calcul scientifique, repose sur deux modèles de composition (un spatial et l'autre temporel). Elle est ainsi plus généraliste que les modèles ESMF ou Cactus.

Cependant, ces modèles présentent tous certaines limitations. L'utilisation de ces modèles de programmation pour la conception d'applications de décomposition de domaine n'est pas encore aisée. Ce type d'application possède une structure constituée d'un ensemble de composants identiques répliqués suivant un schéma précis dépendant de la configuration du problème simulé. Les applications de raffinement de maillage adaptatif ajoutent à cela un aspect dynamique. En effet, la structure de l'application (le nombre de composant, leur placement) change au cours de l'exécution. Ces applications hiérarchiques, dynamiques, dont la structure est la répétition d'un élément, ne sont pas totalement supportées par les modèles de programmation actuels.

La partie qui vient présente une étude des ces différentes limitations.

Deuxième partie

Contribution

Chapitre 5

Applications de décomposition de domaine dans SALOMÉ

5.1 Introduction

Les chapitres précédents ont présenté les limites des modèles de programmation existants. Aucun d'eux ne possède tous les concepts nécessaires à la conception d'application scientifiques complexes. Il n'y a pas, en effet, de modèle de programmation permettant de concevoir une application dont la structure est basée sur la réplication d'un motif (composant et connexion).

Les applications de décomposition de domaine constituent une classe d'application qui pourrait tirer profit de tels concepts. En effet, ce type d'application est un assemblage spatio-temporel de composants dont la structure est constituée d'un type d'interaction répliqué. Cette structure est, de plus, paramétrable.

Ce chapitre présente l'ajout d'une cardinalité aux composants du modèle de programmation de SALOMÉ et son application à la conception d'application de décomposition de domaine.

5.2 Motivation

La finalité de cette étude est d'identifier le ou les concepts permettant le support de la conception d'applications de décomposition de domaine par les modèles de composants.

5.2.1 Décomposition de domaine

La décomposition de domaine est une méthode utilisée au sein de EDF R&D pour accélérer l'exécution de simulations numériques.

Introduction Les applications de simulations numériques permettent de modéliser un phénomène physique sur un domaine donné. Le modèle physique est dans notre cas un système d'équations aux dérivées partielles appliquées sur un domaine de simulation à deux dimensions (Figure 5.1).

Ces équations n'ont pas toujours une solution analytique. Il est néanmoins possible de calculer une approximation de la solution sur le domaine. Pour cela, le domaine de la figure 5.1 est discrétisé. C'est-à-dire que le domaine de calcul est découpé en éléments discrets (Figure 5.2). Sur chaque élément, la solution du problème est approximée par une fonction beaucoup plus simple que la solution analytique.

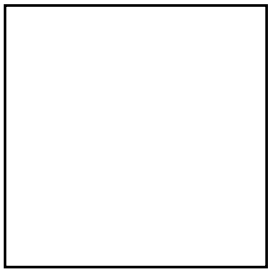


FIGURE 5.1 – Domaine de simulation numérique à deux dimensions

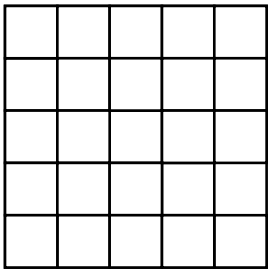


FIGURE 5.2 – Domaine de simulation numérique discrétisé

La précision de la solution dépend du nombre de points de calcul du domaine de simulation. La diminution de l'écart entre les points, et donc l'augmentation de leur nombre permet d'accroître la précision du calcul. L'augmentation de la précision implique donc une augmentation du nombre de calculs à effectuer pour simuler les domaines. Ainsi, la simulation d'un domaine de calcul de grande taille avec un niveau de précision élevé peut nécessiter un temps trop long ou dépasser les capacités de calcul disponibles.

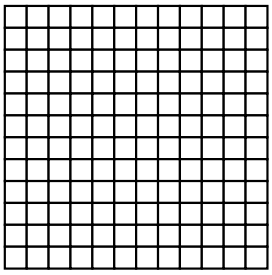


FIGURE 5.3 – Domaine de simulation discrétisé plus finement

La décomposition de domaine apporte une solution à ce problème. Cette méthode propose de découper le domaine de simulation (figure 5.3) discrétisé à un degré de finesse donné en plusieurs sous-domaines.

La figure 5.4 présente les sous-domaines issus du découpage du domaine initial (figure 5.3).

Chacun de ces sous-domaines issus de ce découpage est simulé par une instance différente de l'application de simulation numérique. Le découpage du domaine de simulation permet donc de lancer parallèlement plusieurs instances de l'application de simulation numérique. Cette méthode a deux avantages : diminuer le temps de calcul et permettre de simuler de plus grands domaines. En effet, dans certains cas la taille du système d'équations à résoudre évolue quadratiquement par rapport à la taille du domaine. Il est donc plus rapide de simuler plusieurs petits domaines que l'équivalent en un seul bloc. La méthode de décomposition de domaine permet aussi d'envisager d'utiliser plusieurs machines pour simuler un domaine plus vaste. C'est à dire

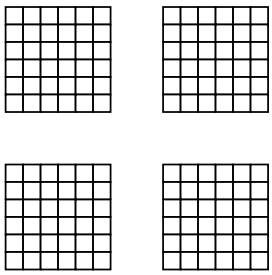


FIGURE 5.4 – Domaine de simulation découpé en plusieurs parties

un domaine pour lequel le temps de simulation et/ou la taille de la mémoire est un facteur limitant.

Les instances de l'application s'exécutant en parallèle doivent périodiquement échanger les données situées sur les frontières de leur domaine de simulation. Il existe différentes méthodes possibles de décomposition de domaine. La méthode utilisée dans ce chapitre est la méthode FETI.

5.2.2 FETI

La méthode de décomposition de domaine utilisée dans ce chapitre est appelé Finite Element Tearing and Interconnect (FETI). Elle a été introduite par Farhat et Roux[64]. Cette méthode propose d'assurer la continuité entre les sous-domaines simulés grâce à un algorithme itératif (dans notre cas le gradient conjugué).

Algorithm 1 Algorithme de l'application FETI

while $T \neq$Tmax **do**
  Simulation des domaines
  Résolution du problème aux interfaces
end while

Il existe plusieurs méthodes FETI mais la complexité mathématique de ces méthodes dépasse le cadre de cet ouvrage. L'algorithme 1 présente la méthode de FETI utilisée dans cette étude. Elle est simplifiée et présentée du point de vue des applications sollicitées. Ainsi, cet algorithme est constitué de la répétition de deux phases. La première étape est la simulation de chaque sous-domaine par une instance différente du code de calcul. Après cette étape de simulation indépendante, les frontières entre sous-domaines voisins doivent être échangées. C'est le rôle de la seconde étape. Celle-ci récupère les valeurs des frontières de chaque sous-domaine et utilise un algorithme itératif pour en assurer la continuité. Une fois calculées, ces valeurs frontières sont renvoyées dans chaque sous-domaine pour la prochaine itération. L'enchaînement de ces deux phases de l'algorithme se répète jusqu'au pas de temps T_{max}.

5.2.3 Application motivante : simulation de phénomènes thermo-mécaniques

Code_ASTER L'application qui a motivé l'utilisation de cette méthode de décomposition de domaine est CODE_ASTER [7]. C'est une application de simulation de phénomènes thermo-mécaniques développée par EDF [8]. Elle est utilisée pour la simulation numérique de chaque sous-domaine lors de la première étape de l'algorithme présenté en figure 1. L'ensemble de son code source est constitué de 1,500,000 lignes de FORTRAN, Python et MPI.

5.2.3.1 Implémentation MPI

Une première implémentation, basée sur MPI, de cette méthode de décomposition de domaine a été conduite dans Code_ASTER. À cause de la taille et la complexité du code, cette tâche a été longue (plusieurs années) et complexe. C'est pour cette raison qu'il a été envisagé une seconde implémentation basée sur la plate-forme SALOMÉ et son superviseur YACS.

5.2.3.2 Implémentation SALOMÉ

Cette implémentation, contrairement à la précédente, est non-intrusive. C'est à dire que l'implémentation de l'algorithme FETI n'a demandé aucune modification de Code_ASTER. Chaque sous-domaine est simulé par une instante différente de Code_ASTER et la résolution du problème aux interfaces s'effectue à l'extérieur de l'application. N'ayant demandé aucune modification de Code_ASTER, l'implémentation a été bien plus aisée (quelques mois). La précision des résultats et les performances de cette implémentation ont été comparées dans une étude [60] à la version basée sur MPI. Les résultats sont strictement identiques.

Architecture de l'implémentation L'implémentation de l'application de Couplage Aster-Aster dans YACS (CAAY) est constituée de deux types de noeuds de calcul. Le premier type de noeud est implémenté par un composant contenant Code_ASTER et est responsable de la simulation numérique d'un domaine. Le second type de noeud est implémenté par le composant *GCPC* (Gradient Conjugué PréConditionné). C'est ce composant GCPC qui implémente la méthode FETI. Ce dernier itère sur les sous-domaines en lançant les simulations et en récupérant les résultats aux frontières des sous-domaines. C'est également lui qui résout le problème aux interfaces.

Ces deux types de composants échangent des données au cours de la simulation. Le composant GCPC récupère régulièrement des données issues des composants Aster et leur fournit de nouvelles valeurs issues de la résolution du problème aux interfaces. Ces échanges de données se font par le moyen de ports *datastream*.

Chaque noeud Aster possède des ports *datastream* (d'entrée et de sortie) pour échanger les données au cours de la simulation. Ces données sont les grandeurs physiques simulées par une instance de Code_ASTER. Le nombre de données est lié à la physique de la modélisation. Il est indépendant de la décomposition de domaine, il ne varie donc pas d'une exécution à l'autre. Le noeud GCPC possède un nombre correspondant de ports *datastream* (entrée et sortie) pour chaque valeur à échanger avec chaque noeud Aster.

La figure 5.5 présente un exemple de schéma de calcul YACS de cette application FETI. Dans ce cas, il y a 3 sous-domaines, donc trois noeuds Aster : *node_aster_1*, *node_aster_2* et *node_aster_3*.

Chaque noeud Aster de cet exemple possède 6 ports de sortie et 7 ports d'entrée *datastream* (ports commençant par la lettre E). Chaque noeud Aster est accompagné d'un noeud de configuration *comm* et relié à lui par l'intermédiaire de ports *jdc* et *argv*. Ces composants permettent de passer des paramètres de configuration aux instances de Code_ASTER.

Les noeuds Aster sont connectés au GCPC *node_gcpc*. On remarque sur cette figure que l'implémentation de l'algorithme FETI (dans le noeud GCPC) utilise le nom de ses ports pour différencier les sous-domaines. Ainsi, les ports de *node_gcpc* qui sont connectés au noeud *node_aster_1* ont un nom se terminant en *%1*, au noeud *node_aster_2* en *%2* et au noeud *node_aster_3* en *%3*.

Le nombre de ports du noeud GCPC dépend directement du nombre de sous-domaines.

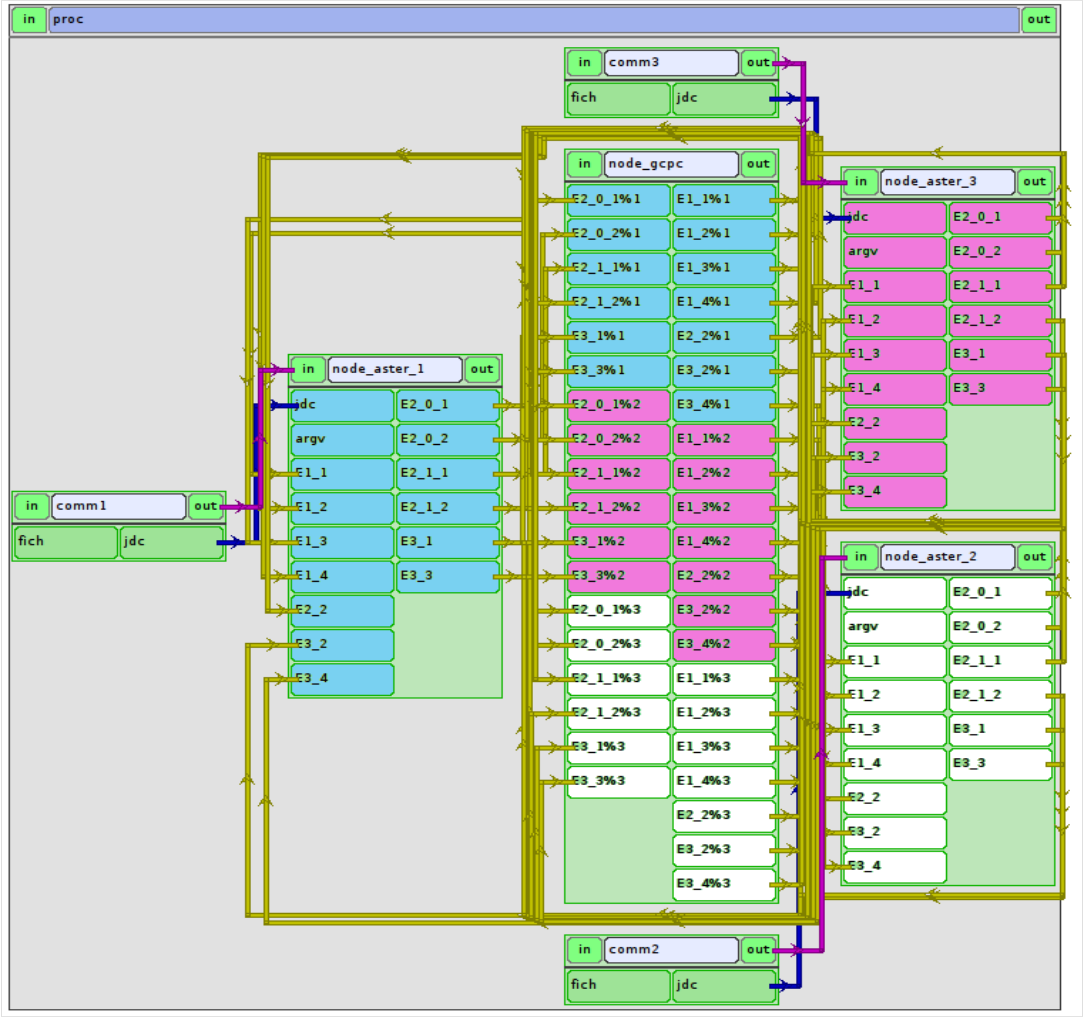


FIGURE 5.5 – Exemple d'application FETI dans YACS avec trois sous-domaines. I/O : input/output *datastream* port.

5.2.4 Limitations du modèle de programmation

La figure 5.6 présente une version simplifiée de l'exemple de la figure 5.5 afin de mettre en avant plus clairement les caractéristiques de cette application.

La structure de cette application dépend d'un seul paramètre : le nombre de sous-domaine. Ce nombre de sous-domaine détermine

- le nombre de noeuds Aster,
- le nombre de ports du noeud GCPC,
- le nom des ports du noeud GCPC.

Le modèle de programmation offert par la plate-forme SALOMÉ ne permet pas d'exprimer ce genre de paramétrisation. Ce modèle de programmation est constitué par la superposition d'un langage de *workflow* (proposé par YACS) et d'un modèle de composants (DSC). Chaque noeud d'un schéma de calcul YACS correspondant à un service d'un composant DSC.

Le langage de *workflow* proposé par YACS ne permet pas de concevoir un schéma dont le nombre de noeuds peut varier d'une exécution à l'autre. De plus, les noeuds d'un schéma de calcul ne peuvent pas non plus faire varier le nombre de leur ports.

Pourtant le modèle de composants DSC ne souffre pas de ces limitations. En effet, le modèle

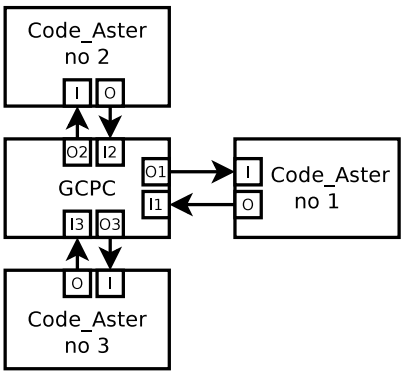


FIGURE 5.6 – Architecture de l'application FETI dans YACS avec trois sous-domaines. I/O : input/output *datastream* port.

de composants DSC permet la création dynamique de composants ainsi que la création de ports à l'exécution.

Ces deux limitations (création de noeuds et ports) du modèle de programmation imposent au développeur de construire une application différente pour chaque configuration de la décomposition de domaine. Pour chaque nombre de sous-domaine et donc chaque nombre de ports, il faut construire une application spécifique.

Adapter l'application à un nombre particulier de sous-domaines impose de changer l'interface du composant GCPC pour adapter le nombre de leurs ports et leurs noms. Ce qui implique donc une recompilation des composants. Le changement de nombre de sous-domaines demande aussi de générer un nouveau schéma de couplage.

5.2.4.1 Automatisation

Reconstruire une application est une tâche fastidieuse. Comme le changement du nombre de sous-domaines est souvent demandé cette tâche a été automatisée. L'implémentation de la méthode FETI avec SALOMÉ comprend donc aussi un ensemble d'outils (scripts shell) facilitant l'adaptation de l'application au nombre de sous-domaines. Cet ensemble de scripts (38 fichiers totalisant 2350 lignes de code) est utilisé pour générer l'ensemble de l'application en fonction du nombre de sous-domaines.

Néanmoins, l'automatisation de la génération d'une application est une tâche complexe. Elle a été effectuée par les utilisateurs de l'application de décomposition de domaine. Cette tâche n'est généralement pas dans leur domaine de compétence et ils n'ont pas beaucoup de temps à y accorder. Cette solution est donc difficilement maintenable et modifiable.

L'ensemble de ces scripts compense les manques du modèle de programmation de SALOMÉ. La suite de ce chapitre présente une extension de ce modèle de programmation permettant les combler et de rendre ainsi ces scripts inutiles.

5.3 Clonage de noeuds et de ports dans YACS

5.3.1 Vue d'ensemble

Le modèle de programmation de SALOMÉ doit être étendu pour permettre de définir un nombre paramétrable d'instances d'un noeud et un nombre variable de ports dans un noeud.

Les deux concepts proposés pour étendre le modèle de programmation de SALOMÉ sont le clonage de noeuds et de ports. Le clonage de noeuds permet d'abstraire la définition d'un

noeud en proposant le concept de collection de noeuds.

Le clonage de ports est une implication du clonage de noeuds. En effet, un noeud d'un schéma de calcul est généralement connecté à d'autres. Lorsque ce noeud est cloné il faut déterminer la façon dont ses connexions sont répliquées. Le clonage de ports apporte une solution à ce problème. Ces deux extensions sont étudiées dans les sous-sections suivantes.

5.3.2 Clonage de noeuds

La première extension proposée vise a attacher à chaque noeud élémentaire une cardinalité. Grâce à celle-ci, un noeud peut exprimer le nombre de clones à instancier à l'exécution. Cette cardinalité est un attribut (une propriété) du noeud de calcul. Cet attribut peut être fixé lors de la création du schéma de calcul ou par un autre noeud.

Cet attribut utilise le concept de propriété. Une propriété est une paire de chaînes de caractères (nom, valeur) qui peut être associée à chaque noeud de calcul. De plus, YACS offre la possibilité de définir des propriété réservées. Ainsi, nous ajoutons la propriété *multi* à la liste des propriétés réservées de YACS. Celle-ci permet de fixer le nombre d'instances d'un noeud à créer.

Cette valeur est utilisée juste avant l'exécution du noeud. Lorsque YACS exécute un schéma de calcul, il parcourt l'arbre des noeuds. S'il trouve un noeud prêt à être exécuté, il vérifie la valeur de la propriété *multi*. En fonction de cette valeur il crée autant de répliques du noeud que nécessaire puis les exécute.

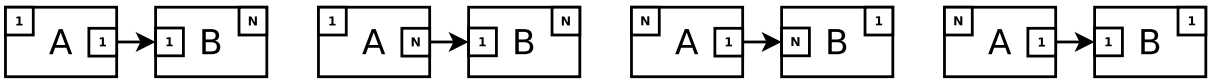
Le clonage de noeuds permet donc la paramétrage dynamique de la réplication d'un noeud. Néanmoins, lorsque ce noeud est cloné, seulement l'implémentation et les attributs de celui-ci sont copiés dans les nouvelles répliques. Il reste à définir la façon dont les noeuds clonés interagissent avec les autres noeuds du schéma de calcul.

5.3.3 Clonage de ports

Lorsque le noeud cloné (noeud de base) est liée à d'autres noeuds dans le schéma de calcul, il est nécessaire de déterminer la forme de l'interaction entre les répliques du noeud de base et les noeuds (noeuds de destination) auxquels le noeud de base était connecté.

Les ports d'un noeud de destination doivent-il être clonés en correspondance avec chaque réplique du noeud de base ou doivent-ils permettre à tous les noeuds de se connecter ? Le choix entre ces deux possibilités se fait par l'utilisation d'une nouvelle propriété qui est associée au noeud de destination. Cette propriété spécifie si les ports doivent être clonés ou non.

Le clonage d'un noeud produit potentiellement quatre configurations d'interactions entre les répliques du noeud de base et le noeud de destination. Ces quatre cas (Figure 5.7) sont présentés par la suite via un exemple. Ils dépendent du sens de la connexion (le noeud de destination est la source ou la destination de la connexion) et du type de la connexion (*dataflow* ou *datastream*). Dans chacun de ces cas, nous assumons qu'A est le noeud source et que B le noeud de destination. La configuration est présentée de manière générique puis sur un exemple comprenant trois instances. Le nom de l'assemblage est constitué de deux groupes de deux caractères reliés par un tiret : `XX-XX` Chaque caractère est soit 1 pour signifier qu'il est simple (non répliqué), soit N pour signifier qu'il est répliqué. Dans un groupe de 2 caractères, le premier se rapporte au noeud, le second aux ports. Le premier groupe correspond au noeud source, le second au noeud destination. Ainsi le premier cas est 11-N1, ce qui signifie que le premier noeud n'est pas cloné et que son port n'est pas répliqué et que le second noeud est cloné et que son port ne l'est pas.



a) 11 - 1N b) N1 - 1N c) 1N - N1 d) 1N - 11

FIGURE 5.7 – Les quatre combinaisons possibles de clonage de noeuds et de ports. Le nom de chaque cas est constitué de deux groupes de deux caractères, le premier pour le noeud A le second pour le noeud B. Le premier caractère correspond au noeud, le second au port. '1'=Non répliqué. 'N'=Répliqué.

Cas 11-N1 Le premier cas (Figure 5.8) présente le cas où le noeud source est simple et le noeud de destination est répliqué. De plus, le port du noeud source doit être connecté à chaque port des répliques du noeud de destination. Ce port n'est donc pas cloné. Le modèle de programmation YACS permet qu'un port *dataflow* de sortie soit connecté à plusieurs ports *dataflow* d'entrée. Cette configuration ne pose donc pas de problème. Le port *dataflow* de sortie du noeud A peut donc être connecté aux ports *dataflow* d'entrée des répliques de B.

De même, les ports *datastream* `uses` peuvent être connectés à plusieurs `provides`. Le noeud A possède donc un ensemble de connexions non nommées. Cette solution est utile dans le cas où le noeud A ne se soucie pas de l'instance du noeud B avec laquelle il communique. C'est la même configuration que pour les ports `multiple uses` dans le modèle CCM.

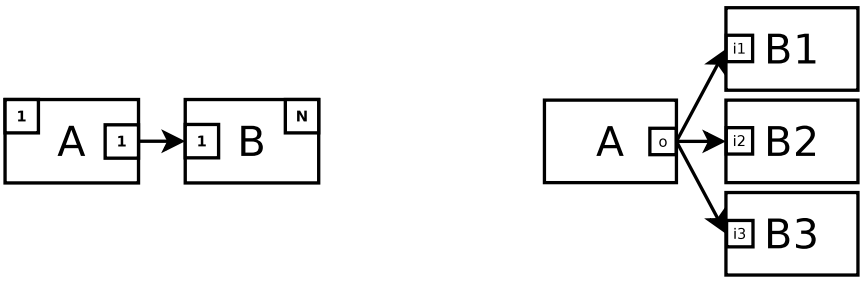


FIGURE 5.8 – Cas 11-1N. À droite la représentation de l'assemblage générique, à gauche un exemple d'assemblage à l'exécution avec trois noeuds clonés. '1'=Simple. 'N'=Répliqué.

Cas N1-1N Dans cette seconde configuration (Figure 5.9), le noeud B est également répliqué et chaque port d'entrée des noeuds répliqués est connecté à un port de sortie différent du noeud A.

