HABILITATION À DIRIGER DES RECHERCHES

présentée devant

L'Université de Rennes 1
Spécialité : Informatique

par

CHRISTIAN PÉREZ

Contribution à la définition et la mise en œuvre d'un modèle de programmation à base de composants logiciels pour la programmation des grilles informatiques

soutenue le 10 novembre 2006 devant le jury composé de :

Président :
M. Desprez Frédréric, Directeur de recherche, INRIA
Examinateur :
M. Berthou Jean-Yves, Chargé de mission, EDF R&D
M. Namyst Raymond, Professeur, Université de Bordeaux
M. Priol Thierry, Directeur de recherche, INRIA
Rapporteur et examinateur
M. Danelutto Marco, Professeur associé, Unisersité de Pise (Italie)
M. Daydé Michel, Professeur, ENSEEIHT
M. Geib Jean-Marc, Professeur, Univ. Sciences et Technologie de Lille

À mes parents,

Remerciements

Je tiens tout d'abord à remercier vivement les membres du jury en commençant par Frédéric Desprez pour m'avoir fait l'honneur de présider le jury. Un merci spécial au trio de rapporteurs Marco Danelutto, Michel Daydé et Jean-Marc Geib. Enfin, je tiens à remercier Jean-Yves Berthou, Raymond Namyst et Thierry Priol pour leur participation à ce jury.

Ce manuscrit doit beaucoup aux personnes que j'ai eu le plaisir d'encadrer durant leur doctorat : Alexandre Denis, André Ribes, Sébastien Lacour, Hinde Bouziane, et Boris Daix. J'adresse également mes remerciement à Zolt Németh qui m'a permis de m'initier aux joies du calcul chimique. Je remercie également les stagiaires et ingénieurs qui ont contribué à ces travaux : Benoît Hubert, Joel Daniels, Nitin Jain, Pierre Sérodes, Landry Breuil, et Gabriel Lopez.

Je remercie chaleureusement Thierry Priol pour avoir su m'attirer dans cette belle région qu'est la Bretagne, pour la confiance qu'il a su me témoigner durant ces années et pour sa gestion de l'équipe-projet PARIS. Je remercie également tous les membres de l'équipe-projet PARIS pour l'excellente ambiance qui y a règné et les nombreuses discussions animées que nous avons pu avoir : Françoise André, Gabriel Antoniu, Jean-Pierre Banâtre, Luc Bougé, Yvon Jégou, David Margery, Christine Morin et Jean-Louis Pazat.

Je tiens également à remercier tout le personnel de l'IRISA/Centre de recherche INRIA de Rennes pour leur professionnalisme et leur gentillesse ; en particulier Claude Labit, Patrick Bouthemy, Huguette Béchu, et Maryse Auffray. Remerciements spéciaux à Élodie Communier.

Merci à tous ceux que j'oublie et qui d'une façon ou d'une autre ont permis la réalisation de cette habilitation.

Table des matières

1 Introduction **1**
1.1 Les grilles informatiques 1
1.2 De la programmation des grilles informatiques 3
1.3 Un modèle de composant parallèle et distribué 4
1.4 Organisation du document 7

2 Modèles de programmation des grilles informatiques **9**
2.1 De l'évolution des modèles de programmation 10
2.1.1 Accumulation de connaissance en programmation . . 10
2.1.2 Des composants logiciels 11
2.1.3 Des composants logiciels et des simulations numériques distribuées 13
2.2 Un modèle d'entités parallèles distribués 14
2.2.1 Introduction 14
2.2.2 Un modèle de programmation parallèle distribué . . 15
2.2.3 Projections du modèle d'entité parallèle distribuée . 20
2.2.4 Discussion 24
2.3 Un modèle de composants avec partage de données 25
2.3.1 Des modèles de données 25
2.3.2 Un modèle de port orienté données 26
2.3.3 Discussion 29
2.4 Vers un modèle de composant logiciel dynamique 30
2.4.1 De la dynamicité 30
2.4.2 Un modèle de composant supportant le paradigme maître-travailleurs 30
2.4.3 Discussion 32
2.5 De la programmation par aspect pour des modèles de composants 33
2.5.1 De la programmation par aspects 33