Cette configuration n'est pas possible pour les connexions de type *dataflow* car les ports d'entrée et sortie correspondent aux arguments du service d'un composant. L'ajout d'un port *dataflow* d'entrée ou de sortie implique donc une modification du code source et une recompilation du composant. Néanmoins, cette situation est possible dans le cas de connexions *datastream*. Ajouter un nouveau port *datastream* de sortie est permis sans modification du modèle de programmation. En effet, DSC permet la création dynamique de ports et les connexions `MxN`. C'est toutefois à l'implémentation du composant de gérer l'ensemble de ses ports de sortie.

Cas 1N-N1 Le troisième cas (Figure 5.10) présente un noeud A qui est répliqué et chaque port de sortie des noeuds répliqués est connecté à un port distinct du noeud B. Pour les mêmes raisons que pour le second cas, cette solution n'est pas permise par le modèle de programmation pour les connexions de type *dataflow*. Néanmoins, le modèle de programmation l'autorise pour

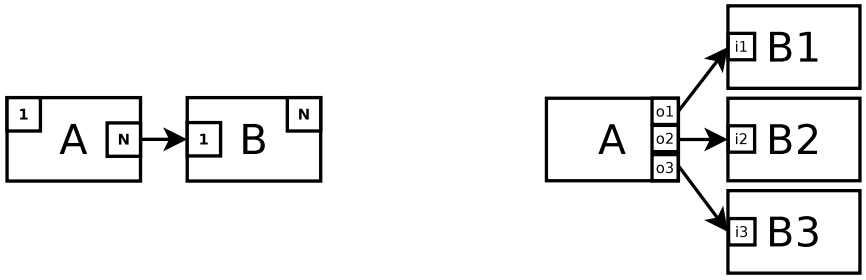


FIGURE 5.9 – Cas N1-1N. À droite la représentation de l'assemblage générique, à gauche un exemple d'assemblage à l'exécution avec trois noeuds clonés. '1'=Simple. 'N'=Répliqué.

les connexions de type *datastream*. Dans ce cas, c'est au noeud B d'être capable de gérer une liste variable de ports d'entrées.

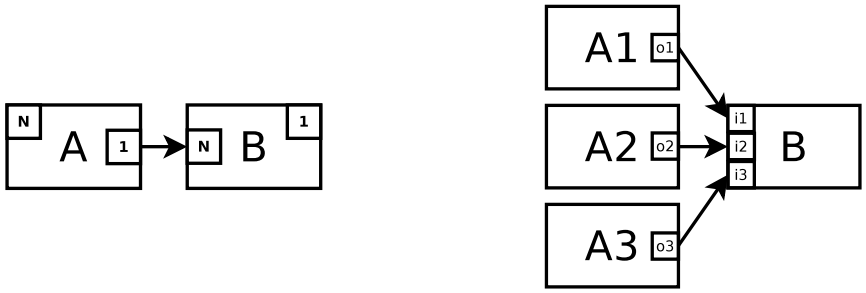


FIGURE 5.10 – Cas 1N-N1. À droite la représentation de l'assemblage générique, à gauche un exemple d'assemblage à l'exécution avec trois noeuds clonés. '1'=Simple. 'N'=Répliqué.

Cas 1N-11 Dans le quatrième cas (Figure 5.11), le noeud A est répliqué et tous les ports de sortie des répliques de A sont connectés au même port d'entrée du noeud B. Cette configuration n'est pas valide pour les connexions de type *dataflow*, car dans ce cas le noeud B recevra plusieurs données alors qu'il n'en attend qu'une seule. Pour que le modèle de programmation puisse supporter cette configuration il faudrait ajouter un composant de réduction utilisant la configuration précédente (1N-N1) entre les répliques du noeud A et le composant B.

Le cas les connexions de type *datastream* est supporté par le modèle de programmation. En effet, un port `provides` peut-être connecté à plusieurs ports `uses`.

5.3.4 Connexions entre noeuds répliqués

Le cas où le noeud de destination et le noeud source sont tous les deux répliqués est un cas connu. Il s'agit des communications parallèles impliquant généralement des redistributions de données. Des propositions ont été faites pour différents modèles de composants tel que CCA [32], CORBA [88] et GCM [34]. Ce cas est déjà implémenté dans SALOMÉ grâce à PaCO++ [89].

5.3.5 Implémentation du clonage de noeuds et de ports

L'API de YACS offre déjà quelques fonctions basiques pour cloner un noeud mais sans tenir compte de ses connexions. Le clonage de noeuds a donc été étendu grâce à l'ajout de la

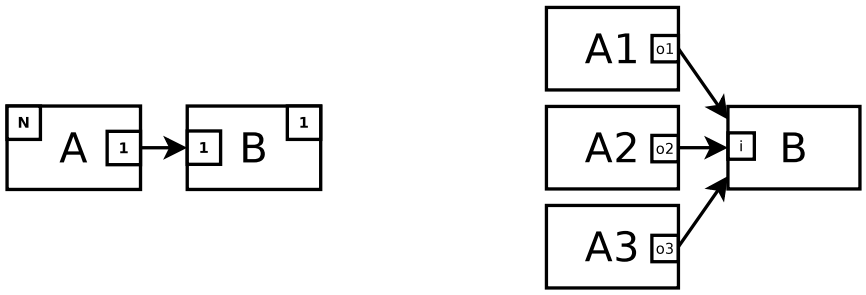


FIGURE 5.11 – Cas 1N-11. À droite la représentation de l'assemblage générique, à gauche un exemple d'assemblage à l'exécution avec trois noeuds clonés. '1'=Simple. 'N'=Répliqué.

propriété *multi* et le clonage de ports ajouté. Plusieurs configurations du clonage de ports sont déjà supportés par le modèle de programmation et son implémentation (comme par exemple le cas 11-1N 5.8), même si certaines conduisent à des configurations non cohérentes (le cas 1N-11 5.11). Les autres cas ont demandé un effort d'implémentation.

5.3.5.1 Ports *dataflow*

Le clonage de ports *dataflow* a été le plus simple à implémenter. Chaque noeud de calcul possède une liste de ports *dataflow* d'entrée et une liste de ports *dataflow* de sortie. La liste des ports *dataflow* d'entrée contient les références des ports *dataflow* de sortie auxquels le noeud est connecté. Un mécanisme, basé sur ces listes, a donc été ajouté pour construire les connexions des nouvelles répliques. Le noeud de calcul nouvellement créé utilise la partie *runtime* de YACS pour connecter les nouveaux ports *dataflow* aux ports *dataflow* de sortie du noeud à partir duquel il a été répliqué.

5.3.5.2 Ports *datastream*

Le clonage des ports *datastream* a demandé plus de travail. En effet, ils ont besoin d'être nommés en fonction du nombre de répliques du noeud de base. Comme pour les ports *dataflow*, en utilisant la liste des ports *datastream* du noeud cloné il est possible d'obtenir le port du noeud auquel il est connecté. Un nouveau port *datastream* dont le nom contient le numéro de réplique est donc créé.

L'implémentation est constituée d'environ 400 lignes de code (C++) insérées dans YACS. Il a fallu modifier deux classes dans la hiérarchie des noeuds, une dans le moteur d'exécution et une seconde dans le *runtime* de YACS. Ces modifications ont eu lieu dans une branche de développement de la plate-forme SALOMÉ et seront disponible pour la prochaine version.

5.3.6 Initialisation des services

Comme les ports *datastream* correspondent aux ports DSC d'un composant implémentant un service proposé par un noeud, ajouter des ports *datastream* à un noeud de calcul implique d'ajouter des ports DSC à un composant. Le modèle de composant de la plate-forme SALOMÉ permet cela et YACS utilise cette fonctionnalité.

En effet, juste avant d'exécuter un service, YACS l'initialise. C'est au cours de cette phase d'initialisation du service que les ports du composant sont créés. La liste des ports à créer est ajoutée à la routine d'initialisation lors de la phase de conception. Ainsi, même si les ports du composant sont effectivement créés à l'exécution du composant, la liste est fixe.

Il a donc été nécessaire de modifier cette phase d'initialisation pour lui permettre de créer les ports du composant correspondants aux ports *datastream* qui sont ajoutés lors de la phase de clonage des ports. Ainsi, le routine d'initialisation crée maintenant les ports en fonction du nombre de port *datastream* et les nomme en conséquence.

5.3.7 Analyse

Le clonage de noeuds et de ports permettent à SALOMÉ de supporter des assemblages plus dynamiques. En effet, les applications moldables (applications dont le degré de parallélisme est fixé au lancement) sont maintenant supportées par SALOMÉ sans besoin de recompilation. Le nombre de noeuds d'une application peut maintenant être fixé par le biais d'une propriété juste avant l'exécution d'un schéma de calcul.

Les applications évolutives, c'est-à-dire avec un nombre variable de noeuds à l'exécution, sont aussi maintenant supportées par la plate-forme SALOMÉ, sans recompilation. Le clonage de noeuds et ports s'effectuant à l'exécution, le nombre de clones d'un noeud peut maintenant être calculé à l'exécution.

5.4 Évaluation

Cette section évalue trois aspects de l'extension du modèle de programmation de SALOMÉ qui vient d'être présenté. Le premier aspect est l'évolution de la complexité de programmation de l'application de décomposition de domaine. Le second est l'impact de cette extension sur les performances. Et enfin, le troisième s'applique aux applications évolutives.

5.4.1 Programmation de l'application de décomposition de domaine

La première évaluation concerne la complexité de programmation. L'extension du modèle de programmation de SALOMÉ simplifie-t-il la programmation d'une application moldable comme l'application CAAY ? Cette section compare également la programmation de l'application CAAY sans et avec l'extension.

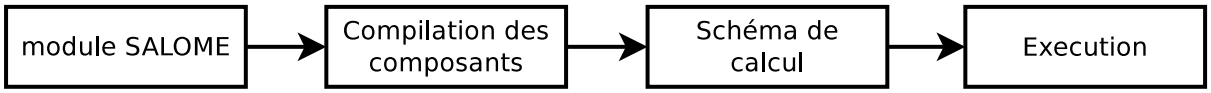


FIGURE 5.12 – Construction de l'application CAAY sans l'extension du modèle de programmation.

Construction native La figure 5.12 présente les différentes étapes du processus de construction de l'application CAAY dans SALOMÉ avant l'ajout de l'extension. Ce processus suit quatre étapes. La première étape est de construire un module SALOMÉ contenant un composant CODE_ASTER et un composant GCPC. Comme expliqué précédemment, un ensemble de scripts (38 fichiers, 2350 lignes de code) prend le nombre de sous-domaines en paramètre et génère les sources du module SALOMÉ pour le composant GCPC. Ce code est ensuite compilé et intégré dans une application SALOMÉ. La seconde étape est l'intégration des composants GCPC et CODE_ASTER dans le module SALOMÉ. Ceci est fait en les encapsulant dans des composants DSC. Cette étape doit être effectuée à chaque fois que le nombre de sous-domaines change. Les codes sources de GCPC et CODE_ASTER sont ensuite compilés et liés avec SALOMÉ. Dans la troisième étape, un script shell génère un schéma de calcul contenant

le nombre de noeuds de calcul et de ports correspondant au nombre de sous-domaines. Dans la quatrième étape, le superviseur YACS exécute le schéma de calcul et récupère les résultats de la simulation numérique. Toutes ces étapes doivent être effectuées chaque fois que le nombre de sous-domaines change.

Construction avec extension L'extension du modèle de programmation permet maintenant de définir l'application indépendamment du nombre de sous-domaines comme présenté sur la figure 5.13.

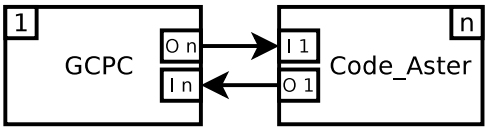


FIGURE 5.13 – Application CAAY Application avec l'extension de clonage de noeuds et ports.

L'application est construite avec le nombre minimal de sous-domaines. Le composant GCPC est créé avec un port d'entrée et un port de sortie. Le schéma de couplage utilise le noeud GCPC et le connecte à un seul noeud CODE_ASTER. Les deux propriétés permettant le clonage de port et de noeuds sont fixées dans les noeuds GCPC et CODE_ASTER. Ces deux propriétés permettent de définir que le composant Aster est répliqué n fois et qu'un nombre proportionnel de ports est créé sur le composant GCPC. Les interactions entre le composant GCPC et les composants CODE_ASTER correspondent donc aux cas 2 et 3 de la figure 5.7.

À l'exécution, l'assemblage est similaire à celui présenté sur la figure 5.6 dans le cas où $n = 3$: le composant GCPC a trois connexions distinctes pour les trois répliques du composant CODE_ASTER. Une seule compilation est désormais nécessaire. Pour l'exécution, le même schéma de calcul est valide pour n'importe quel nombre de sous-domaines.

Comparaison Le clonage de noeuds et ports a augmenté l'expressivité du modèle de programmation de YACS. En effet, cette extension permet d'exprimer que le nombre de noeuds de calcul peut changer durant l'exécution de l'application. Ainsi, les 38 scripts qui étaient utilisés pour adapter l'application au nombre de sous-domaines sont devenus inutiles. Il suffit maintenant de construire le module SALOMÉ puis de construire un schéma de calcul minimal. Ce qui est fait dans deux fichiers (Python et XML) qui totalisent 424 lignes qui peuvent en grande partie être générées par l'interface graphique de SALOMÉ. Le script python permet la construction du module SALOMÉ de l'application. Il définit les types de noeuds utilisés ainsi que leurs ports. Grâce a un utilitaire de la plate-forme (YacsGen) il génère le code source des composants, les compile et les installe dans une application. Le fichier XML est le schéma de calcul minimal (une instance du noeud GCPC, une instance du noeud ASTER). L'adaptation de l'application à un autre nombre de sous-domaines se fait maintenant en changeant uniquement les deux propriétés associées aux deux noeuds GCPC et CODE_ASTER. Il n'est plus nécessaire de passer par une étape de recompilation ni par une étape de génération d'un schéma de calcul.

L'extension du modèle de programmation augmente donc la simplicité de programmation. Le développeur n'a plus à s'occuper lui-même de la réplication des noeuds. Ce processus est maintenant entièrement pris en charge par le modèle de programmation.

5.4.2 Performances à l'exécution

L'algorithme FETI est contenu dans le composant GCPC. C'est en effet le composant GCPC qui lance les simulations des sous-domaines avec les composant CODE_ASTER et qui ensuite

résout le problème aux interfaces. L'algorithme ne débute donc qu'à partir du moment où les composants sont créés. Le processus de réplication intervient durant la phase de création des composants (lors chargement des noeuds de calcul par l'exécuteur de YACS). La réplication n'a donc aucun impact sur les performances de l'application. Les temps d'exécutions sont exactement les mêmes.

Par ailleurs, la transformation de l'application présentée en figure 5.13 en celle présentée en figure 5.6 prend un temps négligeable (très inférieur à 1s).

5.4.3 Complexité de programmation des applications évolutives

Cette section évalue deux aspects dynamiques de l'application CAAY qui correspondent à deux cas d'utilisation.

Premièrement, le nombre de répliques peut ne pas être connu lors du lancement de l'application. En effet, l'extension proposée utilise la propriété qui définit le nombre de répliques à créer juste avant leur exécution. Cette propriété peut donc être fixée à l'exécution.

Pour cela, il a été ajouté un nouveau port d'entré par défaut à tous les noeuds de calcul. Ce port permet de fixer une propriété d'un noeud avec une valeur issue d'un autre noeud. C'est le cas présenté en figure 5.14. Sur cette figure, le noeud B permet de calculer le nombre de sous-domaines à créer en fonction des ressources disponibles.

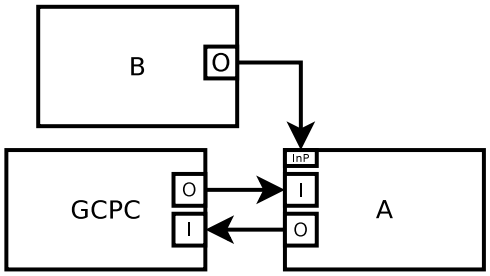


FIGURE 5.14 – Adaptation aux ressources

Deuxièmement, plusieurs exécutions de l'application avec différents nombres de sous-domaines peuvent être nécessaires. Par exemple, lors de la recherche du nombre optimal de sous-domaines. Comme présenté en figure 5.15 le nombre de répliques du composant Aster peut être contrôlé par une boucle externe. En effet, la propriété de réplication peut aussi être liée à la variable d'itération de la boucle. Ce type d'application était auparavant très difficile à développer.

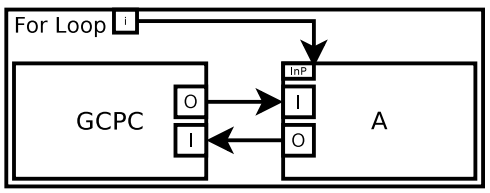


FIGURE 5.15 – Exemple d'utilisation dans une boucle For

5.5 Discussion

Le modèle de programmation de SALOMÉ est composé d'un langage de *workflow* (YACS) au dessus d'un modèle de composition spatiale (DSC). Le modèle de composant DSC est assez proche de CCA [32]. En effet, il permet d'ajouter ou supprimer les ports `provides` ou `uses` lors

de l'exécution d'une l'application. DSC comme CCA ne propose pas de langage de description d'architecture, néanmoins, SALOMÉ diffère de CCA par le fait qu'il complète le modèle de composants avec un langage de workflow.

Contrairement à FRACTAL [50] et GCM [34], SALOMÉ ne supporte pas le concept de composant composite mais propose un modèle de *workflow* qui permet de définir un bloc d'éléments au niveau de ce *workflow*. FRACTAL et GCM n'offrent pas le concept de collection de composant mais offrent le concept de collection d'interfaces. Ce qui permet de définir un nombre arbitraire d'interfaces d'un certain type. Ces interfaces sont créées lors de leur connexions. Le niveau d'abstraction proposé est donc d'un niveau inférieur que le clonage de ports.

Comme le montre [95], certain systèmes de *workflow* permettent de définir des *workflows* abstraits qui sont transformés en *workflows* concrets à l'exécution. YACS est similaire à Taverna [86] ou KEPLER [29] car il propose aussi une manière de définir des *workflows* non-DAG avec des structures itératives. À notre connaissance, il n'existe pas d'autres modèles que YACS offrant le clonage de ports.

Comme SALOMÉ combine un modèle de composant et un modèle de workflow, un modèle de programmation très proche est STCM [46]. Néanmoins, STCM ne possède pas les concepts de clonage de noeuds et de ports. La même analyse s'applique à STKM [26] qui étend STCM avec les squelettes algorithmiques. Bien qu'il soit possible d'étendre STKM avec de squelettes pour implémenter le clonage de noeuds et ports, il semble plus simple de le faire avec HLCM. Néanmoins, HLCM est un modèle de programmation statique.

5.6 Conclusion

L'extension du modèle de programmation de SALOMÉ présentée dans ce chapitre augmente la simplicité de programmation et l'expressivité du modèle. En effet, cette extension permet maintenant de paramétrer le clonage de noeuds et de ports. Cela simplifie fortement la conception d'applications de décomposition de domaine avec SALOMÉ. Elle est maintenant plus rapide et plus facile et sans impact sur les performances. Le clonage de noeud et ports étant maintenant intégré dans le modèle de programmation il est beaucoup plus facile à maintenir ou étendre que l'ensemble des scripts utilisés pour contourner les limitations du modèle de programmation.

De plus, cette abstraction du modèle de programmation ouvre de nouvelles perspectives, notamment pour la conception d'application évolutives au sein de la plate-forme SALOMÉ. Il peut ainsi être envisagé de concevoir une application de raffinement de maillage adaptatif avec la plate-forme SALOMÉ. En effet, dans une telle application, certains noeuds doivent être clonés au cours de l'exécution de l'application en fonction du niveau d'erreur sur le domaine de simulation. Par ailleurs, il est nécessaire de poursuivre les études pour capturer toutes les possibilités et limitations de cette extension dans un cadre d'utilisation dynamique. Cette modification introduit plus d'abstraction dans la conception d'applications. Elle fait apparaître que la conception d'applications est simplifiée en s'éloignant de la représentation des ressources.

Le cas étudié n'est qu'un cas particulier de décomposition de domaine. Dans notre cas la décomposition est de type *quad-tree*, c'est-à-dire d'un facteur 2 dans suivant chaque dimension. Ce facteur de décomposition est différent pour chaque dimension dans certaines applications. Le modèle de programmation de la plate-forme SALOMÉ ne supporte pas la conception ce genre d'applications. Pour cela, il serait nécessaire d'avoir un niveau d'abstraction supplémentaire.

Ce chapitre s'est focalisé sur l'étude de la composition à gros grain. En vue d'une exécution optimale des applications, il est nécessaire de les adapter aux ressources. Notamment dans le cas de ressources hétérogènes. Le chapitre suivant à pour objet la composition à grain plus fin grâce à un modèle de composants de plus bas niveau.

Chapitre 6

Décomposition de domaine et composition bas niveau

6.1 Introduction

L'analyse des différents modèles de composants présentée dans le chapitre 4 montre qu'il n'existe pas de modèle supportant efficacement la décomposition de domaine. En effet, une application de décomposition de domaine peut être amenée à s'exécuter sur des ressources parallèles très hétérogènes comme un super-calculateur, un cluster de noeud multi-coeurs, des GPU, etc. L'implémentation des codes de simulation numériques dépend beaucoup des ressources visées. Les communications et la gestion de la mémoire ne sont pas similaires d'une ressource à l'autre. Ceci implique un effort de développement spécifiquement dédié à l'architecture visée. Cette implémentation et sa validation est un processus long. Les codes validés ont donc tendance à être écrits et validés sur une architecture précise puis adaptés et utilisés pour d'autres. La version adaptée à chaque architecture est alors souvent développée séparément. Le partage de code entre différentes architectures étant de moins en moins important.

Les modèles de composants existants ne permettent pas de concevoir une application pouvant s'adapter à différentes ressources tout en réutilisant des portions de codes générales. L'approche générale consiste a encapsuler le coeur de calcul avec un autre langage comme C++ ou Python. Néanmoins ces deux langages ont montré leurs limitations lorsqu'il s'agit de grands codes, notamment avec la réutilisation de code, la maintenance et l'évolution de ces codes [93]. L'approche basée sur les modèles de composants permet d'identifier clairement les points d'interaction, ce qui pourrait aider à séparer les parties dépendantes des ressources des autres. Les modèles de composants ne sont néanmoins pas très répandus dans le domaine HPC. Ce qui est du au fait que certains modèles de composants introduisent des pertes de performances. De plus les modèles n'offrent généralement pas un niveau d'abstraction suffisant pour adapter les applications à différentes ressources.

Dans ce chapitre, le modèle Low Level Component (L^2C) est présenté. Il a été proposé ainsi que HLCM dans la thèse de Julien Bigot [40]. C'est un modèle proche des abstractions matérielles. Il a pour but d'être très facilement extensible pour supporter les méthodes des interactions "natives". La version courante supporte, sans perte de performance, les interactions comme l'appel de méthode C++, MPI et l'appel de méthode distante (Corba).

Comme L^2C est très proche des abstractions matérielles, les assemblages L^2C ne sont pas destinés à être directement écrit par un développeur mais généré comme High Level Component Model (HLCM) [43, 40]. HLCM est un modèle de composants abstrait qui supporte la hiérarchie, les connecteurs, la généricité et le choix des implémentations de composants et connecteurs. Le modèle HLCM repose sur un mécanisme de transformation qui génère l'application concrète à

partir de la description abstraite. C'est lors de cette phase que les spécificités des ressources peuvent être prise en compte.

L'objet de ce chapitre est d'étudier si le modèle L^2C est assez performant et quels assemblages sont les plus adaptés aux différentes ressources. L'application utilisée dans ce chapitre est une simulation numérique de l'équation de la chaleur utilisant la méthode de Jacobi.

6.2 Motivation

Les applications scientifiques demandent beaucoup de ressources de calcul. Le modèle de programmation utilisé doit donc introduire le plus petit surcoût de performances. L'application doit aussi pouvoir être finement optimisée pour les ressources visées.

Le long cycle de vie de ce genre d'application implique qu'elles devront être exécutées sur une grande variété d'architectures parallèles. Ces ressources vont des supercalculateurs aux clusters à noeuds NUMA mais aussi vers les futures générations de ressources dont l'architecture n'est pas encore connue. De plus, les spécialistes d'un domaine de simulation ne sont généralement pas des experts de machines parallèles.

Un exemple est l'utilisation de la décomposition de domaine. Une application est parallélisée en découpant le domaine de simulation. Chaque sous-domaine est simulé par une unité de calcul différente. La simulation globale itère sur chaque sous-domaine s'occupant de l'échange de données entre sous-domaines voisins.

Néanmoins, certaines variations des ressources matérielles peuvent avoir un impact sur les performances. Comme le nombre de coeurs, leur puissance de calcul, la quantité de mémoire disponible. Les communications entre coeurs ont aussi un impact sur les performances : mémoire partagée, communications par le réseau, affinité des bancs mémoire et des coeurs, la topologie réseau, etc.

Les variations des ressources matérielles impliquent d'adapter l'application pour obtenir de hautes performances. Par exemple, pour une application de décomposition de domaine, le nombre de threads, les sous-domaines qu'ils gèrent et leur placement sur les coeurs doivent être définis.

Les méthodes de communications doivent aussi être choisies. Elles peuvent inclure des partages de mémoire implicites avec synchronisation, des copies en mémoire, du passage de messages sur un réseau haute vitesse ou un WAN. La solution la plus performante peut nécessiter une combinaison de plusieurs méthodes.

Pour supporter ces variations, le modèle de programmation doit faciliter l'adaptation des applications aux variations de ressources matérielles avec le plus grand taux de réutilisation de code entre les différentes versions tout en n'impactant pas les performances et la simplicité de programmation. Ce qui implique que le modèle de programmation doit supporter la séparation des objectifs de développement. D'un côté le développement des codes de simulation de chaque sous-domaine, d'un autre les optimisations pour les différentes variations de ressources matérielles. Ces deux tâches doivent rester indépendantes.

6.3 Modèles de programmation existants

6.3.1 Modèles spécialisés par infrastructures

Un modèle de programmation très utilisé pour la conception d'applications parallèles est l'utilisation d'une librairie de passage de messages telle que MPI. Une telle librairie offre un assez bas niveau d'abstraction au développeur composé d'opérations de communications point-à-point et collectives. Par ailleurs, MPI suppose un modèle de réseau à plat et des machines homogènes.

De là découle de complexes stratégies qui ont été conçues pour adapter les applications aux ressources matérielles [74] et quelques unes dont l'objectif est de laisser les applications s'adapter elles-même aux ressources [78].

Une seconde approche répandue est la programmation par multi-thread. Ce modèle permet d'utiliser efficacement les machines multi-coeurs à mémoire partagée. Les primitives de synchronisation comme POSIX Thread [51] offrent aussi un niveau d'abstraction très proche des ressources. Bien que ce soit un domaine de recherche important il n'y a pas encore de standard pour gérer le placement mémoire. Comme la programmation multi-thread est difficile et sujette à erreurs, de nombreux développeurs utilisent un langage de plus haut niveau tel que OpenMP [56]. Du point de vue du développeur du code de chaque domaine, un programme parallèle OpenMP ressemble à son équivalent séquentiel. Dans le cas d'un code de décomposition de domaine, cela consiste à ajouter quelques directives au code séquentiel pour paralléliser les boucles qui itèrent sur le domaine et ainsi contrôler le niveau de parallélisme.

Comme l'une des plus communes ressource de calcul est composée de machines multi-coeurs interconnectées, les applications basées sur MPI et OpenMP constituent un champ de recherche très actif.

MPI et OpenMP supportent deux modèles de parallélisme complémentaires. Il faut néanmoins noter que les deux modèles demandent des modifications du code de calcul.

Il faut donc, en fonction de la ressource de calcul visée, effectuer un large effort pour adapter les applications à MPI, Pthreads, OpenMP ou un mélange de ceux-ci. La situation se complexifie encore avec l'apparition du GPGPU qui ajoute un autre modèle de programmation (CUDA ou OpenCL ou plus récemment OpenACC). Néanmoins, certains travaux ont pour objectif de gérer de manière transparente les GPU à travers OpenMP [82].

6.3.2 Les modèles s'adaptant aux ressources

Certains modèles de programmation comme Charm++ [76] ou plus généralement les langages Partitioned Global Adress Space (PGAS) comme UPC ou Co-Array fortran [58] ont pour but d'offrir un modèle capable de compiler et d'exécuter les applications sur différentes ressources matérielles. Il manque néanmoins un modèle simple pour la réutilisation de code et la maintenance [93] : ils n'offrent pas de mécanisme pour gérer l'architecture d'une application.

6.3.3 Les modèles de composants logiciels

Les modèles de composants logiciels permettent de gérer l'architecture d'une application [93]. De nombreux codes séquentiels tels que Eclipse (OSGi [14]), FireFox (XPCOM [11]) ou OpenOffice(UNO [3]) sont basés sur les modèles de composants logiciels. De manière similaire, il existe des modèles spécialisés pour le calcul distribué tels que Corba Component Model (Ccm) [87] ou Grid Component Model (Gcm) [34]. Ces modèles supportent la décomposition d'une application en composants indépendants qui interagissent à travers un ensemble d'interfaces clairement définies.

Néanmoins, ces modèles permettent seulement de décrire un assemblage concret de composants, c'est-à-dire un assemblage où chaque composant et connexion sont primitifs. Une autre contrainte est d'être portable. La plupart de ces modèles fixent le type d'interaction supporté : généralement ils ne supportent que l'appel de méthode distante (RMI) avec parfois quelques solutions pour l'hétérogénéité comme OMG IDL pour Ccm. Néanmoins, ceci impacte négativement les performances. Par exemple dans le cas de composants écrits avec le même langage de programmation et s'exécutant sur un même noeud.

Plusieurs modèles ont été proposés pour satisfaire les contraintes des applications HPC. Un modèle connu est le Common Component Architecture (Cca) [39]. Certaines implémentations

comme CCAFEINE ont été conçues pour être utilisées localement (c'est-à-dire sans transparence de réseau) dans l'objectif de minimiser l'impact à l'exécution. Le support des appels inter-langages est fournit par la bibliothèque BABEL qui est utilisée pour les interactions entre composants. Cette bibliothèque introduit néanmoins une baisse des performances lors d'appels entre composants étant dans le même processus. Ceci limite la granularité des composants CCAFEINE. Pour les interactions parallèles, les composants CCA utilisent d'autres modèles comme MPI, mais cela n'apparaît pas dans l'interface des composants. Le support du parallélisme étant le même qu'avec MPI seul, CCA n'aide pas à adapter une application à divers architectures matérielles.

6.3.4 Analyse

Les modèles ou API tels que OPENMP, MPI et leurs combinaisons peuvent très bien convenir à un type de ressource matérielle mais n'offrent pas de solution pour gérer l'adaptabilité des applications aux ressources. Les modèles de composants logiciels offrent une approche intéressante pour permettre aux applications de s'adapter. Néanmoins, soit les modèles de composants logiciels existants impactent trop les performances pour les applications HPC soit ils n'offrent pas un niveau assez élevé d'abstraction pour simplifier l'adaptation des applications aux ressources.

6.4 Low Level Component (L^2C)

On observe qu'une application peut utiliser plusieurs types d'interactions allant de l'appel de méthode locale aux interactions avec transparence de langage et de réseau comme MPI et CORBA en passant par les solutions intermédiaires en terme de performance et de fonctionnalités comme BABEL. Le choix d'un type donné d'interaction pour les communications inter-composants impose pour le concepteur de modèles de composants logiciels de choisir entre performance et portabilité. Ces deux propriétés étant habituellement incompatibles.

Une autre approche consiste à se reposer sur une étape de compilation qui permet de définir un modèle de composants logiciels pour les programmeurs et un modèle de composants logiciels pour l'exécution. Un processus de transformation permet de générer un assemblage "exécutable" à partir d'un assemblage "source". Un exemple d'un modèle de composant permettant de définir un assemblage "source" est High Level Component Model (HLCM) [43, 40].

Une conséquence de cette approche est qu'il faut définir un modèle "exécutable". Comme ce modèle n'est prévu pour être utilisé par un programmeur que pour implémenter les composants primitifs, sa principale propriété doit être de supporter les interactions natives entre composants logiciels. Il doit donc être capable de supporter différentes formes d'interactions. C'est le but du modèle Low Level Components (L^2C) proposé par Julien Bigot [40].

6.4.1 Vue d'ensemble de L^2C

Low Level Component (L^2C) est un modèle qui a été conçu pour supporter plusieurs types d'interactions entre composants logiciels tout en impactant un minimum les performances. Cet objectif est atteint en ayant une partie *runtime* très simple qui propose trois opérations : création/destruction de composants, configuration de composants (qui inclut l'établissement de connexions), et le transfert du contrôle initial à un composant par processus. Dés que le contrôle est passé à un composant, aucun code L^2C n'intervient plus dans l'exécution. Ceci assure que L^2C n'impacte pas les performances après le déploiement.

Un composant logiciel L^2C est décrit en ajoutant quelques méta-données dans le code. Elles permettent au composant logiciel d'être instancié, détruit et donnent des droits en lecture et écriture sur les éléments configurables du composant logiciel. L'accès en lecture permet de

récupérer des informations du composant logiciel, habituellement pour configurer une autre instance. L'accès en écriture permet de configurer le composant logiciel avec des données de l'utilisateur ou avec des éléments venant d'autres composants logiciels dans l'objectif de les connecter.

Par exemple, l'implémentation actuelle de L^2C supporte trois sortes d'interactions : C++, CORBA et MPI. Les appels de méthodes locales en C++ est supporté en copiant un pointeur du composant logiciel fournissant le service au composant logiciel l'utilisant. Un travail en cours applique la même approche aux interactions FORTRAN. Les appels de méthodes inter-processus CORBA sont supportés de manière similaire en copiant les références CORBA. Les communications MPI sont supportées en fournissant un communicateur MPI par l'intermédiaire d'un point de configuration qui représente un groupe de communication. Ajouter le support de nouveaux types d'interactions ne demande pas de changement complexe dans l'implémentation. Il nécessite juste d'ajouter le code pour configurer le nouvel élément.

Une application L^2C peut être construite en *programmant*, c'est-à-dire en utilisant une API pour créer et connecter les composants logiciels. Une alternative plus adéquate pour cette étude est de décrire un assemblage L^2C dans un fichier dédié (XML). Un tel fichier contient une complète description des instances de composants logiciels et des connexions.

6.4.2 Analyse

L^2C est un modèle de composants logiciels très basique. Il ne diffère des autres modèles de composant logiciels que par ces capacités à définir intégrer des interactions entre composants logiciels sans surcoût. Il ne limite donc pas les types d'interactions supportés. Cette propriété était recherchée pour permettre de pouvoir ajouter d'autres types de primitives d'interactions sans perte de performance.

Cependant, il demande un assemblage différent de l'application pour chaque type de machine et/ou d'exécution (ces assemblages pouvant être générés par HLCM par exemple). Il reste néanmoins à déterminer si L^2C est pertinent pour les applications de décomposition de domaine et quels assemblages sont les mieux adaptés en fonction des architectures ciblés.

6.5 Utilisabilité de L^2C avec une application Jacobi

Pour évaluer les avantages de L^2C pour le HPC, nous étudions une implémentation classique d'un problème de différences finies. L'algorithme de base est présenté dans l'algorithme 2. Dans ce cas précis, la fonction f utilisée dans cet algorithme possède un fort degré de localité et sa valeur peut-être calculée en accédant uniquement aux cellules voisines. De plus, seulement les valeurs de l'itération précédente sont utilisées. Il est donc possible de réduire l'utilisation de la mémoire en n'allouant que deux matrices et en les réutilisant.

Pour tirer avantage de ressources parallèles, cet algorithme peut être parallélisé en adoptant la décomposition de domaine. Ceci est possible car il n'y a pas de dépendance de données entre les diverses opérations exécutées dans une seule itération de la boucle externe (ligne 1). Dans ce cas particulier, le code spécifique à chaque domaine est la fonction f, ce qui simplifie le problème.

Pour une machine à mémoire partagée, ceci peut-être fait en utilisant OPENMP. Il s'agit alors d'ajouter une annotation *parallel for* à la boucle de la ligne 2. Un contrôle plus fin peut être obtenu pour mieux s'adapter aux ressources en itérant explicitement sur les sous-domaines et en laissant cette boucle parallélisée comme dans l'algorithme 3

La mémoire peut être gérée de deux façons différentes. Une première approche est d'allouer une matrice pour l'ensemble du domaine dans un seul bloc. Dans ce cas, il est nécessaire de

Algorithm 2 Calcul par différences finies. Le calcul itère sur le temps (boucle ligne 1) et ensuite sur l'espace (boucle ligne 2). La valeur de chaque cellule est calculée en fonction des valeurs à l'itération précédente.

```
1: for i ← 1 to N do
2:   forall pos ∈ DOMAIN do
3:     domain_i[pos] ← f(domain_{i−1}, pos)
4:   end for
5: end for
```

Algorithm 3 Calcul différences finies parallélisé. La boucle parallèle sur les sous-domaines (Ligne 2) distribue la charge de travail sur les coeurs de calcul.

```
1: for i ← 1 to N do
2:   forall subdomain ∈ sub(DOMAIN) do
3:     forall pos ∈ subdomain do
4:       domain_i[pos] ← f(domain_{i−1}, pos)
5:     end for
6:   end for
7: end for
```

mettre en place des mécanismes de synchronisation pour s'assurer que toutes les données ont été calculées avant d'être utilisées (ceci peut être fait avec une barrière dans la boucle externe). Une seconde approche consiste à allouer la matrice du domaine dans plusieurs blocs, un pour chaque fil d'exécution. Dans ce cas, il faut ajouter des zones tampons dans lesquelles les données doivent être échangées à la fin de chaque itération externe pour assurer la consistance des données. Ces échanges permettent aussi de synchroniser les divers flots de contrôle.

Cette seconde approche peut être utilisée pour une machine à mémoire distribuée utilisant MPI mais cela demande de gérer explicitement les communications.

Néanmoins, dans la pratique, paralléliser les applications est un problème. Le code séquentiel est habituellement implémenté par un spécialiste du domaine étudié. Ensuite, plusieurs versions parallèles (OpenMP, MPI, ...) sont dérivées de la version séquentielle par des experts dépendants des ressources matérielles visées. Par ailleurs, le code séquentiel continue d'évoluer, que ce soit pour corriger des bugs ou pour ajouter de nouvelles fonctionnalités. Porter de telles modifications vers les versions parallèles est une tâche complexe à cause des divergences entre les versions du code.

La suite de cette section présente d'abord une version naïve monolithique de l'application, puis une version modulaire.

6.5.1 Une première version de l'application de décomposition de domaine en L²C

L'objectif est d'étudier comment définir des assemblages de composants logiciels destinés à plusieurs sortes de machines - de la machine séquentielle aux machines parallèles - tout en essayant de maximiser la réutilisation de code et la séparation des fonctionnalités. Tous les assemblages et composants présentés dans cette section ont été implémentés et sont évalués en section 6.6.

6.5.1.1 Version de base avec composants logiciels

Six éléments de l'architecture de l'application peuvent être identifiés :

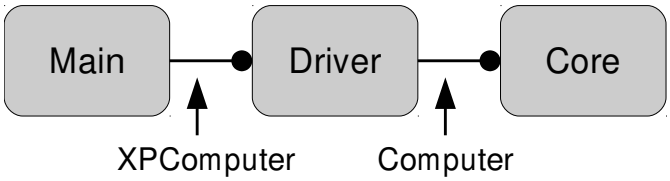


FIGURE 6.1 – Architecture de l'application basée sur trois composants logiciels. Seul le composant `Driver` est spécifique à une stratégie de parallélisation donnée.

Algorithm 4 Algorithme du composant (séquentiel) `Driver`. *core* est le point d'interaction utilisant une interface C++ `Computer` interface.

1: $mem \leftarrow allocate(DOMAIN)$
2: **for** $i \leftarrow 1$ **to** N **do**
3: $\quad core.compute(mem, DOMAIN)$
4: **end for**
5: $release(mem)$

1. L'application principale qui utilise l'algorithme de décomposition de domaine,
2. l'allocation mémoire des matrices,
3. les itérations temporelles,
4. les itérations possibles sur les sous-domaines,
5. les itérations spatiales,
6. le calcul d'une valeur à une position donnée.

Parmi ces éléments, le second et le quatrième dépendent de la stratégie de parallélisation. Une décomposition possible d'une telle application avec L^2C est présentée figure 6.1. Elle isole le code dépendant de la stratégie de parallélisation. Toutes les connexions sont des interfaces C++. Le composant *Core* représente le code de simulation exécuté sur chaque sous-domaine. Il comprend les éléments 5 et 6 – les itérations spatiales et le calcul d'une valeur à une position donnée. Le composant *Driver* encapsule les parties dépendantes des ressources matérielles. Il comprend les éléments 2, 3 et 4. Finalement, le composant *Main* représente le reste de l'application. Dans un cas réel, il serait très probablement composé d'un ensemble d'instances de composants interconnectés.

L'interface entre les composants *Main* et *Driver* est appelé *XPComputer*. Elle permet au composant *Main* de spécifier le domaine et le nombre d'itérations de la simulation et reçoit en retour le résultat du calcul. L'interface entre les composants *Driver* et *Core* est appelée *Computer*. Par son intermédiaire, le composant *Driver* fournit les parties déjà allouées de mémoire qui correspondent au sous-domaine qui est calculé et celui dans l'état précédent comme présenté dans l'algorithme 4 Le composant *Core* calcule les valeurs du sous-domaine courant en fonction des valeurs précédentes.

6.5.1.2 Parallélisation par mémoire partagée

La parallélisation par mémoire partagée de ce code a été implémentée par une version du composant *Driver* appelée *ThreadDriver* qui repose sur la bibliothèque POSIX thread. Ce composant utilise une boucle parallèle sur les sous-domaines qui sont gérés par un fil d'exécution différent. Chaque fil d'exécution itère temporellement et à chaque itération utilise une instance spécifique du composant *Core* pour calculer la prochaine itération. À la fin de chaque itération,

une barrière permet de s'assurer que l'ensemble du domaine a bien été calculé. L'algorithme 5 décrit en pseudo-code le composant *ThreadDriver*. L'architecture complète de l'application pour quatre sous-domaines/fils d'exécutions est présentée en figure 6.2.

Algorithm 5 Algorithme du composant à mémoire partagée parallèle `ThreadDriver`. La boucle parallèle (Ligne 2) utilise un fil d'exécution pour chaque sous-domaine. La synchronisation entre les fils d'exécutions est obtenue grâce à la barrière en ligne 6.

```
1: mem ← allocate(DOMAIN)
2: forall thread ← 1 to Nthreads do
3:    subdomain ← sub(DOMAIN, t)
4:    for i ← 1 to N do
5:       core_t.compute(mem, subdomain)
6:       barrier()
7:    end for
8: end for
9: release(mem)
```

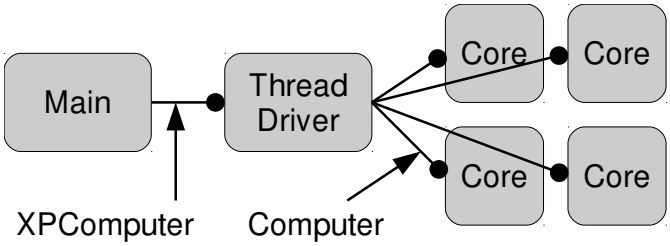


FIGURE 6.2 – Architecture de l'application avec quatre fils d'exécutions en parallèle pour une machine à mémoire partagée. Chaque instance de composant `Core` s'exécute dans un fil d'exécution distinct crée par le composant `ThreadDriver`.

6.5.1.3 Parallélisation par mémoire distribuée

La parallélisation par mémoire distribuée de ce code est implémentée dans une version du composant *Driver* appelée *MpiDriver* qui utilise MPI pour les communications inter-processus. Une instance de ce composant *MpiDriver* s'exécute dans chaque processus avec une instance du composant *Core*. L'algorithme 6 présente le composant *MpiDriver* en pseudo-code. Le composant maître *MpiDriver* qui est appelé par le composant *Main* diffuse l'information qu'il a reçu et laisse chaque instance calculer le sous-domaine qu'il gère. Chaque instance du composant *MpiDriver* itère la dimension temporelle et à chaque itération utilise une instance du composant *Core*. À la fin de chaque itération, les données des zones tampons sont échangées entre les domaines voisins. L'architecture d'une application pour quatre sous-domaines/fils d'exécutions est présentée dans la figure 6.3.

6.5.1.4 Analyse

Cette section a montré comment les composants peuvent être utilisés pour paralléliser par décomposition de domaine. En identifiant le code qui dépend de la parallélisation dans un composant *Driver*, cette approche permet la réutilisation du code de simulation des composants *Main* et *Core* dans plusieurs versions parallèles. Néanmoins, les deux approches (mémoire partagée et mémoire distribuée) ayant été implémentées il est maintenant question de savoir si

Algorithm 6 Algorithme du composant à mémoire distribuée `MpiDriver`. L'allocation de la mémoire est faite localement à chaque processus et inclut les frontières (Ligne 3). La synchronisation des données entre les processus consiste en un MPI *exchange* et est effectuée par la ligne 6 (ce qui correspond en fait à $2D$ groupes de `isend/ireceive` où D est le nombre de dimensions de la simulation.

1: $mpi.broadcast(DOMAIN)$
2: $subdomain \leftarrow sub(DOMAIN, MPI\_RANK)$
3: $mem \leftarrow allocate(subdomain + frontiers)$
4: **for** $i \leftarrow 1$ **to** N **do**
5: $\quad core.compute(mem, subdomain)$
6: $\quad mpi.exchange(mem, frontiers)$
7: **end for**
8: $release(mem)$

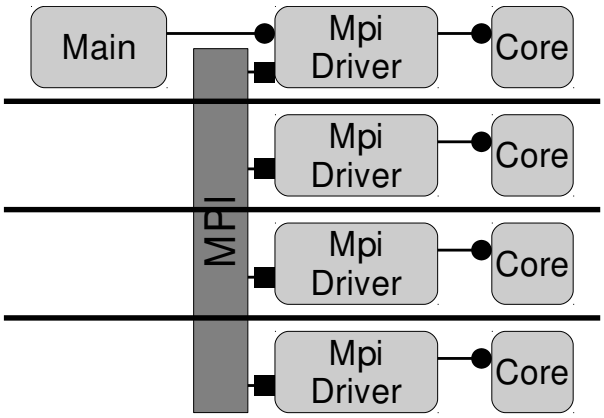


FIGURE 6.3 – Architecture de l'application avec quatre domaines/processus s'exécutant en parallèle avec des espaces mémoire distincts.

on peut combiner les deux approches afin d'utiliser un cluster de machines multi-coeurs. Pour utiliser la mémoire partagée d'un noeud et la mémoire distribuée entre les noeuds d'un cluster multi-coeurs, il faut implémenter une nouvelle variation du composant driver.

Comme ce driver doit combiner les deux approches utilisées dans *ThreadDriver* et *MpiDriver*, il serait intéressant de réutiliser ces codes. Néanmoins, les composants offrant des interfaces différentes ils ne peuvent pas être combinés. La prochaine section analyse et propose une architecture plus modulaire qui rend la combinaison de plusieurs stratégies de parallélisation possible.

6.5.2 Une version modulaire de la décomposition de domaine avec L^{2}C

Pour augmenter la réutilisation de code entre les différentes stratégies de parallélisation, une solution consiste à décomposer les éléments constituants les différents composants *Driver*. D'après ce qui a été présenté dans la section précédente, trois aspects peuvent être identifiés :

- allocation de la mémoire ;
- itération sur la dimension temporelle ;
- gestion de la décomposition, par exemple les interactions des voisins.

La séparation de ces trois aspects en trois composants distincts rend possible la création d'assemblages les combinant de différentes manières tout en augmentant la réutilisation de code comparée à une approche plus monolithique ou chaque combinaison demande le développement d'un nouveau composant *Driver*

Dans la précédente approche, le composant (*Thread*)*Driver* (ou l'ensemble d'instances de composants *MpiDriver* interconnectés) gère les interactions entre tous les sous-domaines. Pour permettre le choix du type d'interaction à un grain plus fin, incluant plusieurs implémentations distinctes dans une seule application, on choisit une approche reposant sur des composants supportant les interactions entre deux sous-domaines chacun. Ces composants offrent une implémentation d'une interface C++ agnostique *Exchange* qui comprend des méthodes pour :

- notifier la disponibilité de données à une itération donnée ;
- spécifier où les données issues d'un voisin sont attendues ;
- attendre les données d'un voisin ;
- demander l'autorisation de réutiliser l'espace mémoire exposé par la première méthode.

Avec cette approche, l'itération sur la dimension temporelle peut être implémentée indépendamment de la parallélisation. C'est le rôle du composant *JacobiCoreNiter*. Il expose un service par l'intermédiaire d'une interface C++ *ComputerNiter* très similaire à l'interface *Computer* décrite en section 6.5.1.1 Grâce à cette interface, l'espace mémoire à utiliser est spécifié ainsi que le nombre d'itérations à exécuter. À chaque itération, *ComputerNiter* fait un appel à l'interface *Computer* pour calculer les données pour la prochaine itération et pour échanger des données avec ses voisins via quatre interfaces *exchange*.

Finalement, l'allocation de mémoire peut être faite après avoir reçu une description du domaine via l'interface *XPComputer* et avant d'appeler *ComputerNiter* pour calculer ce domaine.