2.5.2 Tissage dynamique de ports dans CCM 34
2.5.3 Discussion 36
2.6 Bilan et perspectives 37

3 Une plate-forme d'intégration d'exécutifs communicants 39
3.1 Introduction 39
3.2 Des communications dans les grilles informatiques 41
3.2.1 Des réseaux de communications dans les grilles informatiques 41
3.2.2 Des paradigmes de communications 43
3.2.3 Discussion 44
3.3 Un modèle de plate-forme d'intégration d'exécutifs 45
3.3.1 Un modèle d'abstraction multi-paradigme 46
3.3.2 Architecture d'une plate-forme multi-paradigme . . 47
3.3.3 Abstractions de communications multi-paradigmes . 49
3.3.4 Virtualisation des interfaces de communications . . . 53
3.3.5 Accès arbitré aux ressources 54
3.4 PadicoTM : une implémentation du modèle 55
3.4.1 Évaluation des performance 57
3.5 Bilan et perspectives 58

4 Déploiement automatique d'applications 61
4.1 Introduction 62
4.1.1 Services offerts par les grilles informatiques 63
4.1.2 Applications et grilles informatiques 63
4.2 Vers un déploiement automatique 64
4.2.1 Une architecture de déploiement automatique 64
4.2.2 Descriptions spécifiques des applications 66
4.2.3 GADe : une description générique d'applications . . 70
4.2.4 ADAGE : une implémentation de l'architecture 74
4.2.5 Discussion 75
4.3 Une description des réseaux pour le déploiement 76
4.3.1 Introduction 76
4.3.2 Description des ressources réseaux 76
4.3.3 Un modèle de description des topologies réseaux . . 77
4.3.4 Mise en œuvre 78
4.3.5 Discussion 79
4.4 Utilisation de Gamma pour l'exécution de workflow 80
4.4.1 Des workflow scientifiques 80
4.4.2 Un modèle d'exécution basé sur Gamma 81

4.4.3 Discussion 82
4.5 Bilan et perspectives 83

5 Conclusion et perspectives 85
5.1 Modèle de programmation 85
5.2 Support réseau 86
5.3 Déploiement 86
5.4 Perspectives 87

Bibliographie 90

Chapitre 1

Introduction

Sommaire

1.1 Les grilles informatiques 1
1.2 De la programmation des grilles informatiques 3
1.3 Un modèle de composant parallèle et distribué 4
1.4 Organisation du document 7

1.1 Les grilles informatiques

L'arrivée des réseaux longue distance haut débit dans les années 1990 a apporté une modification profonde dans la manière d'utiliser les ressources informatiques. Depuis lors, il est possible d'accéder efficacement à des ressources distantes. Dans le domaine du calcul scientifique, cela a permis dans un premier temps d'accéder à des supercalculateurs distants. Très rapidement, il est apparu qu'il était également possible d'utiliser *conjointement* plusieurs supercalculateurs pour résoudre un problème. Cette évolution technologique a permis de rendre opérationnelle les grilles de calcul telles que EGEE [81] en Europe, TeraGrid [86] aux États-Unis ou encore la grille Asie-Pacifique ApGrid [79].

La vision originelle des grilles informatiques [29] est inspirée d'une analogie avec le réseau électrique. L'abonnement à un fournisseur d'électricité permet à tout consommateur d'accéder à l'électricité sans se soucier où et comment elle est produite (barrage, centrale nucléaire, éolienne, etc.). Les grilles informatiques visent à offrir de la puissance de calcul sans que la manière dont cette puissance est produite soit pertinente pour les utilisateurs (machines parallèles, grappes de PC, etc.). Afin d'offrir toujours plus de

puissance de calcul, les grilles sont obtenues par l'agrégation de ressources disponibles dans plusieurs organisations. Elles sont ainsi constituées d'un ensemble hétérogène de ressources fournissant de la puissance de calcul ou plus généralement des services : service d'accès à des ressources, service de calcul spécifique, service de stockage, etc.

Toute ressource peut faire partie d'une grille à condition qu'elle soit mise à disposition via une interface d'accès standard. Cette grande flexibilité permet d'inclure au sein d'une même grille des machines avec des processeurs, des quantités de mémoire, des disques, des systèmes d'exploitation différents. Non seulement les machines peuvent être différentes mais également des réseaux très divers peuvent faire partie d'une grille comme Internet et des réseaux longue distance privés, mais aussi toutes sortes de réseaux locaux haute performance (Myrinet, Infiniband, Quadrics, etc.) En ce qui nous concerne, les grilles informatiques sont vues comme un ensemble de machines (supercalculateurs, grappes de PC, machines de visualisation, etc.) interconnectées par tout type de réseau. De manière générale, leur ressources sont organisées en plusieurs niveaux de hiérarchie.

En plus de caractéristiques concernant la diversité des ressources, les grilles informatiques présentent également des caractéristiques de large échelle, de volatilité des ressources, de partage de ressource et de sécurité. Un objectif des grilles est de fédérer le plus grand nombre de ressources. Ainsi, les problématiques relatives à la large échelle doivent être prises en compte aussi bien dans les logiciels de grilles que dans les applications. La large échelle ainsi que la disponibilité des ressources demandent une gestion de la volatilité : non seulement, il est peu probable que deux exécutions aient lieu sur le même ensemble de ressources mais il est fort probable que des ressources disparaissent ou apparaissent durant l'exécution d'une application. Cette caractéristique doit être traitée en conjonction avec le fait que les ressources sont partagées, c'est à dire que plusieurs utilisateurs et/ou applications demandent un accès à ces ressources. L'accès requis étant souvent exclusif, le système doit pouvoir décider à qui et comment attribuer les ressources. Enfin, les ressources des grilles appartiennent à des organisations qui se font une confiance limitée et/ou n'ont pas les mêmes exigences de sécurité. Par exemple, un particulier qui met son PC à disposition pour le décrypthon n'a pas le même niveau de sécurité qu'un centre de traitement de données médicales. Le problème de la sécurité – authentification, autorisation, délégation de droit, etc. – est crucial dans les grilles

En ce qui nous concerne, la problématique retenue est celle des modèles de programmation des grilles en tant qu'infrastructure distribuée et large

échelle. Nous supposons l'existence d'intergiciels de grilles qui prennent en compte les questions de volatilité des ressources, de partage de ressources et de sécurité. Les hypothèses retenues offrent néanmoins un sujet d'étude original. Les grilles posent des questions relatives à la programmation et à l'exécution d'applications dans un système distribué et parallèle en mettant en exergue des problèmes liés à l'acquisition dynamique de ressources. En ce sens, la problématique paraît différente de celle des systèmes distribués qui s'intéressent peu à l'exécution d'une seule application. Elle paraît plus proche des thématiques du calcul parallèle si ce n'est que le calcul parallèle fait généralement les hypothèses de connaître le type de machine et se préoccupe quasiment pas des ressources avec l'hypothèse d'un monde fermé et (généralement) stable.