6.5.2.1 Parallélisation par mémoire partagée

Pour supporter le parallélisme par mémoire partagée, le composant *ThreadXP* est responsable de l'allocation mémoire. Il crée aussi les fils d'exécutions qui vont gérer cet espace mémoire.

Les interactions entre chaque paires de composants voisins ne nécessitent pas de copie mémoire, uniquement une synchronisation. C'est ce que le composant *ThreadConnector* implémente. Il expose quatre services *Exchange*, un pour chaque composant voisin dans le cas 2D. L'opération "receive" doit juste attendre que la donnée soit disponible. Ceci est implémenté en attendant que les autres voisins appellent l'opération "send" qui spécifie que la donnée est disponible.

Pour implémenter la totalité de l'application avec ces composants, une instance du composant *ThreadXP* est créée. Cette instance exécute en parallèle un ensemble d'instances de composants *JacobiCoreNiter* assemblés en grille avec les composants *Core* dont ils dépendent. Ensuite, les composants *JacobiCoreNiter* voisins sont connectés par l'intermédiaire d'un *ThreadConnector*. Cette architecture pour quatre sous-domaines/fils d'exécutions est présentée à la figure 6.4.

6.5.2.2 Parallélisation par mémoire distribuée

Dans le cas de la parallélisation en mémoire distribuée, plusieurs composants responsables de l'allocation mémoire s'exécutent chacun dans des processus distincts. Le composant *ThreadXP* avec un nombre de fils d'exécutions à créer fixé à 1 pourrait être utilisé. Néanmoins, cela impliquerait une dépendance vers la bibliothèque *pthread* qui pourrait ne pas être disponible. Une version qui ne supporte pas les threads, appelée *XP* à donc été créée.

Dans ce cas, les interactions demandent une copie mémoire à travers les frontières. Ceci ne peut pas être effectué avec une seule instance de composant mais demande deux instances

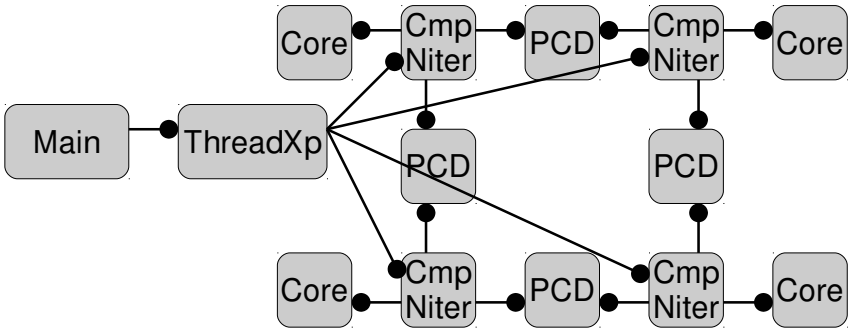


FIGURE 6.4 – Architecture de l'application pour quatre fils d'exécutions sur un espace mémoire partagé. Seuls les composants voisins sont connectés. PCD signifie Posix-`ThreadConnector`.

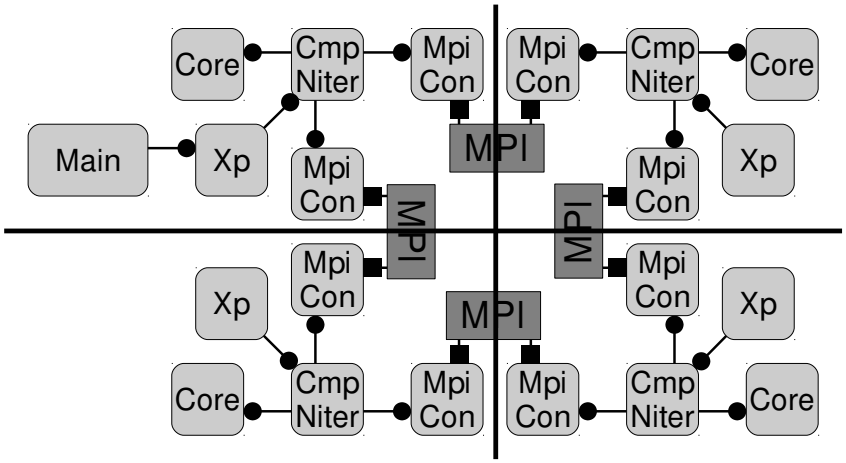


FIGURE 6.5 – Architecture de l'application avec quatre processus s'exécutant dans des espaces mémoires distincts. Seuls les composants voisins sont connectés.

de composant, une dans chaque processus. Le composant *MpiConnector* utilise MPI pour implémenter cette fonctionnalité. Il expose une seule instance du service *Exchange* et utilise un communicateur MPI pour interagir avec le *MpiConnector* du composant voisin. Il fait correspondre les différentes opérations de l'interface *Exchange* sur les méthodes d'échange asynchrones de MPI.

L'architecture de l'application complète basée sur ces composants consiste en une grille de processus contenant chacun les composants *Xp*, *JacobiCoreNiter* et *Core*. Les composants *JacobiCoreNiter* voisins sont connectés par une paire d'instances de composants *MpiConnector*. L'architecture pour quatre domaines/processus est présentée en figure 6.5.

6.5.2.3 Parallélisation hiérarchique

Avec cette approche plus abstraite, il n'est pas nécessaire d'ajouter de nouveaux composants dans le cas d'une infrastructure parallèle à deux niveaux dans laquelle MPI est utilisé entre les noeuds d'un cluster et la mémoire partagée entre les coeurs d'un noeud. Une instance du composant *ThreadXP* est utilisée dans chaque processus pour allouer la mémoire et créer les fils d'exécutions.

L'interaction entre les composants dépend de si les deux composants voisins partagent le même espace mémoire ou non. Si c'est le cas, un composant *ThreadConnector* est utilisé et si ce n'est pas le cas une paire de composants *MpiConnector* est utilisée. L'architecture résultante

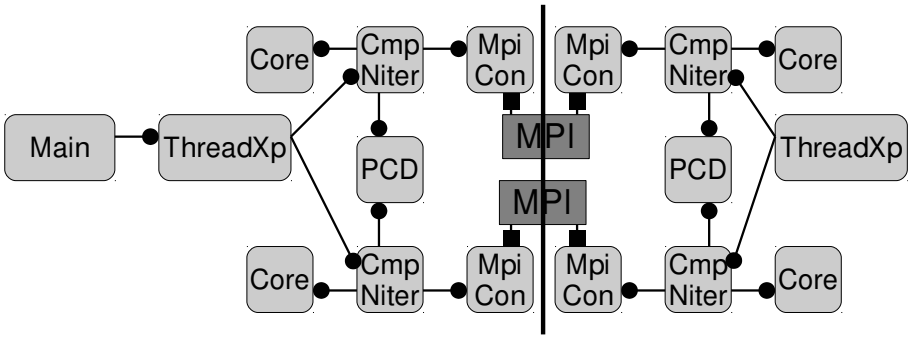


FIGURE 6.6 – Architecture de l'application avec quatre processus s'exécutant en parallèle avec chacun quatre fils d'exécutions.

est présentée à la figure 6.6.

6.5.3 Un seul processus, plusieurs allocations mémoire

Une autre variation qui peut être facilement implémentée est une version avec plusieurs fils d'exécutions dans un seul processus mais qui ne partage pas une même allocation mémoire. Dans ce cas, les interactions entre les voisins impliquent des copies de données. Mais ces copies ne nécessitent pas l'utilisation de MPI. Un composant *CopyConnector* a été implémenté. Il expose la même interface *Exchange* et il est utilisé de manière similaire à *ThreadConnector* excepté le fait qu'il fait un appel à *memcopy* pour copier les données. L'ajout de ce simple nouveau composant rend possible d'implémenter de nombreuses nouvelles variations et combinaisons, comme par exemple une hiérarchie à trois niveaux : plusieurs processus (interconnectés avec MPI) contenant plusieurs allocations mémoires distinctes mais avec plusieurs fils d'exécutions attachés à chaque allocation mémoire. Cette architecture pourrait être pertinente pour les clusters avec grands noeuds NUMA.

6.5.4 Analyse

Cette section a montré qu'introduire des composants logiciels dans une application peut augmenter le taux de réutilisation de code entre variations de l'application pour différentes ressources matérielles. Un premier pas est de simplement encapsuler le code dans des composants logiciels en conservant l'architecture originelle. Cela permet d'identifier les variations induites par les ressources sur les composants, et ainsi de faciliter le remplacement des parties dépendantes du matériel tout en gardant le coeur de calcul. Une analyse plus fine et une adaptation de l'architecture de l'application permet d'identifier les divers aspects variant à grain plus fin tel que l'allocation mémoire et les interactions entre fils d'exécutions. Ceci augmente encore la réutilisation de code et l'adaptabilité à une ressources spécifique en permettant des combinaisons de ces choix à plusieurs niveaux.

Ces variations peuvent être décrites par des assemblages L^2C. Néanmoins, ils deviennent de plus en plus complexes quand la quantité de points de variations augmente comme on peut le voir sur les différentes figures. Chaque variation est décrite par un assemblage distinct dont la variabilité peut être introduite par une spécificité des ressources. C'est pour quoi les assemblages L^2C ne sont pas prévus pour être écrits par un programmeur mais devraient être générés.

Il est possible d'observer une grande similarité entre ces assemblages, autant dans chaque assemblage (par exemple le code s'exécutant dans chaque processus) et entre les assemblages ciblant différentes ressources. Il est donc possible, avec un design intelligent, de grandement

TABLE 6.1 – Architecture des ressources matérielles

Architecture	Version de l'application
mono-noeud, mono-coeur	Version séquentielle
mono-noeud, multi-coeurs	Version multithread
Multi-coeurs, mono-coeur	Version MPI
Multi-noeuds, multi-coeurs	Version hiérarchique (MPI + multithread)

augmenter le taux de réutilisation de code et de faciliter la maintenance. La section suivante évalue quantitativement ces caractéristiques.

6.6 Évaluations expérimentales

Cette section évalue l'approche proposée dans ce chapitre. Elle est basée sur l'implémentation d'une décomposition de domaine de Jacobi. La table 6.1 résume les architectures matérielles utilisées pour les expérimentations : séquentielle, mémoire partagée, mémoire distribuée et architecture hiérarchique. Les expérimentations ont été conduites sur le cluster à noeuds multi-coeurs Griffon de GRID'5000 constitué de 92 noeuds avec 4 coeurs par CPU, 2 CPUs et 16 Go de RAM. Les noeuds sont interconnectés par un réseau Infiniband-20G. Les composants utilisant MPI sont utilisés pour les clusters à noeuds mono-coeurs. Les composants avec plusieurs fils d'exécutions sont utilisés sur les noeuds multi-coeurs. Pour les clusters à noeuds multi-coeurs, une hiérarchie à deux niveaux est utilisée. MPI fait le lien entre les noeuds et la mémoire partagée est utilisée entre les coeurs d'un noeud.

Pour évaluer l'approche proposée, cinq critères sont étudiés : réutilisation de code, accélération, performance, pénalité de performance, et complexité cyclomatique.

Le critère d'évaluation majeure de l'approche étudiée dans ce chapitre est le niveau du taux de réutilisation de code. Un autre aspect très important est l'efficacité, qui est est évalué par trois critères : les performances brutes de l'application, l'accélération de la parallélisation et les pénalités de performance dues à l'approche composant en comparaison avec des versions classiques des codes. Finalement, le dernier critère est la complexité cyclomatique des codes, qui permet d'estimer la complexité des codes.

6.6.1 Réutilisation de code

Comme discuté en section 6.5, la conception d'une application en utilisant un modèle de composants logiciels vise à augmenter le taux de réutilisation de code. Pour quantifier ce taux de réutilisation de code, on compte d'abord le nombre de lignes du code dans sa version non composant. Le tableau 6.2 présente les résultats.

TABLE 6.2 – Nombre de lignes des différentes versions.

Version Jacobi	Nombre de lignes (C++ code)		
	Non composant	Driver	Connecteur
Séquentiel	161	239	388
Multithread	338	386	643
MPI	261	285	446

Le tableau 6.3 présente le nombre de lignes des différentes versions de l'application avec composants. "Driver" correspond à la version basée sur l'encapsulation du code natif dans des composants. "Connecteur" correspond à la version plus modulaire, à base de "connecteurs". Le

TABLE 6.3 – SLOC détaillé pour chaque composant.

Version de l'assemblage	Nom du composant	SLOC	Réutilisé version seq.
Driver & Connecteur	JacobiCore	25	oui
Driver & Connecteur	DataInitializer	68	oui
Driver & Connecteur	Main	105	oui
Driver	SeqDriver	109	
Driver	MpiDriver	155	
Driver	ThreadDriver	256	
Connecteur	XP	71	oui
Connecteur	JacobiCoreNiter	187	oui
Connecteur	ThreadXP	186	
Connecteur	ThreadConnector	140	
Connecteur	MpiConnector	58	

tableau présente le nombre de lignes de chaque composant primitif. Il mentionne aussi quel composant de l'assemblage séquentiel de composants a été réutilisé.

6.6.1.1 Version de driver

Comme présenté dans la section 6.5, la version séquentielle de l'application basée sur des composants logiciels est composée de trois composants : *Main*, *SeqDriver* et *JacobiCore*. Le nombre total de lignes est 239. La version multithread de l'application basée sur des composants est aussi constituée de trois composants : *Main*, *ThreadDriver* et *JacobiCore* pour un total de 405 lignes de code. La version MPI est aussi composée de trois composants : *Main*, *MpiDriver* et *JacobiCore* pour 285 lignes de code. Le nombre total de lignes est du même ordre que pour la version sans composants, mais les composants permettent la réutilisation de code : les composants *Main* et *JacobiCore* sont partagés entre les trois versions, ce qui fait un taux de réutilisation de 26% entre les versions séquentielles et multithread et 32% entre les versions séquentielles et MPI.

6.6.1.2 Version avec connecteurs

Pour la seconde approche, la version séquentielle a besoin des composants *XP* et *JacobiCoreNiter* en plus de *Main* et *JacobiCore*. La version multithread est basée sur les composants *ThreadXP* et *ThreadConnector*. Similairement, la version MPI est basée sur les composants *XP* et *MpiConnector*. La version hiérarchique utilise les composants *XP*, *MpiConnector* et *ThreadConnector*. Le taux de réutilisation de code entre les versions séquentielles et multithread est de 31% et il est de 87% entre la version séquentielle et MPI. La version hiérarchique ne demande aucun nouveau code, juste une nouvelle composition de composants. Le taux de réutilisation de code est donc de 100% par rapport aux versions multithread et MPI.

Les composants augmentent le taux de réutilisation de code. En effet, le composant de base (l'algorithme Jacobi dans le composant *JacobiCore*) est utilisé dans toutes les applications à base de composants. Si des changements sont nécessaires dans l'algorithme Jacobi, seul ce composant doit être modifié. La séparation des rôles imposé par les composants aide à rapidement créer une application.

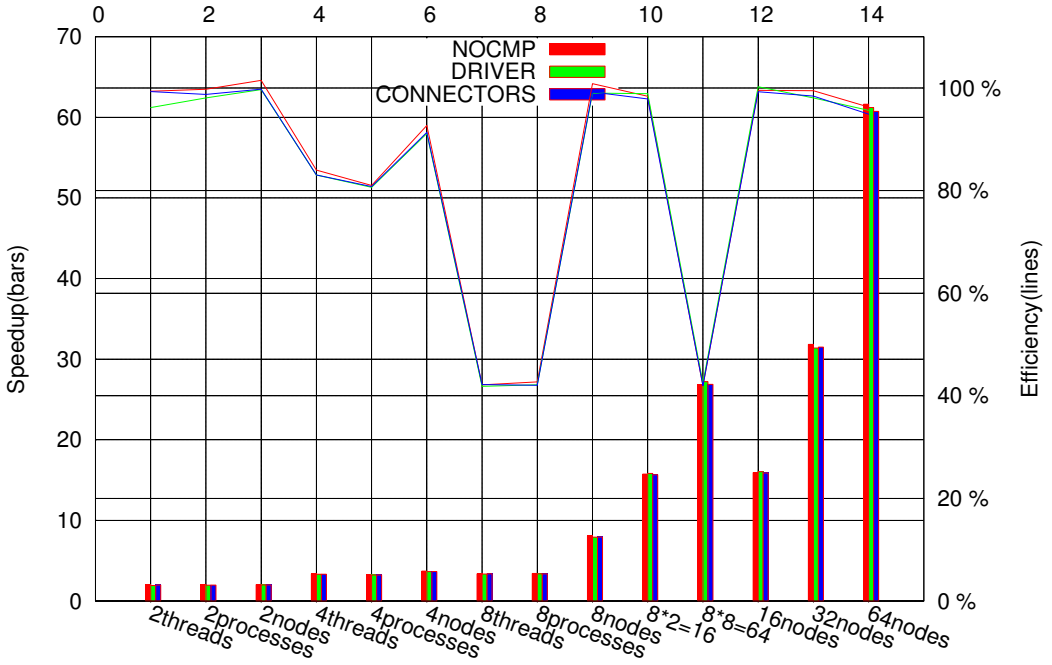


FIGURE 6.7 – Accélération de l'application Jacobi avec passage à l'échelle.

6.6.2 Accélération

Un aspect important du HPC est la faculté d'utiliser les ressources de manière efficiente. Cela signifie que les performances de l'application doivent suivre correctement l'évolution du nombre de coeurs de calcul disponibles. La figure 6.7 présente l'accélération et l'efficacité obtenue pour plusieurs versions de l'application Jacobi correspondants à plusieurs scénarios de déploiement. L'axe horizontal représente différents parallélismes avec plusieurs tailles du tableau de données de l'application Jacobi. Les scénarios thread et processus sont obtenus en utilisant un seul noeud. Pour les scénarios multi-noeuds, un coeur par noeud est utilisé sauf indication : 8x2 signifie que 8 noeuds et 2 coeurs par noeuds sont utilisés. L'axe vertical gauche est l'accélération et l'axe vertical droit est l'efficacité parallèle.

Le principal objectif de cette expérimentation est de montrer que des performances similaires à celles de l'application native peuvent être obtenues avec les versions basées sur les composant L^2C. L'objectif est aussi de montrer que toutes les versions présentent une très bonne efficacité. Cependant, lorsque 8 coeurs par noeud sont utilisés - 8 fils d'exécutions ou 8 processus - l'efficacité chute à cause d'un problème de bande passante de la mémoire. Trop de coeurs accèdent en même temps à la mémoire. La prochaine section détaille plus précisément ces expérimentations.

6.6.3 Pénalité de performance

La figure 6.8 rapporte les résultats obtenus sans composants, avec un driver monolithique et avec une version basée sur les connecteurs dans la configuration des expérimentations précédentes. Néanmoins, elle présente la durée de calcul d'une cellule du tableau en nanosecondes. Le résultat est en accord avec la précédente figure de l'accélération. Il apparaît que la baisse de performance apparaît lorsque plus de deux threads accèdent en même temps à la même mémoire, ce qui conduit à une petite dégradation pour 4 coeurs et à une grande pour 8.

La figure 6.9 présente les résultats des mêmes expérimentations mais normalisés par rapport à la version sans composants. Celle-ci montre que la pénalité de performance de la version

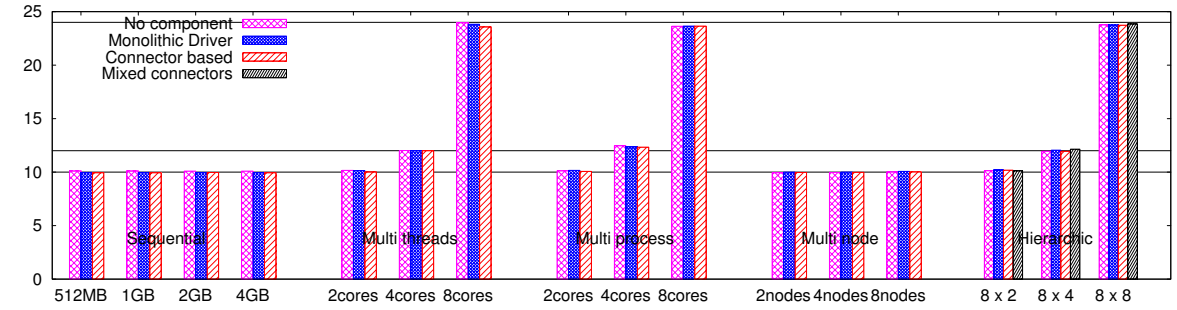


FIGURE 6.8 – Durée en nanosecondes du calcul d'une cellule normalisée relativement au nombre de coeurs utilisés.

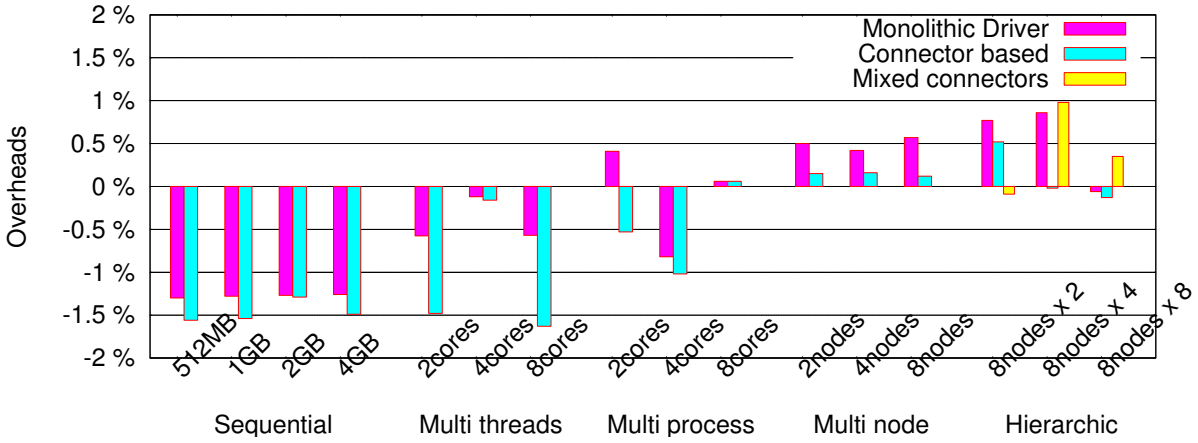


FIGURE 6.9 – Pénalité de performance en pourcentage de l'application basée sur le modèle de composants comparée à l'application native.

avec composants est toujours en dessous de 1%. Dans certains cas, l'application basée sur les composants est même meilleure que l'application native.

De ces expériences, nous pouvons conclure qu'il est possible de convertir l'application Jacobi en version avec composants sans impact sur les performances. Si le nombre de coeurs actifs par noeud est choisi avec attention, le temps de calcul d'une cellule de la matrice peut être le même que dans la version séquentielle.

6.6.4 Performances multi-coeurs

Pour mieux comprendre comment configurer l'application pour maximiser les performances, plusieurs expérimentations ont été conduites avec un nombre différent de threads dans des scénarios mono-noeuds et multi-noeuds.

La figure 6.10 présente les résultats pour un cluster de 8 coeurs par noeuds. 8 threads obtiennent habituellement la meilleure accélération, bien que 4 threads soit un meilleur choix en considérant l'accélération et l'efficacité.

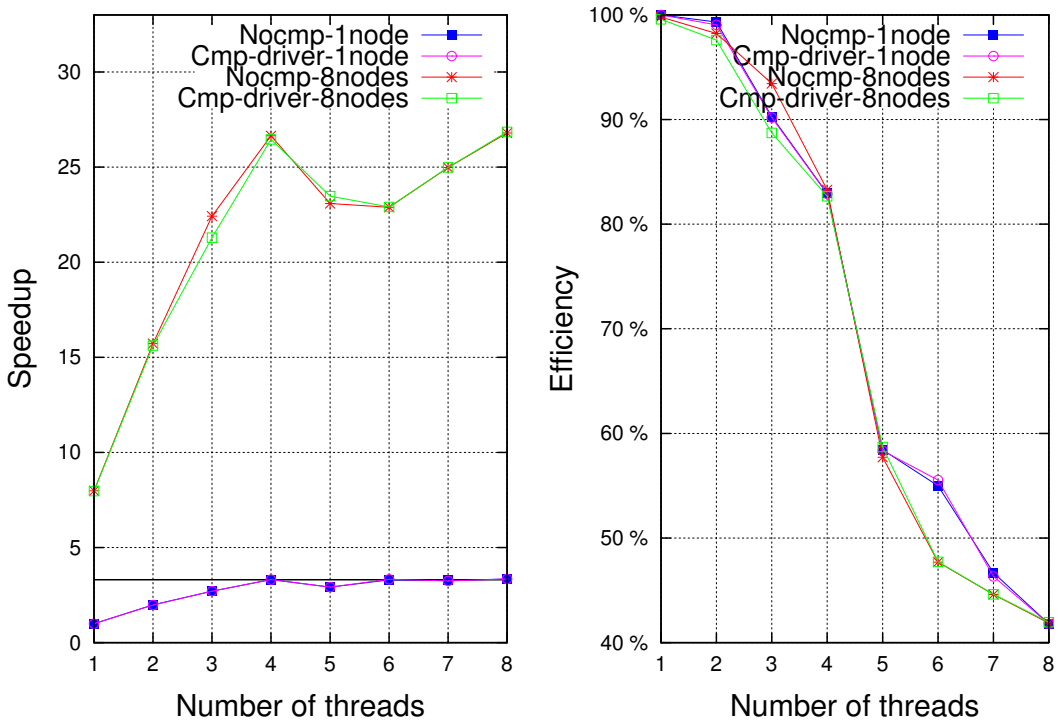


FIGURE 6.10 – Accélération et efficacité de Jacobi pour 1 à 8 threads par noeuds.