1.2 De la programmation des grilles informatiques

Les grilles informatiques offrant un mélange d'architecture répartie et parallèle, des travaux ont eu lieu pour les programmer avec le paradigme distribué ou avec le paradigme parallèle. Au sein de chacun de ces paradigmes, l'existence de plusieurs modèles de programmation a conduit à la proposition d'autant de modèles de programmation dédiés. Le plus commun est sans conteste la vision d'une grille comme une ferme de services déjà installés. C'est notamment le paradigme utilisé par les applications paramétriques via des environnements comme SETI@Home [47], Boinc [78], XtremWeb [31] ainsi que les serveurs d'applications Netsolve [16], Diet [15] et Ninf [72]. Les serveurs d'applications se démarquent car ils intègrent des ordonnanceurs pour affecter le calcul à la meilleure instance du service demandé. La plupart du temps, le critère d'optimalité est la minimisation du temps d'attente pour le client, temps qui inclut le temps de calcul mais également le temps de transfert des données entre le client et le serveur.

Cette vision d'appel de service a naturellement été prolongée avec des environnements de workflow qui permettent d'enchaîner les calculs. D'une exécution centralisée des workflows – ou orchestration –, les environnements sont en train d'évoluer vers une exécution distribuée – ou chorégraphie – qui permet d'éliminer le goulot d'étranglement que représentait la centralisation.

Une deuxième approche a consisté à adapter les modèles de passage de messages des environnement parallèles aux grilles informatiques. Ainsi, de nombreux travaux ont adapté le standard MPI [32] aux grilles informatiques : MPICH-G2 [40], PACX-MPI [43], MagPIe [45], etc. Ces travaux se

sont surtout concentrés sur l'optimisation des performances des opérations collectives de MPI aux architectures réseaux multi-niveaux. MPICH-G2 a proposé des extensions à la norme MPI pour permettre aux applications de construire des communicateurs regroupant tous les nœuds d'une grappe. Des travaux similaires ont été entrepris pour les environnements fournissant une mémoire virtuellement partagée avec le développement de protocoles de cohérence adaptés [4].

L'adaptation a minima des modèles de programmation parallèle existants pour les grilles présente comme principale limite de n'exploiter que partiellement les grilles informatiques : en effet ces modèles gardent la vision d'une grille comme une architecture distribuée ou comme une architecture parallèle et non comme une architecture à la fois distribuée et parallèle. Ainsi, les modèles avec un paradigme réparti ne savent pas prendre en compte les codes parallèles ni les réseaux locaux haute performance. Les modèles avec un paradigme parallèle n'ont pas de notion d'interopérabilité, ont une vision très simpliste du déploiement de l'application, gèrent très mal voir pas du tout la volatilité des nœuds et surtout imposent implicitement d'organiser l'application en fonction de l'architecture sous-jacente pour obtenir de haute performance.

1.3 Un modèle de composant parallèle et distribué

Mon objectif est d'étudier et de proposer un modèle de programmation général, efficace et offrant un niveau d'abstraction élevé pour les grilles informatiques. Les grilles étant un mélange d'architecture répartie et parallèle, le modèle de programmation se doit de supporter ces deux architectures. Étant donné que les ressources ne sont connues que lors du lancement d'une application, il est vital de découpler le modèle de programmation des ressources. Ainsi, une même application devrait pouvoir s'exécuter sur des machines multi-cœur, des grappes de PC ou bien des grilles, quelque soit le réseau et sans recompilation. Le modèle de programmation doit donc permettre au système de choisir au mieux les ressources pour une instance de l'application en fonction de ses contraintes d'exécution.

Afin de motiver un tel modèle de programmation, nous avons choisi de nous intéresser aux applications de couplages de codes. Ce choix est motivé par le fait que ce type d'applications est particulièrement pertinent pour les grilles informatiques à plus d'un titre. Tout d'abord, les applications de couplages de codes requièrent toujours plus de puissance de calcul afin d'obtenir des simulations plus réalistes (diminution des pas de discrétisations

ou de temps, prise en compte de plus en plus de phénomènes, etc.) : les machines existantes comme les grappes de PC ne délivrent pas assez de puissance de calcul. Ensuite, elles offrent une grande variété de structures. Ainsi, certaines applications ont une structure statique qui peut se projeter directement sur les grilles en associant un code parallèle à une grappe de PC de la grille. D'autres applications de couplages sont plus dynamiques car contenant par exemple le paradigme maître-travailleur. De plus, les différents codes parallèles sont généralement écrits par des équipes différentes, dans des langages de programmation différents. Nous disposons ainsi de multiples sources d'hétérogénéité qu'il faut maîtriser. Enfin, il y a un réel intérêt à démontrer que ce type d'applications souvent jugé trop complexe et trop gourmand en ressources peut être simplement et efficacement mis en œuvre sur des grilles.

Afin de définir un modèle de programmation pour de telles applications, j'ai organisé mes activités de recherche autour des fonctionnalités importantes à supporter. La caractéristique fondamentale a été de disposer d'un modèle de programmation supportant l'hétérogénéité des codes et pouvant intégrer des codes d'origines diverses sous des contraintes de haute performance. En effet, les applications de couplage de codes relient des codes patrimoniaux très complexes. Il n'apparaît pas raisonnable d'imposer de les ré-écrire dans un nouveau modèle. Une deuxième caractéristique importante était de disposer d'un modèle de déploiement de l'application, c'est à dire d'un modèle permettant de relier une description de l'application à des ressources ; la description de l'application ne devant contenir idéalement aucune référence à des ressources particulières. En effet, les grilles sont basées sur le concept de l'affectation temporaire de ressources à une application via des services spécifiques de courtage de ressources, de transfert de fichiers et de lancement à distance de processus. C'est pourquoi, j'ai systématiquement chercher à distinguer clairement ce qui concerne la description des applications de ce qui concerne leurs exécutions effectives.

L'état de l'art sur les modèles de programmation montre que les modèles de composants logiciels offrent d'excellentes propriétés pour supporter ces caractéristiques : leur définition [71] est basée sur la notion de composition et d'unité de déploiement. Cependant, quelques fonctionnalités manquaient aux modèles de composants et à leurs implémentations pour supporter les applications de couplage de code. Il s'agit de l'encapsulation efficace de codes parallèles, du support de plusieurs intergiciels communicant et du déploiement de ces codes sur des infrastructures de grilles. Ces trois thématiques ont constitué les trois axes structurant de mes activités de

recherche.

Le premier axe a été de proposer un modèle de composants logiciels encapsulant efficacement des codes parallèles. L'objectif était de définir un modèle de composants parallèles supportant très efficacement les communications entre deux composants parallèles ; le parallélisme du composant étant traité comme une propriété non fonctionnelle, elle n'est visible que pour le développeur du composant. Ce traitement du parallélisme nous apparaît important afin de découpler un maximum la spécification du calcul de la manière dont il est réalisé. À partir de ce modèle de composant parallèle, la stratégie a consisté à lever une à une les contraintes que nous nous sommes fixé afin de tendre vers un modèle permettant de supporter simplement et efficacement tout type d'applications sans référence à des notions spécifiques à l'infrastructure d'exécution. Premièrement, nous avons étendu les types de relation entre composants afin de supporter également le partage de donnée. C'est un type de relation qui peut simplifier et améliorer la performance des applications car il évite de diffuser ou de distribuer dans le code fonctionnel les données. Deuxièmement, nous avons commencé à travailler sur la dimension temporelle de la programmation. En effet, les modèles de composant classiques permettent de décrire l'architecture spatiale des composants mais ne prennent pas en compte la dimension temporelle. Ce sous-objectif me paraît particulièrement important car il permet de faire le lien avec les travaux sur les workflows qui ne prennent en compte que la dimension temporelle. Troisièment, nous nous sommes intéressé aux autres dimensions de la programmation [44] et en particulier au modèle de programmation par aspect.