TABLE 6.4 – Complexité cyclomatique des codes natifs et des codes à composants

Version	Non Compo.	Driver	Connecteur
Séquentiel	28	32	8
Threaded	76	41	26
MPI	55	22	13

6.6.5 Passage à l'échelle

La figure 6.11 présente les résultats obtenus pour l'implémentation native (threads ou MPI) et pour certains assemblages de composants (les versions avec connecteurs). La taille du tableau est fixée à 22016 x 22016. Dans ces conditions de passage à l'échelle en *strong scaling*, la meilleure configuration est obtenue avec quatre threads par coeur. Les versions composants obtiennent les mêmes performances que les versions natives. Pour les expérimentations à 256 coeurs, chaque coeur gère moins de 16 Mo de données.

6.6.6 Complexité cyclomatique

La complexité cyclomatique est une mesure de la complexité d'un code. C'est le nombre de chemins linéairement indépendants dans le code source. La complexité cyclomatique a été calculée en utilisant `pmccabe`, un paquet standard de calcul de complexité cyclomatique.

Le tableau 6.4 montre la complexité cyclomatique totale des codes avec et sans composants. Les composants réduisent la complexité cyclomatique car ils forcent la séparation des rôles. Seule la version avec composants du driver augmente la complexité cyclomatique par rapport au code natif. Ceci est du à l'intégration de nombreuses fonctions dans un seul gros composant. La version connecteur est bien meilleure.

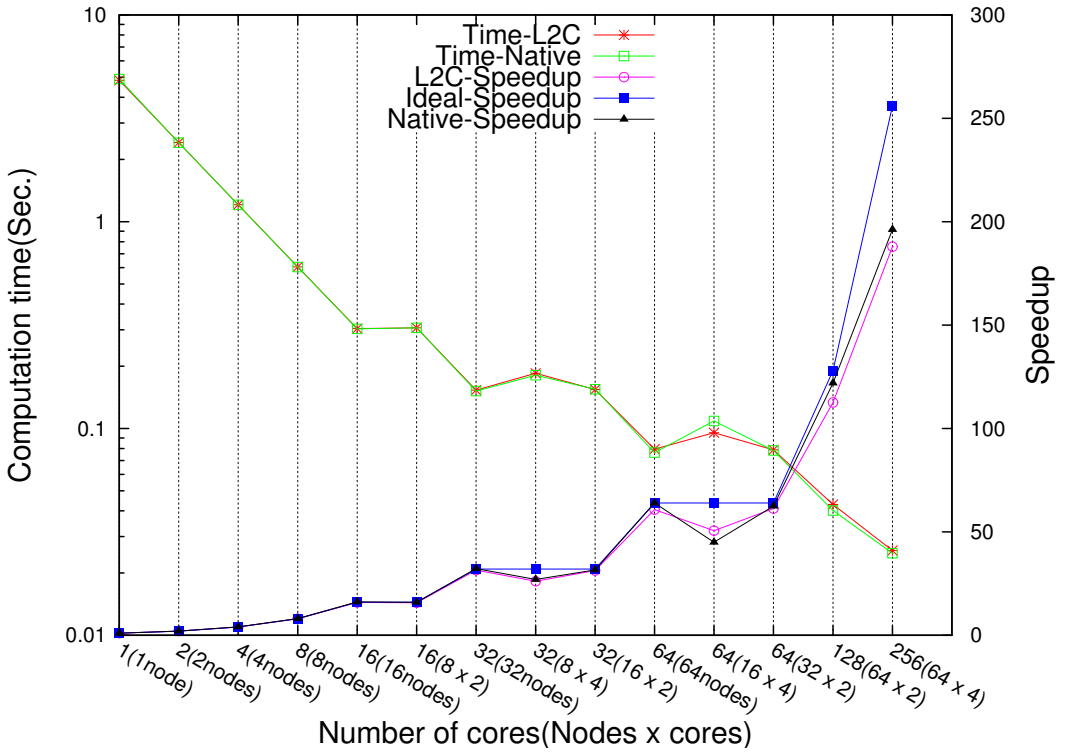


FIGURE 6.11 – Performance du benchmark de l'application Jacobi HPC pour un ensemble sélectionné d'applications natives et à base de composants.

6.6.7 Analyse

Le modèle L^{2}C permet d'augmenter la réutilisation de code, de réduire la complexité du code et d'obtenir les mêmes performances que les versions sans composants. Néanmoins, le modèle de composants introduit de nouvelles tâches. Pour convertir un code en version avec composants il faut créer les composants et décrire leur assemblage pour chaque architecture matérielle. Comme cette dernière tâche est fastidieuse et propice aux erreurs, un tel assemblage devrait être généré automatiquement. C'est l'un des objectif du modèle HLCM [43]

6.7 Conclusion

Pour le développement et l'adaptabilité des applications de calcul haute performance sur différentes architectures, nous avons étudié la pertinence d'utiliser low level compoennt model (L^{2}C) pour gérer la spécialisation en fonction des ressources. L^{2}C supporte différentes sortes d'interactions entre composants avec un impact minimal sur les performances. Les évaluations ont été conduites sur des architectures typiques avec une application benchmark Jacobi. Les résultats expérimentaux montrent que L^{2}C réussi à implémenter la séparation des rôles entre le code de simulation du domaine et les codes spécifiques aux ressources. Il s'adapte aux différentes ressources. De plus il apporte le bénéfice de la réutilisation de code sans dégrader les performances.

Bien que L^{2}C rende possible de décrire l'architecture d'une application parallèle basée sur l'appel de méthode locale, MPI et CORBA, il ne supporte pas l'adaptation aux ressources lui-même. Pour cela il doit être utilisé avec un modèle plus abstrait comme HLCM dont le rôle est de générer les assemblages concrets spécifiques à chaque ressource.

Ce chapitre à présenté une étude à grain plus fin de la conception d'applications de type décomposition de domaine. Le chapitre suivant poursuit l'étude de la conception de ce genre d'applications en passant à un grain plus élevé et en introduisant une composante dynamique dans les applications.

Chapitre 7

Composition et raffinement de maillage adaptatif

7.1 Introduction

L'objectif de ce chapitre est d'étudier la conception d'applications de raffinement de maillage par modèle de composants. Cette étude est menée grâce à l'implémentation dans deux modèles de composants de cette méthode. Un code simple (équation de la chaleur) a été choisi comme code de simulation d'un domaine. La première étude utilise un modèle de composition spatiale et temporelle possédant certaines caractéristiques (relative notamment à la dynamicité) pouvant faciliter la conception d'applications de raffinement de maillage. La seconde étude utilise le modèle de programmation proposé par la plate-forme SALOMÉ.

7.1.1 Méthode de raffinement de maillage adaptatif

La résolution d'équations différentielles partielles implique un domaine discret constitué d'un ensemble de points. Une estimation de la solution est calculée sur chacun de ces points. L'espacement entre ces points détermine l'erreur de la solution et le coût en calcul de la simulation. Une discrétisation plus fine implique donc plus de précision mais plus de calcul. Pour de nombreux problèmes l'erreur locale n'est pas uniformément répartie sur le domaine de simulation. Elle est plus importante à certains endroits. Un moyen d'obtenir une solution précise sur l'intégralité du domaine de simulation est d'augmenter la finesse de discrétisation jusqu'à atteindre une erreur locale minimale en tout point du domaine. Néanmoins, cette solution introduit inutilement de nombreux calculs. En effet, la plus grande partie du domaine de simulation se retrouve maillée plus finement que nécessaire (notamment dans les zones à faible niveau d'erreur).

Le raffinement de maillage adaptatif (AMR), proposé par Berger et Oliger [38], apporte une solution à ce problème. Cette méthode propose de focaliser les calculs uniquement sur les parties du domaine de simulation qui le nécessitent en ne raffinant le maillage que dans ces endroits. Cela permet d'éviter de nombreux calculs inutiles. À cette fin, une estimation de l'erreur est calculée sur le domaine. Elle permet de décider périodiquement ou adaptativement, quelles sont les parties qui demandent plus de précision.

Il existe deux stratégies principales. La décomposition est basée soit sur un schéma de partitionnement géométrique précédemment défini (comme les quad-tree pour un domaine 2D) soit sur un ensemble de patchs calculés à l'exécution [37]. L'étude présentée dans ce chapitre utilise la méthode de raffinement basée sur un partitionnement quad-tree.

L'algorithme de base est simple. Au début de la simulation, l'ensemble du domaine est raffiné à un certain degré de manière homogène. Cette étape est présentée à la figure 7.1. L'ensemble

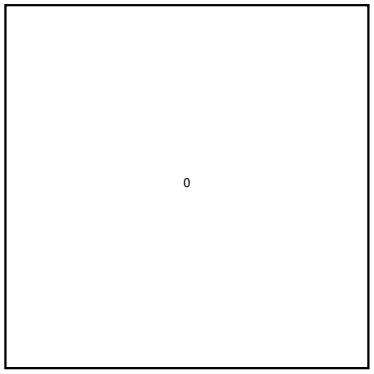


FIGURE 7.1 – Exemple d'exécution de l'algorithme de raffinement de maillage adaptatif. Étape 1.

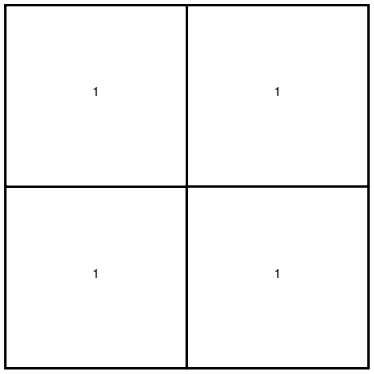


FIGURE 7.2 – Exemple d'exécution de l'algorithme de raffinement de maillage adaptatif. Étape 2.

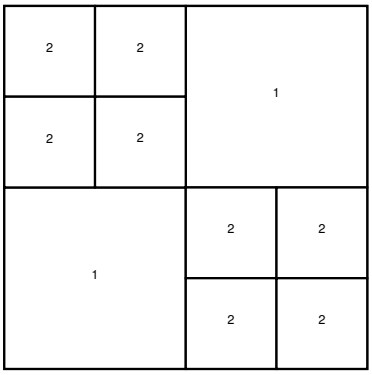


FIGURE 7.3 – Exemple d'exécution de l'algorithme de raffinement de maillage adaptatif. Étape 3.

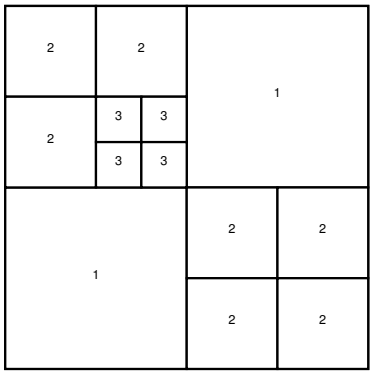


FIGURE 7.4 – Exemple d'exécution de l'algorithme de raffinement de maillage adaptatif. Étape 4.

du domaine est couvert par un seul maillage. Le niveau de raffinement est donc 0.

Au cours de l'évolution de la simulation, une estimation de l'erreur est calculée. Elle permet de choisir les parties qui ont besoin d'être raffinées et donc subdivisées. Dans notre exemple, l'erreur calculée est trop importante, il faut donc raffiner. Le domaine de simulation est donc subdivisé en quatre sous-domaines. Chaque sous-domaine est maillé deux fois plus finement que lors de l'étape précédente.

Un exemple de cette étape est présenté en figure 7.2. Sur cette figure, le domaine est couvert par quatre sous-domaines. Ces sous-domaines sont raffinés à un degré deux fois plus fin, c'est le niveau de raffinement 1.

La poursuite de l'exécution de cet l'algorithme peut produire l'état présenté en figure 7.3 puis l'état présenté en figure 7.4. Dans cette dernière figure, on remarque que certains sous-domaines avec un degré de raffinement 1 lors de l'étape 2 (figure 7.2) ont été raffinés et subdivisés pour obtenir des sous-domaines de niveaux de raffinements 2. Puis, certains sous-domaines de niveau 2 ont à leur tour étés raffinés et subdivisés pour obtenir le niveau de raffinement 3. L'arbre de sous-domaines correspondant à cet exemple est présenté en figure 7.5.

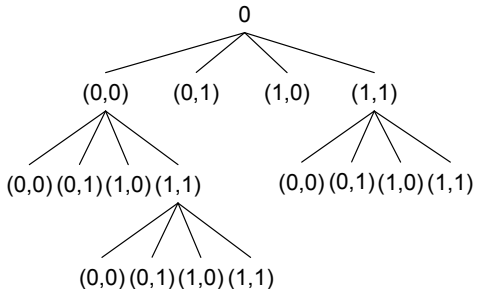


FIGURE 7.5 – L'arbre résultant des sous-domaines correspondant à l'exemple présenté à la figure 7.4.

7.1.2 AMR et composants

Dans l'implémentation de la méthode AMR avec un modèle de composants, le code de simulation numérique d'un domaine est encapsulé dans un composant. Ce composant supporte un maillage ayant un certain degré de finesse.

Lors de l'étape de raffinement (figure 7.2) la finesse globale du maillage est multiplié par deux, dépassant ainsi la contrainte implicite de capacité d'un composant. Le domaine de simulation est donc subdivisé en quatre sous-domaines et chacun de ces sous-domaines est simulé par un nouveau composant. L'implémentation de la méthode AMR avec un modèle de composants implique donc que celui-ci permette la création de composants à la demande.

Par ailleurs, cette implémentation implique de prévoir des mécanismes pour
- l'échange des frontières au cours de la simulation entre les composants simulant des sous-domaines voisins,
- l'initialisation des nouveaux composants avec les données issues des calculs précédents,
- le calcul de l'estimation d'erreur.

Ces mécanismes sont présentés dans les sections suivantes à travers l'implémentation dans le modèle de composants ULCM et la plate-forme SALOMÉ.

7.2 AMR avec Ulcm

Cette section présente la première étude d'implémentation de la méthode de raffinement de maillage adaptatif. L'objectif est d'étudier la faisabilité de la conception de ce type d'application en utilisant un modèle de composant possédant des concepts avancés. Le modèle de composants utilisé est ULCM.

7.2.1 Vue générale d'Ulcm

Unified LEGO Component Model (ULCM) [31] est un modèle de composants proposé pendant l'ANR LEGO. Il est basé sur un modèle de composants hiérarchique *uses/provides*. Il offre des concepts originaux comme le partage de données entre composants, les relations master-worker et un langage de description d'architecture spatio-temporel [46].

Un composant ULCM peut avoir quatre sortes de port :

a) les ports *uses/provides*,

b) les attributs,

c) les ports de partage de donnée et

d) les ports input/output pour la composition temporelle.

La dernière catégorie des quatre sortes de ports expose un type de donnée alors que *uses/-provides* expose une interface orienté objet (comme CCM ou CCA).

Les composants composites du modèle de composants ULCM sont constitués par une combinaison spatiale et temporelle (workflow) de composants (de la même manière que défini dans le modèle STCM [46]). La figure 7.6 présente un exemple de définition d'un composant composite avec ULCM. Dans la partie *decl* le composant composite déclare l'assemblage spatial de composant. Ici il s'agit uniquement d'une instance du composant *Hello*. Ensuite, le service *Execute* est déclaré. Celui-ci permet d'exécuter le service *world* de l'instance *hello*.

```
component Exemple {
  content {
    composite {
      decl {
        Hello hello;
      }
      service Execute {
          exec hello.world;
      }
    }
  }
}
```

FIGURE 7.6 – Exemple simple de composant composite en ULCM avec une instance de composant primitif et un service.

ULCM est un modèle de composant abstrait. En effet, les composants primitifs de ce modèle de composants sont définis indépendamment du modèle de composants réalisant leur implémentation. Ainsi, les composants primitifs peuvent être implémentés avec CCM, CCA, SCA ou une classe JAVA. ULCM ne pose aucune contrainte sur le modèle d'implémentation des composants primitifs.

Ce modèle de composants a été choisi car il possède des caractéristiques essentielles pour faire une première étude de la faisabilité de la conception d'une application AMR avec des modèles de composants. Ainsi, ce modèle permet la composition spatiale et temporelle et donc la création dynamique de composants. De plus, il est abstrait et son implémentation comporte un back-end JAVA, ce qui permet de développer rapidement des composants JAVA pour l'exécution locale.

7.2.2 ULCMi : Une implémentation de Ulcm

ULCMi est une implémentation prototype en JAVA d'ULCM. Celle-ci utilise un moteur de *workflow* centralisé pour gérer les parties dynamiques des composants composites. Cette implémentation supporte actuellement cinq *back-ends* pour la définition des composants primitifs :

- **simulation** qui permet de simuler l'exécution de l'application,
- JAVA, qui permet d'utiliser des objets JAVA comme composants primitifs,
- FRIM (une implémentation C++ de FRACTAL),
- Charm++,
- CCM

Seuls les *back-ends* Charm++ et CCM permettent des exécutions distribuées.

7.2.3 Implémentation de l'AMR dans Ulcm

La méthode AMR basée sur un partitionnement quad-tree a été implémentée dans ULCM en utilisant des composants primitifs JAVA. Cette implémentation comprend un total de trois

composants composites et quatre composants primitifs.

Les quatre composants primitifs sont :

– **Heat**, le code de simulation d'un domaine. Dans notre cas il s'agit de l'équation de la chaleur. Ce composant est présenté en figure 7.7. Il fournit et utilise des interfaces de type *border* correspondants aux quatre frontières du domaine de simulation. L'interface de type *border* permet l'échange de frontières entre composants voisins. Ce composant possède également un port qui utilise une interface de type *gradient* dont l'objectif est de transmettre l'erreur calculée sur le domaine. Enfin, ce composant utilise une interface de type *GData* qui lui permet de s'initialiser ou de transmettre les données issue d'un calcul précédent du domaine.

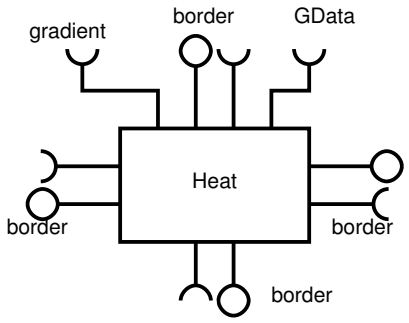


FIGURE 7.7 – Composant primitif **Heat**.

– **Proxy** est un composant dont le rôle est de transmettre les frontières des domaines. Ce composant est présenté en figure 7.8. Il possède trois couples de ports *uses/provides* de l'interface *border*. Ces ports permettent l'échange de frontières entre deux niveaux successifs de raffinements différents. En effet, dans l'exemple de raffinement présenté en figure 7.9 le domaine en bas à gauche de la figure doit échanger ses frontières avec 5 sous-domaines voisins. La figure 7.10 présente l'assemblage correspondant à la partie inférieure de la figure 7.9. À la figure 7.9, le composant simulant le sous-domaine en bas à gauche est *Heat 1*. Il doit échanger ses frontières avec les composants responsables des sous-domaines sur sa frontière droite (*Heat 2* et *Heat3*). Comme le montre la figure 7.10, c'est le composant *proxy* qui permet de faire le lien entre les composants *Heat*. Le passage d'un niveau de raffinement n à un niveau de raffinement $n+1$ implique l'utilisation d'un composant *proxy*. Il faut donc autant de *proxy* qu'il y a d'écarts entre les niveaux de raffinement. Ainsi, entre les niveaux 1 et 3 de la figure 7.9 il y a 2 *proxy*. Par ailleurs, le composant *heat 1* est connecté à deux composants (*heat 2* et *heat3*) par l'intermédiaire du composant *proxy*. Le composant *proxy* doit donc interpoler les frontières qu'il transmet.
– **Average** est un composant qui permet de rassembler les estimations d'erreur issues de chaque composant **Heat**. Ce composant est présenté en figure 7.11. Il possède deux ports qui fournissent et utilisent une interface de type *gradient*. Le port *provides* est utilisé par les composants **Heat** pour transmettre l'estimation d'erreur qu'ils ont calculé sur leur domaine. Le port *uses* est utilisé pour récupérer la moyenne des estimations d'erreur de chaque composant. Chaque composant *Heat* est identifié par ses coordonnées dans l'assemblage correspondant au sous-domaine auquel il participe.
– **Interpol** permet l'interpolation des résultats entre différents niveaux de raffinement. Ce composant est présenté en figure 7.12. Il possède deux ports *uses/provides* de l'interface *GData*. Cette interface permet de récupérer ou envoyer les résultats de simulation sur un sous-domaine. Ce composant est utilisé lors des phases de raffinement pour initialiser les

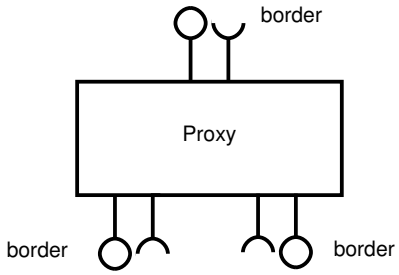


FIGURE 7.8 – Composant primitif **Proxy**.

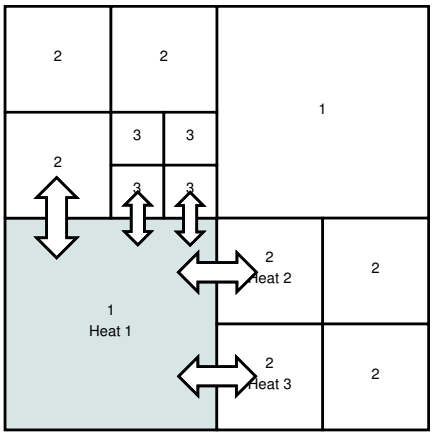


FIGURE 7.9 – Échanges de frontières dans un exemple de configuration AMR

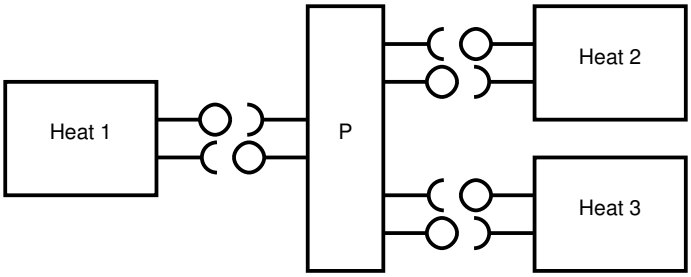


FIGURE 7.10 – Assemblage correspondant à la partie inférieure de l'exemple présenté en figure 7.9.

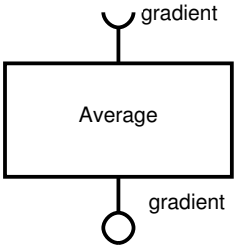


FIGURE 7.11 – Composant primitif **Average**.

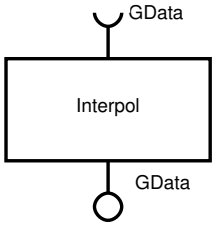


FIGURE 7.12 – Composant primitif **Interpol**.

nouveaux composants avec les valeurs calculées avant le raffinement. De même, dans le cas du déraffinement les valeurs calculées sur un domaine par un ensemble de composants sont utilisées pour initialiser un composant au niveau supérieur.

Les trois composants composites sont :

- **Application**, l'application globale ;
- **HeatM**, le composant principal (capable de se subdiviser) ;
- **HmainM**, un composant auxiliaire contenant quatre instances du composant principal.

Ces trois composants composites sont présentés plus précisément par la suite.

Application L'implémentation avec ULCM de l'application AMR est accessible grâce au composant composite dont le code est présenté en figure 7.13. C'est ce composant qui permet de lancer la simulation. Il est constitué d'une instance du composant principal de l'application **HeatM** (présenté dans le paragraphe suivant).

Ce composite possède un seul service (`run`) qui permet d'effectuer les itérations de l'application. Dans l'exemple d'application présenté en figure 7.13, 20 itérations sont prévues. Au cours de chaque itération, les services `compute` et `reconfigure` de l'instance du composant **HeatM** se succèdent. Le service `compute` est d'abord invoqué. Il effectue la simulation numérique du domaine. Ensuite le service `reconfigure` est appelé. Il permet de reconfigurer l'application en fonction des résultats de la simulation numérique précédente.

```
component Application {
  content {
    composite {
      decl {
        HeatM h;              // h instance de type HeatM
      }
      service run {
        for ( i = 0 to 20 ) {
          exec h.compute;     // phase de calcul
          exec h.reconfigure; // phase de reconfiguration
} } } } }
```

FIGURE 7.13 – Composant composite de l'**Application** AMR contenant la boucle d'itération.