Le deuxième axe de recherche a concerné le support exécutif afin de valider nos propositions avec des mises en œuvre. En effet, le modèle de composants logiciels parallèles et les grilles informatiques nous ont amené à travailler sur le découplage des interfaces de communications de leur implémentation effective. Le problème a ses origines dans l'utilisation d'interfaces spécifiques pour l'exploitation efficace des réseaux haute performance tels que Myrinet, Quadrics, etc. A priori, il s'agit d'un problème de virtualisation d'interfaces réseaux. Cependant, le problème est plus complexe car certaines interfaces sont orientées parallélisme et d'autres sont orientés réparties. La contrainte de haute performance que nous nous sommes fixé demande de réaliser ce découplage sans nuire au performance tout en supportant les nombreux intergiciels communicants utilisés par les applications comme par exemple MPI, CORBA, Web Services, etc.

Le troisième axe de recherche a été de maîtriser le processus de déploiement d'une application sur une grille. Cette étape fait le lien entre la

description d'une application et une grille informatique. Les objectifs sont d'automatiser autant que faire se peut ce processus sans être lié à un modèle de programmation particulier. En effet, étant donné que notre proposition de modèle de programmation pour les grilles informatiques est appelée à s'enrichir progressivement de nouvelles fonctionnalités, il apparaît important d'essayer de ne pas lier les deux. Une autre motivation est que les applications, surtout avec une vision composants, n'ont pas de borne sur le nombre d'intergiciels communicants qu'elles peuvent utiliser (MPI, CORBA, JAVA-RMI, WebServices, etc.). C'est pourquoi nous visons dès le départ tout type d'application.

En résumé, mes activités de recherche décrites dans ce manuscrit visent à étudier un modèle de programmation pour les applications scientifiques pour les grilles informatiques :

- à base de composants logiciels, car ils mettent en avant l'opération de composition ;
- intégrant les paradigmes parallèle et répartie pour prendre en compte la double nature des grilles informatiques ;
- supportant plusieurs intergiciels de communication pour construire des applications en réutilisant des codes existants ;
- sous des contraintes de haute performance ;
- permettant un déploiement automatisé.

1.4 Organisation du document

La suite de ce document contient quatre chapitres. Le chapitre 2 est consacré à nos travaux relatifs aux modèles de composants logiciels. Le chapitre 3 présente nos travaux concernant le support réseau nécessaire à la réalisation de notre modèle de programmation. Le chapitre 4 est dédié aux travaux réalisés autour de la problématique du déploiement automatique d'applications sur les grilles informatiques. Enfin, le chapitre 5 dresse un bilan des travaux réalisés et discutent des perspectives de ces travaux.

Chapitre 2

Modèles de programmation des grilles informatiques

Sommaire

2.1 De l'évolution des modèles de programmation 10
2.1.1 Accumulation de connaissance en programmation 10
2.1.2 Des composants logiciels 11
2.1.3 Des composants logiciels et des simulations numériques distribuées 13
2.2 Un modèle d'entités parallèles distribués 14
2.2.1 Introduction 14
2.2.2 Un modèle de programmation parallèle distribué 15
2.2.3 Projections du modèle d'entité parallèle distribuée 20
2.2.4 Discussion 24
2.3 Un modèle de composants avec partage de données . . 25
2.3.1 Des modèles de données 25
2.3.2 Un modèle de port orienté données 26
2.3.3 Discussion 29
2.4 Vers un modèle de composant logiciel dynamique . . . 30
2.4.1 De la dynamicité 30
2.4.2 Un modèle de composant supportant le paradigme maître-travailleurs 30
2.4.3 Discussion 32
2.5 De la programmation par aspect pour des modèles de composants 33
2.5.1 De la programmation par aspects 33
2.5.2 Tissage dynamique de ports dans CCM 34

2.5.3 Discussion 36
2.6 Bilan et perspectives 37

2.1 De l'évolution des modèles de programmation

2.1.1 Accumulation de connaissance en programmation

Depuis le début de l'informatique, la recherche en programmation a pour but de faciliter l'expression des programmes et non de permettre l'expression de nouveaux programmes. En effet, il est bien connu que tout calcul est exprimable via une machine de Turing. Cependant, les applications ne sont pas écrites pour cette machine car il est très fastidieux de programmer en remplissant des tables de transitions d'état.

L'expérience acquise au fur et à mesure du développement de programme a mis en évidence des bonnes pratiques de programmation qui ont été prises en compte par les modèles de programmation : c'est ainsi que de la connaissance est accumulée et partagée. Nous pouvons distinguer quatre formes principales de partage de la connaissance [71] :

- sous forme descriptive, ce sont les patrons de conceptions, ensembles de problèmes récurrents et de leur solution ;
- sous forme binaire, ce sont les bibliothèques, solutions accessible via une interface et directement réutilisables ;
- sous forme d'architecture partielle, ce sont les canevas de programmation qui intègrent potentiellement plusieurs patrons de conceptions et bibliothèques ;
- sous forme de langages de programmations, mettant en œuvre, souvent de façon contraignante, de bonnes pratiques comme discuté ci-après.

La forme la plus perceptible de l'accumulation de connaissances réalisée au travers des différents modèles de langages est l'évolution des modèles de programmation. Sans prétendre être totalement exhaustif, nous pouvant observer quatre étapes principales allant des langages assembleur aux langages objets. Les langages assembleurs ont été les premiers langages de programmation. Ils sont très proches des machines, et sont majoritairement des langages impératifs avec des groupes d'opérations d'accès à la mémoire (load/store), de logique et de calcul entier (ALU) et de calcul flottant. Ensuite, les langages structurés sont apparus. Ils matérialisent la bonne pratique du patron de conception de la répétition via les trois variantes (`for`, `while`, et `do while`). Puis, ce fut le tour des langages procéduraux. Ils ré-

pondent au problème de la répétition d'une séquence d'instruction en de multiples lieux d'un programme en regroupant ces instructions dans une procédure (avec les variantes co-routine et fonction). Ainsi, de nouvelles *instructions* peuvent être créées (via les procédures), mécanisme permettant d'aller vers de plus en plus abstraction. Enfin, face à la difficulté de contrôler les accès à une variable, les langages objets matérialisent la bonne pratique de l'encapsulation des données et du code les modifiant. Ils apportent également la notion d'héritage comme mécanisme de structuration des programmes.

D'autres bonnes pratiques ont été trouvées et mises en œuvre dans certains langages. Citons par exemple le typage qui permet de vérifier la cohérence de type, la généricité qui permet d'abstraire le type de données sur lequel s'applique des instructions et enfin le mécanisme d'exception, autre mécanisme visant à éviter l'utilisation des goto tout en gardant un flot de contrôle imbriqué.

La tendance de l'évolution des langages de programmation est donc vers de plus en plus d'abstraction afin d'une gérer des codes de plus en plus complexes.

2.1.2 Des composants logiciels

Les modèles de composants logiciels ont connu leur vrai démarrage à la suite de la défaillance des modèles objets pour de grandes applications séquentielles [71]. Ainsi, bien que les composants logiciels commencent à arriver à maturité depuis quelques années [10], leur intuition n'est pas toute récente [56].

Les modèles de composants logiciels mettent en avant la programmation par *assemblage* plutôt que par *développement*. Les avantages attendus sont de permettre de focaliser l'expertise sur un domaine, d'améliorer la qualité des logiciels, et de diminuer le temps de développement grâce à la réutilisation de codes existant. Leur objectifs sont très proches de ceux des modèles objets, mais l'espoir est qu'ils permettront réellement de gérer de grands codes. Ainsi, les modèles de composants privilégient la bonne pratique de l'utilisation de la relation de délégation au lieu de celle de l'héritage. De plus, ils visent le cycle complet du logiciel : définition, implémentation, packaging, déploiement et exécution.