HeatM Ce composite, présenté en figure 7.15, dont le code ULCM correspondant est en figure 7.14, implémente l'algorithme de base de l'AMR. Il peut être dans deux états : subdivisé ou non. Cet état est reflété par l'attribut R. Lorsque $R = 0$, le composant n'est pas subdivisé. Lorsque $R = 1$ le composant est dans l'état subdivisé. Ces états correspondent à deux implémentations possibles de ce composant composite.

Lors de sa première exécution il est constitué d'un assemblage comprenant uniquement le composant primitif *Heat* présenté plus haut en figure 7.7. Au cours de l'exécution de l'application son état et donc son implémentation peuvent changer. Ainsi, lors de la phase de raffinement, l'implémentation constituée d'un composant primitif *Heat* est remplacée par une instance du composant composite *HmainM* (présenté dans le paragraphe suivant).

```
component HeatM {
  ports {
    client name=hu_grad type="gradient";  // déclaration des ports
    server name=hp_grad type="gradient";
    server name=hp_Nborder type="border";
    client name=hu_Nborder type="border";
    ...
    client name=hu_gdata type="GData";
    server name=hp_gdata type="GData";

    attribute name=i type=int;            // coordonnées du composant
    attribute name=j type=int;            //   dans l'assemblage
    attribute name=level type=int;        // niveau de raffinement
    attribute name=R type=int;            // état de subdivision
   }
  content {
    composite {
      decl {
        include "HmainM.ulcm"
        Heat h;                           // h instance de type Heat
        h.hu_gdata -- self.hu_gdata;      // connexion des ports
        ...
        set self.R 0;                     // état non subdivisé
      }
      service reconfigure { ... }
      service compute { ... }
} } }
```

FIGURE 7.14 – **HeatM** est le composant de base de l'implémentation. *client* et *server* permettent de décrire respectivement les ports *uses* et *provides*. Les interfaces sont *gradient* (pour le calcul de la moyenne du gradient), *border* (pour l'échange des frontières du domaine) et *GData* (pour l'initialisation du domaine).

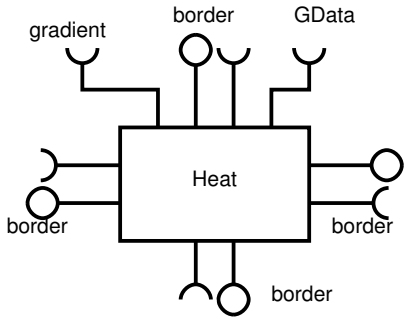


FIGURE 7.15 – Composant **HeatM** dans l'état non subdivisé.

Le composant **HeatM** possède deux services, `compute` et `reconfigure`, qui sont respecti-

vement décrit dans les figure 7.16 et 7.17.

Le service `compute` permet la simulation des domaines. Comme présenté à la figure 7.16, si le composant est dans l'état non subdivisé ($R = 0$) il transmet ses coordonnées dans l'assemblage de composants au composant *heat* qui simule le domaine puis il l'exécute. Dans le cas où le composant est dans l'état subdivisé ($R = 1$), le service `compute` transmet l'appel `compute` au composant composite *HmainM*.

```
service compute {
  if ( self.R == 0 ) { // état non subdivisé
    set h.i self.i;    // fixe les coordonnées
    set h.j self.j;    //   de h
    exec h;            // exécute h
  } else               // état subdivisé
    exec hm.compute;   // exécute le composite hm
} }
```

FIGURE 7.16 – Service *compute* du composant **HeatM**.

Le service `reconfigure` (décrit en figure 7.17) permet la transformation du composant *HeatM* dans l'état subdivisé et son retour dans l'état non subdivisé. Ce service permet le raffinement et déraffinement.

Si l'état du composant n'est pas *subdivisé* (`self.R == 0`) il vérifie s'il faut raffiner. Dans ce cas, son implémentation est réalisée par le composant *Heat*. Il teste alors l'attribut `g` du composant *Heat*. Cet attribut est l'estimation d'erreur (dans notre cas le gradient) que le composant *Heat* a calculé sur son domaine. Dans l'exemple, le service `reconfigure` teste si l'estimation d'erreur est supérieure à 80 (ligne `hg > 80`). Si c'est le cas, le service déconnecte le composant *Heat* (lignes `unset ...`), change l'état du composite à *subdivisé* (`set self.R 1;`), crée une instance du composant *HmainM* et connecte ses ports.

Si l'état du composant est déjà dans l'état *subdivisé* (`self.R == 1`) il vérifie si le raffinement est toujours nécessaire. Dans ce cas, l'implémentation du composite est réalisé par une instance du composant *HmainM*. Le service vérifie si l'estimation d'erreur issue de ce composant est trop faible (`hm.g < 10`). Si c'est le cas, il bascule dans l'état *non-subdivisé* (`set self.R 0`). Puis il reconnecte l'instance du composant *Heat* et détruit le composant *HmainM*.

```
service reconfigure {
  if ( self.R == 0 ) {                        // état non subdivisé
    if ( h.g > 80 ) {                         // erreur trop élevée
      unset h.hp_Nborder -- self.hp_Nborder; // déconnexion des ports
      ...
      set self.R 1;                           // état subdivisé
      HmainM hm;                              // instance de type HmainM
      set hm.hu_grad -- self.hu_grad;         // connexion de ses ports
      ...
    } else {                                  // état subdivisé
      exec hm.reconfigure;
      if ( hm.g < 10 ) {                      // erreur trop faible
        set self.R 0;                         // état non subdivisé
        set h.hu_gdata -- self.hu_gdata;      // connexion des ports
        ...
        del hm;                               // suppression du composite
} } } }
```

FIGURE 7.17 – Service *reconfigure* du composant **HeatM**.

HmainM Comme présenté dans la paragraphe précédent, le composant composite *HmainM* se substitue au composant **Heat** lors de l'étape de raffinement. Cette substitution est possible car le composant *HmainM* présente la même interface que le composant **Heat**.

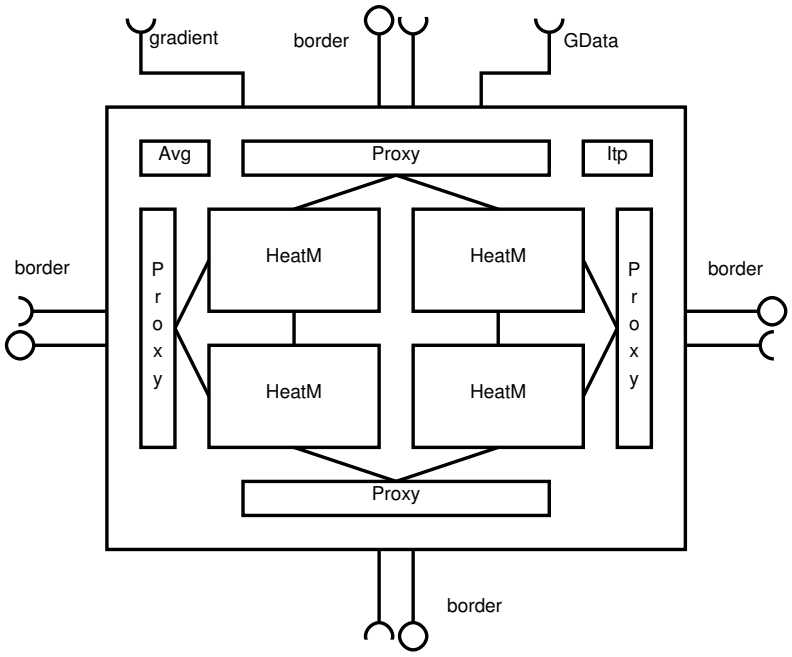


FIGURE 7.18 – Assemblage constituant le composant **HmainM**.

Ce composant est constitué d'un assemblage présenté à la figure 7.18. Il est composé de
- quatre instances du composant *HeatM* (une pour chaque sous-domaine),
- de quatre composants *Proxy* (pour permettre l'échange des frontières entre les différents composants *HeatM* d'un composant *HmainM* et le composant auquel *HmainM* sera connecté),
- du composant *Average* dont le rôle est de calculer la moyenne de l'erreur issue des quatre composants *HeatM* et enfin
- d'un composant *Interpol* qui permet de transmettre les données d'un niveau à l'autre du raffinement lors des phases de raffinement et déraffinement.

Le code ULCM de ce composant est présenté à la figure 7.19. Il possède deux services `reconfigure` et `compute` qui permettent chacun de transmettre parallèlement la requête correspondante aux niveaux inférieurs.

7.2.4 Discussion

Cette implémentation d'une version simple d'une application AMR avec ULCM a été effectuée avec des composants primitifs écrits en JAVA. ULCM étant indépendant des modèles de composants sur lesquels il repose, l'implémentation de cette application en utilisant CCM ou CHARM++ ne changerait rien aux composants présentés dans cette section. Il faudrait seulement remplacer les composants primitifs JAVA par une implémentation utilisant CCM ou CHARM++.

Cette première étude a permis de faire apparaître différentes limitations imposés par le modèle de programmation et son implémentation.

```
component Hmain {
  ports { // Similaires à HeatM }
  content {
    composite {
      decl {
        Average Av;                      // déclaration des
        Interpol Int;                    //  instances de
        HeatM h11; HeatM h12;            //  composants
        HeatM h21; HeatM h22;
        Proxy N; Proxy S;
        Proxy E; Proxy W;
        Int.hp1_gdata -- h11.hu_gdata;  // connexion des ports
        ...
      }
      service reconfigure {
        parallel {
          section: exec h11.reconfigure;
          section: exec h12.reconfigure;
          section: exec h21.reconfigure;
          section: exec h22.reconfigure;
        }
      }
      service compute {
        parallel {
          section: exec h11.compute;
          section: exec h12.compute;
          section: exec h21.compute;
          section: exec h22.compute;
        }
        set self.g Av.g;
} } } }
```

FIGURE 7.19 – Composant auxiliaire *HmainM* constitué de quatre instances du composant principal.

- L'implémentation repose sur un partitionnement *quad-tree*. La décomposition de domaine à chaque niveau est ainsi d'un facteur quatre. Bien que ULCM permette avec l'instruction *forall* d'exécuter parallèlement un nombre indéfini de composants, ce modèle ne permet pas d'exprimer d'autres schéma de décomposition. Par exemple, avec un facteur de raffinement différent suivant les deux dimensions, ou avec une décomposition de domaine irrégulière.
- L'équilibrage de charge n'est pas pris en compte. En effet, le modèle ne permet pas de définir la localisation des nouveaux composants créés lors des étapes de raffinement.
- Le moteur de *workflow* est centralisé, ce qui limite le passage à l'échelle de cette application.
- Finalement, la boucle itérative de l'application est divisée en deux étapes. La phase de *reconfiguration* ne s'exécute pas en même temps que la phase de *calcul*. Ceci est du au fait que lors de la phase de *reconfiguration* des composants sont crées ou détruits et les connexions modifiés en conséquence. ULCM ne propose pas de mécanisme permettant d'exécuter un composant dont les ports ne sont pas encore connectés.
- L'évolution de l'application fait apparaître une hiérarchie de composants primitif (*Heat*, *Proxy*) qui pourrait être simplifié et optimisée. En effet, si, par exemple, l'ensemble du domaine est raffiné quatre fois l'échange de frontières entre deux sous-domaines peut être amené à transiter à travers huit composants *Proxy* à certains endroits du domaine. Alors

qu'une connexion directe entre composants serait possible.

7.3 AMR avec SALOMÉ

Cette section étudie la conception d'application de raffinement de maillage adaptatif en utilisant le modèle de programmation offert par une plate-forme SALOMÉ. L'objectif est d'étudier les possibilités et limites d'un modèle de programmation utilisé en production. La méthode AMR est de plus en plus utilisée, c'est donc une application pouvant être amenée à être implémenté sur cette plate-forme. De plus, la structure de cette application (récursive, hiérarchique, dynamique) est similaire à d'autres applications utilisées au sein d'EDF. Il est donc intéressant d'étudier son support par ce modèle de programmation.

Comme présenté dans les chapitres précédents, la plate-forme SALOMÉ propose un modèle de programmation constitué d'un modèle de composant (DSC) sur lequel se superpose un langage de *workflow*. Ce langage de *workflow* est rendu accessible par le module YACS. Dans le contexte de YACS un *workflow* est appelé schéma de couplage et une tâche correspond à un noeud du schéma de couplage.

Le module YACS permet de concevoir ces schémas de couplage en liant des noeuds par différents types de connexions et en utilisant différentes structures de contrôle. Les noeuds d'un schéma de couplage sont implémentés par les services offerts par les composants de l'application. Les ports de type *dataflow* d'un noeud du schéma de couplage correspondent aux arguments du service correspondant au noeud et implémenté par un composant DSC. Les ports de type *datastream* d'un noeud correspondent aux ports DSC du composant implémentant le services correspondant. C'est la définition des liens *datastream* entre les noeuds d'un schéma de couplage qui permet à YACS de connecter les ports DSC des composants correspondant lors de l'exécution du schéma.

Bien que le modèle de composants DSC permette la création de composants à la demande, YACS ne permet pas de définir des schémas de couplage avec création de noeuds à la demande. Le schéma de couplage composé dans YACS n'est donc pas modifiable lors de son exécution. Cependant la définition d'un schéma de couplage peut se faire de trois manières différentes. En utilisant l'interface graphique offerte pas le module YACS il est possible de définir un schéma de couplage puis de l'exécuter en contrôlant et visualisant son exécution. Ce schéma est ensuite sauvée sous forme *XML*. Il est aussi possible de concevoir le schéma directement sous forme *XML* et de l'exécuter sans l'interface graphique. La troisième façon de définir un schéma de couplage est d'utiliser l'Api Python proposée par YACS. Cette méthode permet de définir et exécuter un schéma de calcul (instancier et connecter les noeuds de calcul et structures de contrôle) dans un script Python. Il est donc possible, grâce à cette Api, de définir un schéma de calcul et de le modifier en fonction de résultats d'une exécution précédente.

C'est cette méthode qui a été choisie pour implémenter l'AMR avec SALOMÉ.

7.3.1 Composants SALOMÉ

La méthode AMR implémentée est la même que celle présentée dans la section Ulcm. Similairement, quatre composants primitifs sont utilisés :

- *Heat* permet la simulation d'un domaine,
- *Proxy* permet la redistribution des frontières entres différents niveaux de raffinement,
- *Average* collecte les estimations d'erreur issues des composants *Heat*,
- *Interpol* permet la redistribution des résultats de simulation entre les niveaux de raffinement (utilisé pour l'initialisation des nouveaux composants lors du raffinement et déraffinement).

Ces composants possèdent les mêmes ports que quand dans l'implémentation Ulcm. Afin de pouvoir définir un schéma de couplage avec YACS, les composants doivent implémenter des services.

Le seul composant à réellement proposer un service est le composant *Heat*. Celui-ci propose un service de simulation d'un domaine. Les trois autres composants (*Proxy*, *Average*, *Interpol*) ne proposent pas de services, ils ne font que fournir ou utiliser des interfaces. Or, c'est le module YACS qui demande la connexions des ports des composants. Il le fait lorsqu'il rencontre des noeuds connectés, prêts à être exécutés, dans un schéma de calcul. Le noeud étant implémenté par un service, il est donc nécessaire d'ajouter un service aux composants *Proxy*, *Average* et *Interpol* pour qu'ils puissent apparaître dans un schéma de calcul et ainsi voir leur ports connectés.

7.3.2 Structure de l'application

L'application est construite grâce à un script Python utilisant l'Api de YACS. Le modèle de programmation de la plate-forme SALOMÉ ne proposant pas le concept de composant composite l'architecture de l'application a donc été définie directement ce script Python. Ce script construit la structure de l'application et le schéma de calcul correspondant puis l'exécute.

Une version simplifiée de ce script est présentée en figure 7.3.2. Ce script joue le rôle des services *compute* et *reconfigure* des composites de l'implémentation en Ulcm. De la même manière, l'exécution est divisée en deux phases : calcul et reconfiguration.

À la fin du script (à partir de la ligne 142) on trouve une boucle dont les paramètres définissent le nombre d'itérations à effectuer. Au cours d'une itération se succèdent `A.modif()` et `yacs_execute(first)` qui correspondent respectivement aux services *reconfigure* et *compute* du composant *application* de l'implémentation Ulcm 7.13. L'appel `A.modif()` permet la modification du schéma de couplage en fonction de l'exécution précédente. `yacs_execute(first)` permet l'exécution du schéma une fois modifié.

```
1  # initialisation du schema
2  def yacs_init():
3    global p,r,e,td,ti
4    SALOMERuntime.RuntimeSALOME_setRuntime()
5    r = pilot.getRuntime()
6    e = pilot.ExecutorSwig()
7    p = r.createProc("pr")                  # creation de la procedure
8                                            # contenant le schema de calcul
9  # execution du schema de calcul
10 def yacs_execute(firsttime):
11   global p,e
12   if firsttime :                          # premiere execution
13     e.RunW(p,0,True)                      # execution
14   else:
15     p.setState(pilot.ACTIVATED)           # changement de l'etat
16     p.exUpdateState()                     # de la procedure
17     e.RunW(p,0,False)                     # execution
18
19 # creation d'un noeud heat
20 def yacs_heat(parent, node_name, level, number):
21   global r, ti
22   nn=r.createCompoNode("",node_name)      # creation du noeud
23   nn.setRef("Heat")                       # choix de l'implementation
24   nn.setMethod("run")                     # service correspondant au noeud
25   parent.edAddChild(nn)                   # insertion de noeud dans le schema
26
27   nn.edAddInputDataStreamPort("P_BN",td)# ajout des ports du noeud
```

```
28    nn.edAddInputDataStreamPort("P_BE",td)#       P correspond a un port provides
29    nn.edAddInputDataStreamPort("P_BS",td)#       U correspond a un port uses
30    nn.edAddInputDataStreamPort("P_BW",td)
31    nn.edAddOutputDataStreamPort("U_G",td)
32    nn.edAddOutputDataStreamPort("U_Data",td)
33    nn.edAddOutputDataStreamPort("U_BN",td)
34    nn.edAddOutputDataStreamPort("U_BE",td)
35    nn.edAddOutputDataStreamPort("U_BS",td)
36    nn.edAddOutputDataStreamPort("U_BW",td)
37    return nn
38
39  # creation d'un noeud gradient
40  def yacs_grad(parent, node_name, level):
41    ...
42
43  # creation d'un noeud proxy
44  def yacs_proxy(parent, node_name, level, number):
45    ...
46
47  # creation d'un noeud average
48  def yacs_interpol(parent, node_name,level,size):
49    ...
50
51  class amr:
52    def __init__(self,idd,parent,Compo,level,number):
53      self.ID = idd
54      self.children = []
55      self.grad =  None
56      self.parent = parent
57      self.level = level
58      self.bloc = None
59      if Compo == "heat":
60        self.n = yacs_heat(parent, self.ID,self.level,number)
61      elif Compo == "grad":
62        self.n = yacs_grad(parent, self.ID,self.level)
63      elif Compo == "interpol":
64        self.n = yacs_interpol(parent, self.ID,self.level)
65      elif Compo == "proxy":
66        self.n = yacs_proxy(parent, self.ID,self.level,number)
67
68    def raffine(self):
69      #create a bloc b
70      self.bloc = r.createBloc("b"+self.ID)
71      self.parent.edAddChild(self.bloc)
72      self.children.append(amr(self.ID+"H1",self.bloc,"heat",self.level+1,1))
73      self.children.append(amr(self.ID+"H2",self.bloc,"heat",self.level+1,2))
74      self.children.append(amr(self.ID+"H3",self.bloc,"heat",self.level+1,3))
75      self.children.append(amr(self.ID+"H4",self.bloc,"heat",self.level+1,4))
76      self.children.append(amr(self.ID+"G5",self.bloc,"grad",self.level+1,5))
77      self.children.append(amr(self.ID+"I6",self.bloc,"interpol",self.level+1,6))
78      self.children.append(amr(self.ID+"P1",self.bloc,"proxy",self.level+1,7))
79      self.children.append(amr(self.ID+"P2",self.bloc,"proxy",self.level+1,8))
80      self.children.append(amr(self.ID+"P3",self.bloc,"proxy",self.level+1,9))
81      self.children.append(amr(self.ID+"P4",self.bloc,"proxy",self.level+1,10))
82      self.bloc.exUpdateState()
83
84  # connexion des ports
85    def connect(self):
86      if self.level >= 0:
87        p.edAddLink(self.bloc.getChildByName(
88          self.children[0].ID).getOutputDataStreamPort("U_BN"),
```

```
89            self.bloc.getChildByName(self.children[2].ID).getInputDataStreamPort("P_BN"))
90          p.edAddLink(self.bloc.getChildByName(
91            self.children[0].ID).getOutputDataStreamPort("U_BW"),
92            self.bloc.getChildByName(self.children[1].ID).getInputDataStreamPort("P_BW"))
93          ...
94
95    def reinit(self):
96      if self.bloc != None :
97        self.connect()
98        self.bloc.init(False)
99        self.bloc.exUpdateState()
100     else :
101       self.n.init(False)
102       self.n.exUpdateState()
103
104   def get_child(self,i):
105     return self.children[i]
106
107 # choix de du raffinement
108   def modif(self):
109     if len(self.children) !=0 :
110       for i in range(4):
111         self.children[i].modif()
112     else:
113       if self.parent.getChildByName(self.ID).getOutputPort("grad").getPyObj() > 353:
114         self.raffine()
115     self.reinit()
116
117 # initialisation du schema de calcul
118 yacs_init()
119
120 # creation du premier objet AMR
121 A = amr("N0",p,"heat",0,0)
122
123 # premiere execution du schema de calcul
124 yacs_execute(first)
125
126 first = False
127
128 for i in range(20):                    # execution de 20 iterations
129
130   A.modif()                            # etape de reconfiguration
131   p.saveSchema("amr"+str(i+1)+".xml")  # sauvegarde du schema
132
133   yacs_execute(first)                  # execution du schema
```

7.3.2.1 Construction du schéma de couplage

La phase de reconfiguration est la construction du schéma de couplage. Le script commence par créer des noeuds et leur ports, les initialise et connecte les noeuds du schéma. Les noeuds d'un schéma de calcul peuvent être de deux types : élémentaires ou composés. Les noeuds élémentaires correspondent à des services implémentés par des composants. Les noeuds composés correspondent à des structures de contrôles (bloc, while, switch, boucle parallèle, etc).

Le schéma de calcul est lui-même un noeud composé de type bloc. C'est le premier noeud qui est crée lors de l'appel à la routine `yacs_init()` avec la ligne `p = r.createProc("pr")`. Ce noeud `p` est la base du schéma. Tous les autres noeuds du schéma de couplage s'y attacheront.

Les autres noeuds de l'application sont ensuite créés grâce à la classe `AMR`. Cette classe

permet de construire un arbre d'objets AMR correspondant à l'arbre des sous-domaines lors de l'exécution de l'application. Chaque objet AMR possède un nom, un noeud (de type *heat*, *proxy*, *average* ou *interpol*), des références vers le noeud parent et les noeuds fils. La classe AMR possède plusieurs méthodes :

- `__init__` (le constructeur) permet de créer le noeud et ses ports. Celui-ci est inséré dans le schéma de calcul grâce à la référence vers le noeud parent.
- `raffine` permet l'étape de raffinement. Cette méthode crée un noeud composé de type *bloc* ainsi que dix noeuds de calcul : quatre noeuds *heat*, quatre noeuds *proxy*, un noeud *average* et un noeud *interpol*). Ces noeuds sont chacun associés à un objet AMR et insérés dans la hiérarchie AMR.
- `connect` permet de connecter les ports *datastream* des noeuds nouvellement crées.
- `modif` permet de tester si l'erreur (ici le gradient) est trop élevé. Et donc s'il faut raffiner.

Le premier noeud de l'application AMR est crée avec la ligne `A = amr("N0",p,"heat",0,0)`. Ce noeud s'appelle *N0*, s'attache au noeud *p* (base du schéma de calcul) et est de type *heat*. Le schéma de calcul est donc constitué d'un seul noeud *heat* qui simule la totalité du domaine.

Une boucle permet ensuite d'effectuer un nombre donné d'itérations. `yacs_execute(first)` exécute le schéma, `A.modif()` utilise les données issues de l'exécution précédente pour raffiner si nécessaire.

7.3.2.2 Exécution du schéma

L'exécution du schéma de couplage ce fait grâce à l'appel `yacs_execute(first)`. Cette routine appelle la méthode `RunW()` de l'exécuteur de YACS. Celle-ci parcourt le schéma de couplage chargé en mémoire et lance les noeuds lorsque nécessaire. Chaque noeud d'un schéma de couplage possède un état qui permet à l'exécuteur de YACS de décider s'il doit le lancer ou pas. Cet état est initialisé lorsque le noeud est créé ou lors de l'appel à `exUpdateState`. Après exécution, le noeud change d'état et ne peut donc plus être lancé par YACS.

C'est pourquoi la routine `yacs_execute(first)` est divisée en deux partie : une première pour la première exécution du schéma et une seconde pour les exécutions suivantes. Lors de la première exécution `RunW(p,0,True)`, le paramètre `True` signifie à l'exécuteur de réinitialiser l'état de tous les noeuds du schéma. Lors des exécutions suivantes, il faut modifier l'état de certains noeuds pour permettre une nouvelle exécution du schéma de calcul. Ainsi, le noeud de base du schéma de calcul `p = r.createProc("pr")` doit voir son état réinitialisé à chaque itération. Les composants ajoutés dans l'étape de raffinement doivent être initialisés et les composants déjà présents doivent conserver leur état pour ne pas perdre les données simulées.

7.3.3 Analyse

La réalisation de cette implémentation a permis de mettre en évidence les limitations du modèle de programmation proposé par la plate-forme SALOMÉ. Le modèle de composant DSC permet la création dynamique de composants et de ports DSC mais le langage de *workflow* proposé par YACS ne permet pas la création de noeuds à la demande dans un schéma de calcul. Ce manque de dynamicité peut être contourné en utilisant l'Api python de YACS. Il est néanmoins nécessaire de prendre en considération des aspects techniques comme la gestion de l'état interne des noeuds.

Par ailleurs, le modèle de programmation de la plate-forme SALOMÉ ne supporte pas le concept de composant composite. Une grande partie de la complexité de ce script python est due à la gestion de l'arbre de composants et à la gestion des connexions/déconnexions de leurs ports. Le concept de composant composite aurait pu fortement simplifier ce processus.

Finalement, comme pour ULCM, SALOMÉ possède également deux limitations majeures. D'une part, le moteur de *workflow* de YACS est centralisé. D'autre part, la phase de reconfiguration ne peut pas s'exécuter en même temps que la phase de calcul. L'impact de cette dernière limitation peut néanmoins être diminué en n'effectuant pas l'étape de reconfiguration à chaque pas de calcul.

7.4 Évaluation

L'évaluation des limitations du modèle de programmation de SALOMÉ a été menée grâce à plusieurs expérimentations. Elles ont été conduites sur la plate-forme expérimentale GRID'5000 sur le cluster *graphene* de Nancy constitué de 144 noeuds avec 1 CPU par noeud, 4 coeurs par CPI et 16 Go de RAM. Les expériences visent à comparer l'exécution locale et distribuée de l'application pour différentes tailles de données afin de déterminer différents surcoûts.

7.4.1 Évolution d'un scénario

La première expérience consiste en l'observation de l'exécution d'un scénario. Un scénario présente l'exécution contrôlée (c'est-à-dire que le choix du composant à raffiner est fixé par l'expérimentation) et simplifiée de l'algorithme AMR en vue de la mesure des temps d'exécutions des différentes étapes de l'algorithme. C'est un scénario simplifié car les composants *proxy* ne sont pas instanciés – il n'y a donc pas de communications inter-composites.

Au cours de ce scénario les noeuds sont successivement raffinés. La simulation débute avec un seul noeud. Il est exécuté un certain nombre d'itérations puis intervient l'étape de raffinement. Au cours de cette étape le noeud est subdivisé. Cette version simplifiée provoque donc la création de quatre noeuds *heat* et un *gradient* – il n'y a pas de noeuds *proxy*. Le scénario se poursuit en exécutant les noeuds un certain nombre d'itérations puis en subdivisant un noeud du plus bas niveau. Ce scénario permet d'atteindre un nombre de 105 noeuds de calcul répartis sur 11 niveaux de raffinement comprenant chacun un seul noeud subdivisé par niveau.

La figure 7.20 présente l'évolution du temps d'exécution par itération au cours de ce scénario pour une taille de domaine de 256x256 et 10 itérations entre chaque reconfiguration. La courbe en pointillé correspond à l'exécution distribuée et l'autre à l'exécution locale.

On remarque que la première exécution après chaque étape de reconfiguration est beaucoup plus longue que les autres. Cela est du à la création des conteneurs des nouveaux composants. En effet, lors de chaque reconfiguration, de nouveaux noeuds sont ajoutés au schéma de calcul. L'exécution du schéma YACS demande donc la création des composants correspondants.

La figure 7.21 présente l'évolution du temps de reconfiguration et d'exécution de la première itération après chaque reconfiguration, divisés par le nombre de noeuds, en fonction du nombre de composants. Il s'agit du même scénario que précédemment, à la différence que dans ce nouveau scénario, à chaque reconfiguration, l'intégralité des noeuds est recréée. Les composants sont donc, également recrées

La courbe avec des $*$ présente le temps de reconfiguration par noeud en millisecondes, c'est-à-dire le temps d'ajout d'un noeud dans le schéma de calcul. La courbe montre que le temps nécessaire à l'ajout d'un noeud au schéma de calcul est constant quelque soit le nombre de noeud à ajouter. Deux dixièmes de milliseconde par composant est assez faible pour un nombre pas trop grand composants – typiquement une grappe d'une centaine de noeuds. Néanmoins, cela est trop coûteux pour des supercalculateurs avec des dizaines de milliers de noeuds.

Les deux autres courbes présentent le temps en seconde nécessaires à la création d'un conteneur et l'exécution d'une itération de calcul par composant. On remarque également que le temps de création de conteneur par composant est constant par rapport au nombre de com-

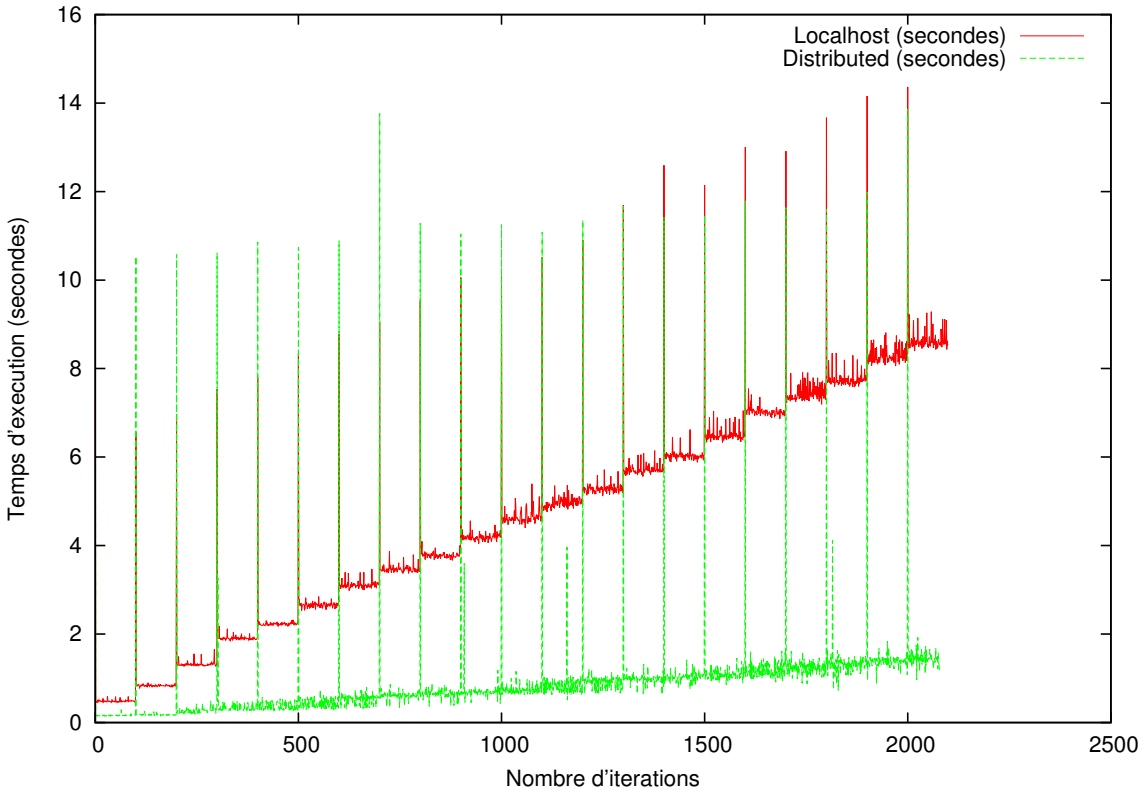


FIGURE 7.20 – Temps d'exécution en local et distribué pour une taille de domaine de 256x256 et 10 itérations entre chaque étape de reconfiguration.

posant. La création de ces conteneurs est effectuée en série par SALOMÉ. À chaque création de conteneur, SALOMÉ attends 1 seconde pour vérifier son lancement. On retrouve ce temps d'une seconde correspondant à la création d'un conteneur.

Le temps de création des composants est constant et de une seconde, ce qui pose problème lors de l'exécution avec un grand nombre de noeuds. Il faudrait réduire ce temps en implémentant par exemple un algorithme de déploiement qui passe à l'échelle – typiquement avec un complexité en logarithme du nombre de composants à créér.

7.4.2 Temps d'exécution entre étapes de reconfiguration

L'expérimentation consiste en l'exécution du scénario initial, c'est-à-dire le raffinement contrôlé de l'application. Elle est présentée en figure 7.22. Elle permet de visualiser l'évolution du temps d'exécution entre les étapes de reconfiguration en fonction du nombre de processus. La figure présente les résultats obtenus avec une version locale et une version distribuée.

On observe que le temps d'exécution augmente après chaque étape de reconfiguration. Ceci est du au transfert des données à travers YACS. En effet, lorsque YACS exécute chaque service, il lui transmet des arguments et il en attend certains en retour (notamment le gradient des composants *heat*). Lorsque le nombre de noeuds augmente au cours de la simulation, les transferts augmentent d'autant.

Au début de cette courbe le schéma de calcul contient 5 noeuds, à la fin 105 noeuds. L'augmentation est de de 8 secondes en local et de 1 seconde en distribué. Comme pour le déploiement, SALOMÉ offre un cout acceptable pour un cluster de quelques centaines de noeuds mais la centralisation de YACS fera apparaître des surcouts trop important pour des machines avec

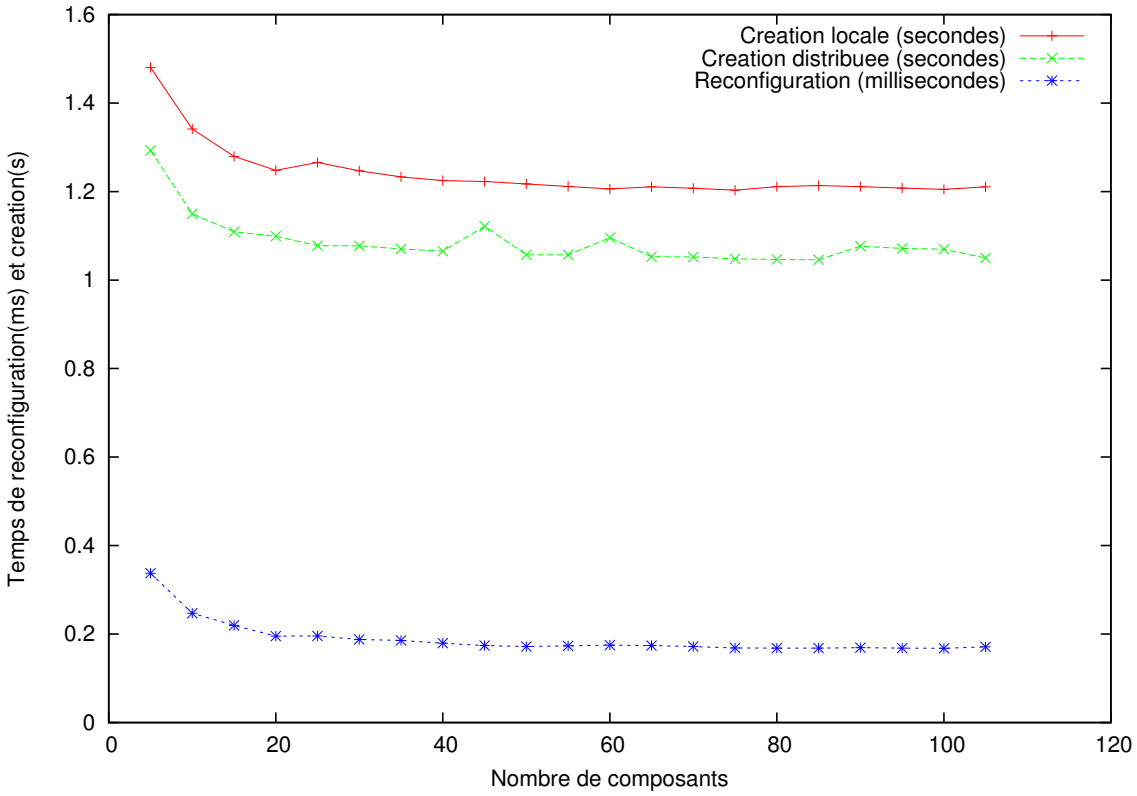


FIGURE 7.21 – Évolution du temps de reconfiguration et de création moyen par composant en fonction du nombre total de composants.

des dizaine de milliers de noeuds.

7.5 Discussion

Ces deux implémentations de l'AMR dans ULCM et SALOMÉ ont permis de faire apparaître les concepts qu'un modèle de composant doit supporter pour faciliter le développement d'application AMR.

Les **composants primitifs** permettent la réutilisation du code natif pour la simulation du domaine. Les **ports** permettent les communications directes entre composants. Le concept de *composant composite* permet de gérer plus simplement les niveaux dans la hiérarchie de composants. Le support des communications **MxN** est aussi très utile lorsque deux zones, raffinées à des niveaux différents, communiquent. Dans notre implémentation, les composants *proxy* jouent ce rôle. Une **optimisation inter-composites** permettrait aussi de simplifier la chaîne de composants *proxy* qui apparaît lorsque le niveau de raffinement augmente. Un tel problème n'est pas pris en compte avec les modèles de composants courants. Néanmois, un modèle tel que HLCM offre une solution à ce problème. Une solution moins radicale serait d'appréhender l'assemblage de composants au niveau primitif comme dans SALOMÉ, mais cela complique le code utilisateur.

Par ailleurs, la possibilité de **créer et détruire dynamiquement des instances de composants** est une propriété fondamentale que doit posséder le modèle. L'algorithme AMR crée et détruit des composants en fonction des raffinements et déraffinements.

L'utilisation d'un **langage de description d'architecture (ADL)** est aussi le bienvenu. Il

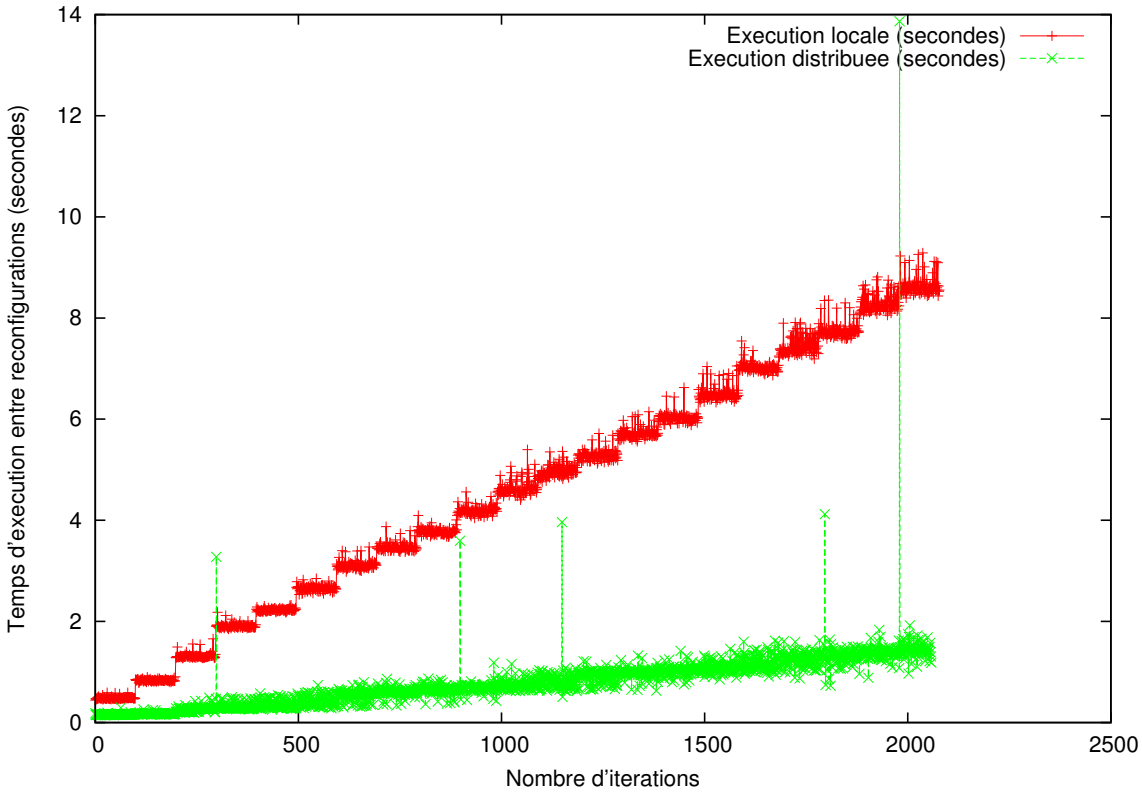


FIGURE 7.22 – Temps d'exécution entre chaque étape de de reconfiguration au cours de l'exécution d'un scénario.

permet de séparer la structure du code de son aspect fonctionnel. Néanmoins, dans le cas d'une application dynamique, l'ADL doit supporter l'aspect temporel/workflow de la structure de l'application. Il est aussi nécessaire de concevoir un moteur d'exécution distribué pour permettre l'exécution à grande échelle de l'application.

Enfin, la **généricité** [41] serait aussi intéressante. Elle permettrait de réutiliser un squelette[85] AMR pour différentes applications et ainsi faciliterait la conception d'applications AMR.

7.6 Conclusion

Ce chapitre a présenté et discuté l'implémentation d'applications AMR avec deux modèles de composants spatio-temporels. Les deux modèles utilisés, ULCM et SALOMÉ, permettent chacun de l'implémenter. Néanmoins, le niveau de complexité n'est pas le même. En particulier, le manque de création dynamique et de composant composite dans SALOMÉ complexifie fortement le développement.

Par ailleurs, deux facteurs limitants sont l'utilisation d'un moteur d'exécution centralisé et l'impossibilité de paralléliser la phase de reconfiguration et de calcul.

Troisième partie

Conclusion

Chapitre 8

Conclusion et perspectives

8.1 Conclusion générale

8.1.1 Objectif et problématique

Les simulations numériques sont de plus en plus utilisées dans tous les champs de recherche. Elles permettent de simuler des phénomènes qui sont inaccessibles à l'expérimentation, que ce soit par impossibilité physique, ou pour raison budgétaire. Cependant, l'évolution des ressources de calcul permet d'envisager des simulations de plus en plus précises.

Il devient possible de simuler des phénomènes en couplant des modèles existants. Chacun de ces modèles étant spécialisé dans un domaine particulier. Ce couplage de modèles amène de plus en plus à construire des applications à l'architecture complexe. Faire fonctionner ces différents modèles ensemble est une tâche délicate. Les modèles ne sont généralement pas développés par les mêmes équipes et n'ont souvent pas été conçus originellement dans l'optique d'interagir avec d'autres modèles.

De plus, l'évolution des ressources de calcul implique aussi une complexification de la conception des applications scientifiques. En effet, l'exécution efficace d'une application nécessite une optimisation en fonction des ressources visées, ce qui est une tâche très complexe qui peut demander du temps (jusqu'à plusieurs années). Les applications ayant généralement une durée de vie ($\sim$30 ans) supérieure à la durée de vie moyenne d'une ressource ($\sim$3 ans), il est donc régulièrement nécessaire de porter une application d'une architecture à la suivante.

L'utilisation de modèles de programmation adaptés est une solution intéressante pour faire face à cette complexité. Ils permettent d'une part de mieux structurer les applications et d'autre part de simplifier l'adaptation aux ressources en minimisant les efforts de portage.

La programmation par composants logiciels est un modèle de programmation entrant dans cette catégorie. Ils permettent de modulariser l'architecture d'une application en définissant clairement les rôles et interactions de chacun. La programmation par composants peut être vue comme une évolution de la programmation objet. Dans la programmation objet, une classe exprime les services qu'elle offre par le biais de son interface. Il est donc possible de construire une application en utilisant les classes offrant les services nécessaires. La programmation par composants logiciels propose d'étendre ce mécanisme de déclaration d'interfaces aux dépendances. Ainsi, un composant déclare explicitement par le moyen de ports les interfaces qu'il utilise et fournit. Les composants peuvent être considérés comme des boites noires. Toute l'information nécessaire à leur utilisation est comprise dans la déclaration de leurs ports.

La construction d'une application avec ce modèle de programmation se fait par composition. Les ports des composants sont connectés, faisant correspondre les apports des uns aux besoins des autres. Ce modèle de programmation permet de modulariser le code en séparant

correctement les rôles des différentes parties. De plus, les interfaces étant clairement définies, il devient aisé de remplacer l'implémentation d'un composant par une autre. Il est aussi possible de réutiliser un composant dans plusieurs applications sans modification.

Néanmoins, les modèles de composants existants ne permettent pas encore de développer simplement et efficacement certains types d'applications scientifiques. Ainsi, les applications de décomposition de domaine ou de raffinement de maillage présentent une structure difficilement exprimable dans les modèles de composants actuels. Hiérarchiques, dynamiques et récursives, ces applications sont de plus constituées d'un motif répliqué en fonction de la configuration d'exécution.

Cette thèse a étudié et proposé des solutions pour permettre la conception de ces applications avec les modèles de composants. Ces contributions sont les suivantes.

8.2 Contributions

8.2.1 Ajout dynamique d'une cardinalité aux noeuds de la plate-forme SALOMÉ

La conception d'applications de décomposition de domaine avec le modèle de programmation de la plate-forme SALOMÉ est une tâche complexe et fastidieuse. En effet, ce type d'application possède une structure dépendante d'un paramètre (le nombre de sous-domaines de la décomposition), mais le modèle de programmation de la plate-forme ne permet pas de l'exprimer.

Nous avons proposé une extension au modèle de programmation de la plate-forme SALOMÉ pour permettre la conception de ce genre d'applications. Cette extension permet d'associer une cardinalité à chaque composant de l'application. Ainsi, lors de l'exécution, la plate-forme peut adapter la forme de l'application en dupliquant les composants en fonction de cette cardinalité. Cette extension implique un mécanisme permettant de gérer les interactions des composants dupliqués avec les autres composants de l'application.

Ainsi, l'expressivité du modèle de programmation de la plate-forme SALOMÉ a donc été augmentée et la conception de ce type d'application a été simplifiée. L'extension a été appliquée à une application de décomposition de domaine expérimentale développée chez EDF.

8.2.2 Utilisation d'un modèle de composants logiciels de bas niveau pour la conception d'applications

La seconde contribution a étudié l'utilisation d'un modèle de composants logiciels de bas niveau supportant nativement des connecteurs (MPI, appel de méthode, CORBA) pour l'adaptation des applications aux ressources. À travers l'implémentation d'une forme simple de décomposition de domaine, nous avons montré que l'utilisation de ce modèle de composants a permis d'augmenter le taux de réutilisabilité entre les différentes versions de code et a offert des possibilités de composition (notamment pour les ressources hétérogènes) tout en conservant des performances similaires à celles de l'application native.

8.2.3 Conception d'applications de raffinement de maillage adaptatif à base de composants

La troisième contribution a concerné l'étude de la conception d'applications de raffinement de maillage adaptatif dans deux modèles de composants (ULCM et SALOMÉ). Cette contribution a permis de mettre en avant les limitations des modèles de composants et de leurs implémentations. Ce type d'application a besoin de pouvoir créer des composants à la demande, ce qui

implique d'avoir un modèle de composition supportant les compositions spatiales et temporelles ainsi que la création de nouvelles instances. Par ailleurs, l'application de raffinement de maillage présente une structure hiérarchique et récursive. Le modèle de programmation doit donc proposer le concept de composant composite pour supporter simplement la conception de ce genre d'applications. En effet, il facilite la création et la gestion des niveaux de raffinement au cours de l'exécution et permet d'implémenter la récursivité.

8.3 Perspectives

Les études menées au cours de cette thèse ouvrent différentes perspectives à plus ou moins long terme. L'utilisation de l'extension présentée dans le chapitre 5 et l'ajout du concept de composant composite à SALOMÉ sont les perspectives les plus simples. À l'inverse, le support de squelettes algorithmiques par SALOMÉ ou l'extension de HLCM aux applications dynamiques est une tâche plus complexe.

8.3.1 Utilisation de l'extension du modèle de programmation de SALOMÉ pour la conception d'applications AMR

L'extension du modèle de programmation de la plate-forme SALOMÉ présentée dans le chapitre 5 pourrait être utile dans le contexte de la conception de l'application AMR présentée dans le chapitre 7. En effet, une des caractéristiques de l'application AMR est qu'elle a besoin de créer à la demande des instances de composants au cours de son exécution. C'est la raison qui a poussé à décomposer l'application en deux phases : calcul et reconfiguration. La reconfiguration pouvant modifier le schéma de calcul entre les phases de calcul. La possibilité de dupliquer un noeud à la demande lors de l'exécution d'un schéma permettrait de simplifier la phase de reconfiguration.

8.3.2 Augmentation du niveau d'abstraction dans SALOMÉ

La conception de l'application de décomposition de domaine et d'AMR dans la plate-forme SALOMÉ a permis de mettre au jour le manque d'abstraction du modèle de programmation. En effet, le concept de composant composite simplifierait la conception de ces applications. Dans le cas de l'application AMR, un composant composite permettrait de définir explicitement un niveau de raffinement, ce qui, avec la possibilité de dupliquer les noeuds, permettrait d'intégrer les étapes de raffinement à l'exécution du schéma. L'ajout du concept de composite demande la définition d'un nouveau type de noeud dans YACS et de mécanismes permettant à l'exécuteur de connecter les instances de noeuds.

Par ailleurs, l'ajout de mécanismes d'abstraction plus poussés comme la possibilité de définir de squelettes algorithmiques serait un plus certain. Il permettrait, par exemple, de proposer un squelette pour l'application de décomposition de domaine dont les paramètres seraient le type des composants de simulation et le nombre de sous-domaines. Cependant, l'ajout de ces concepts demande la création d'un niveau de composants plus abstrait et l'utilisation d'algorithmes pour concrétiser de tels composants.

8.3.3 Extension du modèle HLCM à la dynamicité

Le modèle de composants L^2C qui est utilisé dans le chapitre 6 est conçu pour être utilisé avec un modèle de composants plus abstrait. Le modèle de composants HLCM peut jouer ce rôle. L'application conçue avec le modèle de composants HLCM est ensuite "compilée" pour

obtenir un assemblage de composants L^2C adapté à une configuration d'exécution précise. Néanmoins, HLCM ne supporte actuellement que les applications dont la structure est statique. Il pourrait être intéressant d'ajouter le support des applications dynamiques à HLCM. Ainsi, il serait possible de définir plus simplement des applications dont la structure évolue au cours de l'exécution. En conjonction avec le modèle de composant L^2C, cela permettrait de simplifier la conception de ces applications tout en optimisant leur adaptation à des ressources hétérogènes.

8.3.4 Algorithmes de choix

Le chapitre 6 a présenté le modèle de composant L^2C qui permet une adaptation fine aux ressources d'exécution. L'assemblage de composants L^2C est conçu pour être généré automatiquement par un "compilateur" à partir d'un modèle de plus abstrait comme HLCM. Ainsi, la transformation de la description abstraite d'une application HLCM est convertie en une version concrète adaptée aux ressources. L'étape de transformation depuis le modèle abstrait vers le modèle concret implique des choix dépendants des ressources d'exécution. Le développement d'algorithmes de choix basés sur des modèles de ressources est un problème de recherche ouvert. Notamment les questions telles que la possibilité de définir des algorithmes de choix généraux ou la composition de tels algorithmes.

Bibliographie

[1] Albert Einstin Institute. http://www.aei.mpg.de/english/contemporaryIssues/home/index.php.

[2] Anr lego. http://graal.ens-lyon.fr/LEGO/.

[3] Apache openoffice. *UNO Component Model*. http://www.openoffice.org/udk/common/man/componentmodel.html.

[4] Berkeley Open Infrastructure for Network Computing. http://http://boinc.berkeley.edu/.

[5] Cactus. http://www.cactuscode.org/.

[6] Center for Computation & Technology. http://www.cct.lsu.edu/home.

[7] Code Aster. http://www.code-aster.org/.

[8] EDF. http://www.edf.fr/.

[9] ESMF. http://www.esmf.ucar.edu.

[10] Eucalyptus. http://www.eucalyptus.com/.

[11] Mozilla. *The XPCom projet*. http://www.mozilla.org/projets/xpcom.

[12] ObjectWeb : Open Source Middleware. https://www.objectweb.org.

[13] OpenNebula. http://www.opennebula.org/.

[14] The osgi alliance. *OSGi Service Platform Specifications*. http://www.osgi.or.

[15] PALM. http://www.cerfacs.fr/globc/PALM_WEB/index.html.

[16] The SALOME Plateform. http://www.salome-platform.org/.

[17] CORBA component model, v4.0. Document formal/2006-04-01, OMG, April 2006.

[18] Amazon Elastic Compute Cloud, 2010. http://aws.amazon.com/ec2/.

[19] EGI, European Grid Initiative, 2010. http://www.egi.eu.

[20] GÉANT2 Network, 2010. http://www.geant2.net/.

[21] The FutureGrid Project, 2010. http://futuregrid.org.

[22] The TeraGrid Project, 2010. https://www.teragrid.org.

[23] Top 500 supercomputers, 2010. http://www.top500.org.

[24] CycleCloud Achieves Ludicrous Speed! (Utility Supercomputing with 50,000-cores), 2012. http://blog.cyclecomputing.com/2012/04/cyclecloud-50000-core-utility-supercomputing.html.

[25] Green 500 supercomputers, 2012. http://www.green500.org/.

[26] M. Aldinucci, H. Bouziane, M. Danelutto, and C. Pérez. Stkm on sca : a unified framework with components, workflows and algorthmic skeletons. In *15th International European Conference on Parallel and Distributed Computing (Euro-Par 2009)*, LNCS, Delft, Pays-Bas, 2009.

[27] Marco Aldinucci, Marco Danelutto, Hinde Lilia Bouziane, and Christian Pérez. Towards software component assembly language enhanced with workflows and skeletons. In *Proceedings of the 2008 compFrame/HPC-GECO workshop on Component based high performance*, CBHPC '08, pages 3 :1–3 :11, New York, NY, USA, 2008. ACM.

[28] Marco Aldinucci, Alessandro Petrocelli, Edoardo Pistoletti, Massimo Torquati, Marco Vanneschi, Luca Veraldi, and Corrado Zoccolo. Dynamic reconfiguration of grid-aware applications in assist. In *Proceedings of the 11th international Euro-Par conference on Parallel Processing*, Euro-Par'05, pages 771–781, Berlin, Heidelberg, 2005. Springer-Verlag.

[29] I. Altintas, A. Birnbaum, K. K. Baldridge, W. Sudholt, M. Miller, C/ Amoreira, Y. Potier, and B. Ludaescher. A framework for the design and reuse of grid workflows. In *International Workshop on Scientific Aspects of Grid Computing*, pages 120–133. Springer-Verlag, 2005.

[30] Ilkay Altintas, Adam Birnbaum, Kim K. Baldridge, Wibke Sudholt, Mark Miller, Celine Amoreira, and Yohann. A framework for the design and reuse of grid workflows. In *First Intl. Workshop on Scientific Applications of Grid Computing (SAG'04))*, pages 120–133, Berlin/Heidelberg, 2005. Springer.

[31] G. Antoniu, H. Bouziane, L. Breuil, M. Jan, and C. Pérez. Unified lego component model. Technical report, ANR LEGO, feb 2009. ANR-05-CIGC-11, Available at `http://padico.gforge.inria.fr/ulcm`.

[32] Rob Armstrong, Gary Kumfert, Lois Curfman McInnes, Steven Parker, Ben Allan, Matt Sottile, Thomas Epperly, and Tamara Dahlgren. The cca component model for high-performance scientific computing. *Concurr. Comput. : Pract. Exper.*, 18(2) :215–229, 2006.

[33] Amnon Barak. The mosix multicomputer operating system for high performance cluster computing. *Journal of Future Generation Computer Systems*, 13 :4–5, 1998.

[34] Françoise Baude, Denis Caromel, Cédric Dalmasso, Marco Danelutto, Vladimir Getov, Ludovic Henrio, and Christian Pérez. Gcm : a grid extension to fractal for autonomous distributed components. *Annales des Télécommunications*, 64(1-2) :5–24, 2009.

[35] Albeaus Bayucan, Robert L. Henderson, Casimir Lesiak, Bhroam Mann, Thomas Proett, and Dave Tweten. Portable batch system : External reference specification. Technical report, MRJ Technology Solutions, November 1999.

[36] T. Bellwood, L. Clément, and C. von Riegen. Uddi spec technical committee specification, version 3.01. Technical report, W3C, 2004. `http://uddi.org/pubs/uddi-v3.0.1-20031014.htm`.

[37] M. J. Berger and P. Colella. Local adaptive mesh refinement for shock hydrodynamics. *J. Comput. Phys.*, 82(1) :64–84, 1989.

[38] M. J. Berger and J. E. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. Technical report, CA, USA, 1983.

[39] D. E. Bernholdt, B. A. Allan, R. Armstrong, F. Bertrand, K. Chiu, T. L. Dahlgren, K. Damevski, W. R. Elwasif, T. G. W. Epperly, M. Govindaraju, D. S. Katz, J. A. Kohl, M. Krishnan, G. Kumfert, J. W. Larson, S. Lefantzi, M. J. Lewis, A. D. Malony, L. C. McInnes, J. Nieplocha, B. Norris, S. G. Parker, J. Ray, S. Shende, T. L. Windus, and S. Zhou. A component architecture for high-performance scientific computing. *International Journal of High Performance Computing Applications*, 20(2) :163–202, 2006.

[40] J. Bigot. *Du support générique d'opérateurs de composition dans les modèles de composants logiciels, application au calcul à haute performance.* PhD thesis, Université de Rennes 1, IRISA, Rennes, 2010.

[41] J. Bigot and C. Pérez. Increasing reuse in component models through genericity. In *Proceedings of the 11th International Conference on Software Reuse, ICSR '09*, LNCS, pages 21–30, Berlin, Heidelberg, oct 2009. Springer-Verlag.

[42] Julien Bigot and Christian Pérez. Enabling connectors in hierarchical component models. Research Report RR-7204, INRIA, August 2010.

[43] Julien Bigot and Christian Pérez. *High Performance Scientific Computing with special emphasis on Current Capabilities and Future Perspectives*, chapter On High Performance Composition Operators in Component Models. Advances in Parallel Computing. IOS Press, 2011. To appear.

[44] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. Technical report, XEROX, October 1983. XEROX CSL-83-7.

[45] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard. Web services architecture. Technical report, W3C, 2004. http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/.

[46] H. Bouziane, C. Perez, and T Priol. A Software Component Model with Spatial and Temporal Compositions for Grid Infrastructures. In *Proc. of the 14th Intl. Euro-Par Conference*, volume 5168, pages 698–708, Las Palmas de Gran Canaria, Spain, August 2008. Springer.

[47] Hinde Lilia Bouziane, Christian Pérez, and Thierry Priol. A software component model with spatial and temporal compositions for grid infrastructures. In *Proceedings of the 14th international Euro-Par conference on Parallel Processing*, Euro-Par '08, pages 698–708, Berlin, Heidelberg, 2008. Springer-Verlag.

[48] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. Frystyk Nielsen, S. Thatte, and D. Winer. Simple object access protocol (soap) 1.1. Technical report, W3C, 2000. http://www.w3.org/TR/2000/NOTE-SOAP-20000508/.

[49] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau, and J. Cowan. Extensible markup language (xml) 1.1. Technical report, W3C, 2004. http://www.w3.org/TR/2004/REC-xml11-20040204/.

[50] E. Bruneton, T. Coupaye, and J.B. Stefani. The Fractal Component Model, version 2.0-3. Technical report, ObjectWeb consortium,, February 2004.

[51] David R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[52] Dušan Bálek and František Plášil. Software connectors and their role in component deployment. In *in Proceedings of DAIS'01, Krakow*, page p. Kluwer, 2001.

[53] Nicolas Capit, Georges Da Costa, Yiannis Georgiou, Guillaume Huard, Cyrille Martin, Grégory Mounié, Pierre Neyron, and Olivier Richard. A Batch Scheduler with High Level Components. *CoRR*, abs/cs/0506006 :9, 2005.

[54] Franck Cappello, Samir Djilali, Gilles Fedak, Thomas Herault, Frédéric Magniette, Vincent Néri, and Oleg Lodygensky. Computing on large-scale distributed systems : Xtrem web architecture, programming models, security, tests and convergence with grid. *Future Gener. Comput. Syst.*, 21(3) :417–437, March 2005.

[55] Eddy Caron and Frédéric Desprez. Diet : A Scalable Toolbox to Build Network Enabled Servers on the Grid. *International Journal of High Performance Computing Applications*, 20(3) :335–352, 2006.

[56] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP : Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[57] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (wsdl) 1.1. Technical report, W3C, 2001. http://www.w3.org/TR/wsdl.

[58] Cristian Coarfa, Yuri Dotsenko, John Mellor-Crummey, François Cantonnet, Tarek El-Ghazawi, Ashrujit Mohanti, Yiyi Yao, and Daniel Chavarría-Miranda. An evaluation of global address space languages : co-array fortran and unified parallel c. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '05, pages 36–47, New York, NY, USA, 2005. ACM.

[59] Fokke Dijkstra and Aad J. van der Steen. Integration of two ocean models within cactus. *Concurr. Comput. : Pract. Exper.*, 18(2) :193–202, 2006.

[60] J. Pellet E. H. Moussi. Evaluation et amélioration d'une méthode de décomposition de domaine non-intrusive autour de code_aster. Technical report, EDF, September 2010.

[61] Johan Eker, Jorn Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Sonia Sachs, and Yuhong Xiong. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, 91(1) :127–144, January 2003.

[62] T Fahringer, R Prodan, F Nerieri, S Podlipnig, M Siddiqui, A Villazon, and M Wieczorek. Askalon : A grid application development and computing environment. *The 6th IEEEACM International Workshop on Grid Computing 2005*, pages 122–131, 2005.

[63] T. Fahringer, J. Qin, and S. Hainzer. Specification of grid workflow applications with agwl : an abstract grid workflow language. In *Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05) - Volume 2 - Volume 02*, CCGRID '05, pages 676–685, Washington, DC, USA, 2005. IEEE Computer Society.

[64] C. Farhat and F.X. Roux. A method of finite element tearing and interconnecting and its parallel solution algorithm. *Internat. J. Numer. Meths. Engrg.*, 32 :1205–1227, 1991.

[65] Gilles Fedak, Haiwu He, and Franck Cappello. Bitdew : a programmable environment for large-scale data management and distribution. In *SC '08 : Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[66] Message Passing Interface Forum. Message passing interface standard. Technical report, University of Tennessee, May 1994.

[67] Message Passing Interface Forum. Mpi-2 : Extensions to the message-passing interface. Technical report, University of Tennessee, May 1997.

[68] Ian Foster. What is the Grid ? - a three point checklist. *GRIDtoday*, 1(6), July 2002.

[69] Ian Foster and Carl Kesselman, editors. *The Grid : blueprint for a new computing infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[70] Ian Foster, Carl Kesselman, and Steven Tuecke. The Anatomy of the Grid : Enabling Scalable Virtual Organizations. *Int. J. High Perform. Comput. Appl.*, 15(3) :200–222, 2001.

[71] N. Furmento, A. Mayer, S. McGough, S. Newhouse, T. Field, and J. Darlington. Iceni : Optimisation of component applications within a grid environment. *Journal of Parallel Computing*, 28(12) :1753–1772, December 2002.

[72] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM : Parallel Virtual Machine, A Users' Guide and Tutorial for Networked Parallel Computing*. Scientific and Engineering Computation. MIT Press, 1994.

[73] Tristan Glatard, Johan Montagnat, Diane Lingrand, and Xavier Pennec. Flexible and efficient workflow deployment of data-intensive applications on grids with moteur. *Int. J. High Perform. Comput. Appl.*, 22(3) :347–360, August 2008.

[74] Emmanuel Jeannot and Guillaume Mercier. Near-optimal placement of mpi processes on hierarchical numa architectures. In *Proceedings of the 16th international Euro-Par conference on Parallel processing : Part II*, Euro-Par'10, pages 199–210, Berlin, Heidelberg, 2010. Springer-Verlag.

[75] C.R. Johnson, S. Parker, D. Weinstein, and S. Heffernan. Component-based problem solving environments for large-scale scientific computing. *Journal on Concurrency and Computation : Practice and Experience*, 14 :1337–1349, 2002.

[76] Laxmikant V. Kale, Eric Bohm, Celso L. Mendes, Terry Wilmarth, and Gengbin Zheng. Programming Petascale Applications with Charm++ and AMPI. In D. Bader, editor, *Petascale Computing : Algorithms and Applications*, pages 421–441. Chapman & Hall / CRC Press, 2008.

[77] Subramanian Kannan, Mark Roberts, Peter Mayes, Dave Brelsford, and Joseph Skovira. *Workload Management with LoadLeveler*. IBM Press, 2001.

[78] Nicholas T. Karonis, Brian Toonen, and Ian Foster. Mpich-g2 : a grid-enabled implementation of the message passing interface. *J. Parallel Distrib. Comput.*, 63(5) :551–563, May 2003.

[79] Katarzyna Keahey and Tim Freeman. Contextualization : Providing One-Click Virtual Clusters. In *ESCIENCE '08 : Proceedings of the 2008 Fourth IEEE International Conference on eScience*, pages 301–308, Washington, DC, USA, 2008. IEEE Computer Society.

[80] S. Kohn, G. Kumfert, J. Painter, and C. Ribbens. Divorcing language dependencies from a scientific software library. In *10th SIAM Conference on Parallel Processing*, Portsmouth, March 2001.

[81] Derrick Kondo, Michela Taufer, Charles L. Brooks III, Henri Casanova, and Andrew A. Chien. Characterizing and Evaluating Desktop Grids : An Empirical Study. In *Parallel and Distributed Processing Symposium, International*, page 26, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[82] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. Openmp to gpgpu : a compiler framework for automatic translation and optimization. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 101–110, New York, NY, USA, 2009. ACM.

[83] Selma Matougui and Antoine Beugnard. Two ways of implementing software connections among distributed components. In *Proceedings of the 2005 OTM Confederated international conference on On the Move to Meaningful Internet Systems : CoopIS, COA, and ODBASE - Volume Part II*, OTM'05, pages 997–1014, Berlin, Heidelberg, 2005. Springer-Verlag.

[84] Christine Morin, Pascal Gallard, Renaud Lottiaux, and Geoffroy Vallée. Towards an efficient single system image cluster operating system. *Future Gener. Comput. Syst.*, 20 :505–521, May 2004.

[85] C. Murray. Bringing skeletons out of the closet : a pragmatic manifesto for skeletal parallel programming. *Parallel Comput.*, 30(3) :389–406, 2004.

[86] T. Oinn, M. Addis, J. Ferris, D. Marvin, T. Carver, M. R. Pocock, and A. Wipat. Taverna : A tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20 :2004, 2004.

[87] OMG. The Common Object Request Broker : Architecture and Specification (Revision 3.0.3). OMG Document formal/04-03-12, March 2004.

[88] C. Pérez, T. Priol, and A. Ribes. A parallel corba component model for numerical code coupling. *The International Journal of High Performance Computing Applications*, 17(4) :417–429, 2003.

[89] C. Pérez, T. Priol, and A. Ribes. Paco++ : A parallel object model for high performance distributed system. In *Distributed Object and Component-based Software Systems Minitrack in the Software Technology Track of the 37th Hawaii Intl Conf. on System Sciences (HICSS-37)*, page 274a, Big Island, Hawaii, USA, jan 2004.

[90] A. Ribes and C. Caremoli. Salome platform component model for numerical simulation. In *COMPSAC '07 : Proceedings of the 31st Annual International Computer Software and Applications Conference*, pages 553–564, Washington, DC, USA, 2007. IEEE Computer Society.

[91] K. Seymour, H. Nakada, S. Matsuoka, J. Dongarra, C. Lee, and H. Casanova. Gridrpc : A remote procedure call api for grid computing. In *Grid Computing*, Baltimore, 2002. LNCS.

[92] R. Srinivasan. RPC : Remote procedure call protocol specification version 2. Ietf request for comment 1831, August 1995.

[93] C. Szyperski. *Component Software – Beyond Object-Oriented Programming*. Addison-Wesley / ACM Press, 1998.

[94] Ian Taylor, Matthew Shields, Ian Wang, and Andrew Harrison. Visual grid workflow in triana. *Journal of Grid Computing*, 3 :153–169, 2005. 10.1007/s10723-005-9007-3.

[95] J. Yu and R. Buyya. A taxonomy of scientific workflow systems for grid computing. *SIGMOD Rec.*, 34 :44–49, September 2005.

[96] S. J. Zhou. Coupling climate models with the earth system modeling framework and the common component architecture. *Concurr. Comput. : Pract. Exper.*, 18(2) :203–213, 2006.

[97] Le projet GRID'5000. http://www.grid5000.org/.

[98] Le projet XSEDE. http://www.xsede.org/.

[99] Renater : Le Réseau National de Télécommunications pour la Technologie, l'Enseignement et la Recherche. http://www.renater.fr/.

[100] The Object Management Group. World Wide Web document, http://www.omg.org.

[101] World wide web consortium. http://www.w3.org/.

Résumé

La conception d'applications scientifiques à base de couplage de code est une tâche complexe car elle demande de concilier une facilité de programmation et une obtention de haute performance. En outre, les ressources matérielles (supercalculateurs, grappes de calcul, grilles) permettant leur exécution forment un ensemble hétérogène en constante évolution. Les modèles à base de composants logiciels forment une piste prometteuse pour gérer ces deux sources de complexité car ils permettent d'exprimer les interactions entre les différents constituants d'une application tout en offrant des possibilités d'abstraction des ressources. Néanmoins, les modèles existants ne permettent pas d'exprimer de manière satisfaisante les applications constituées de motifs répliqués dynamiques et hiérarchiques. Ainsi, cette thèse vise à améliorer l'existant – et en particulier la plateforme générique de simulation numérique SALOMÉ – pour une classe d'applications très répandue : les applications à base de décomposition de domaine et la variante utilisant le raffinement de maillage adaptatif. Tout d'abord, nous avons proposé d'étendre le modèle de composition spatial et temporel de SALOMÉ en ajoutant la possibilité de définir dynamiquement la cardinalité des composants. Cela demande en particulier de gérer les communications de groupes ainsi induites. La proposition a été implémentée dans SALOME et validée via une application de décomposition de domaine à base de couplage de plusieurs instances de Code_ASTER. Ensuite, nous avons étudié la pertinence d'utiliser un modèle de composant supportant des connecteurs natifs (MPI, mémoire partagée, appel de méthode) pour permettre une composition plus fine des interactions entre composants. Les résultats d'expériences montrent que des performances équivalentes aux versions natives sont obtenues tout en permettant de manipuler facilement l'architecture de l'application. Enfin, nous avons étudié les extensions nécessaires aux modèles à composants (abstraction, hiérarchie, dynamicité) pour la conception d'applications de raffinement de maillage adaptatif. Les modèles de composants spatio-temporels les plus avancés permettent ainsi d'exprimer ce type d'application mais les performances sont limitées par leur mise en œuvre centralisée ainsi que par le manque de moyens efficaces pour modifier à la volée des assemblages de composants.

Abstract

Designing scientific applications based on code coupling is a complex task. It requires both an easy programming process and high-performance. In addition, execution resources (supercomputers, computer clusters, grids) are heterogeneous and constantly evolving. Software components models offer a promising perspective to manage this double complexity because they can express interactions between the different parts of an application while providing abstraction of resources. However, existing models cannot accurately express the applications made of dynamic and hierarchical patterns. The aim of this thesis is to improve the existing models, and in particular the generic platform for numerical simulation SALOMÉ, for a class of widespread applications : applications based on domain decomposition, and its dynamic variant using adaptive mesh refinement. Firstly, we proposed to extend the spatial and temporal composition model provided by SALOMÉ, by adding the ability to dynamically set component cardinality. This requires in particular to manage group communications induced. The proposal has been implemented into SALOMÉ and validated via a domain decomposition application based on coupling several instances of Code_ASTER. Then, we have studied the relevance of using a component model supporting native connectors (MPI, shared memory, method invocation), in order to allow finer composition interactions between components. The experiment results show that performances obtained are equivalent to those of the native versions, while allowing to easily manipulate the application architecture. Finally, we studied the necessary component models extensions (abstraction, hierarchy, dynamicity) for designing adaptative mesh refinement applications. The most advanced spatio-temporal component models can express this type of application but performances are limited by their centralized implementation and by the lack of efficient ways of modifying component assembling at execution time.