Il existe nombreuses définitions sur les composants logiciels, celle de Szyperski [71] semble assez couramment admise : *Un composant logiciel est une unité de composition qui spécifie contractuellement ses interfaces et qui définit explicitement ses dépendances de contexte. Un composant logiciel peut être déployé*

indépendamment et est sujet à composition par une tierce partie.

Ainsi, les modèles de composants se focalisent sur la composabilité entre composants qui est réalisée via des ports compatibles et non sur la sémantique des ports. La sémantique des ports, c'est à dire le type d'opération que réalise les ports, est orthogonale à la notion de composant. Il existe des modèles de composants ayant des types de ports divers. Le type de port le plus répandu est l'interface, c'est à dire un ensemble d'opérations, dans un mode RPC ou RMI. Le passage de message est également présent soit en asynchrone (événement) soit en synchrone [54]. Enfin, il existe également le type de port de flots de données [1].

Assemblage de composants

Les modèles de composants logiciels permettent la construction d'application par assemblage de composant. Cette composition peut être statique ou dynamique. Les langages de description d'architecture (ADL) sont particulièrement bien adaptés pour décrire les assemblages. Ainsi, le modèle de composant de CORBA [17] définit un ADL qui représente l'état initial de l'application. Il en va de même pour l'ADL de Fractal [13]. Par contre, le Common Component Architecture (CCA) ne définit pas d'ADL [12]. Tout se passe à l'exécution où un composant peut décider de créer des ports et de se connecter à qui il veut avec ou sans l'aide de l'environnement.

Enfin, la plupart des modèles de composants sont des modèles plats, c'est à dire qu'un assemblage de composant n'est pas un composant. Cependant, il existe quelques modèles de composants hiérarchiques, comme Fractal [13] ou Darwin [54], qui permettent de définir un composant comme un assemblage d'autres composants. On parle alors de composite. Des problèmes restent ouverts quant à la gestion des propriétés non fonctionnelles comme la persistance ou les transactions dans des modèles hiérarchiques.

Discussion

Les modèles de composants représentent l'état de l'art concernant l'intégration de bonnes pratique au sein d'un modèle de structuration des programmes. Il est à noter que tout comme les modèles objets ne remettent pas en cause la programmation structurée et procédurale, les modèles de composants ne rendent pas obsolète la programmation par objets. En effet, un composant doit être implémenté et les langages de programmation orienté objet offrent actuellement le meilleur modèle.

Les modèles de composant sont nés pour faire face à la complexité

des applications séquentielles où les modèles objets ont montré leurs limites. L'extension des modèles de composant au réparti fut assez immédiate puisque les modèles de composants séquentiels utilisent la notion d'interface d'objet pour assembler deux composants. Par exemple, le modèle Fractal a été porté en réparti sur ProActive [11] en se servant du RMI Java [33]. Un autre exemple est le modèle de composant CORBA [17] qui est basé sur le modèle d'objet distribué CORBA [19].

Nous nous sommes donc intéressé à l'utilisation de modèles de composants logiciels pour la programmation d'applications de simulation numérique car les composants logiciels promettent un modèle pouvant gérer la complexité sans cesse croissante des simulations numériques.

2.1.3 Des composants logiciels et des simulations numériques distribuées

Notre objectif est d'étudier comment permettre la programmation de simulation numérique distribuée sur des grilles informatiques. Comme nous l'avons vu précédemment, les grilles informatiques semblent bien adaptées aux modèles de programmation où l'architecture matérielle est cachée. Les simulations numériques sont des applications très complexes et requérant de haute performance. Ainsi, non seulement nous recherchons des fonctionnalités de haut niveau mais elles doivent pouvoir être mises en œuvre très efficacement.

Nous nous sommes focalisé sur les simulations à base de couplage de code. Elles représentent un type d'application capable de tirer parti d'une grille car elles sont d'une très grande complexité de mise en œuvre, appartiennent à un domaine en plein essor de la simulation numérique, et la plupart des machines parallèles ne semblent pas assez puissantes pour supporter leur exécution.

Le premier travail a concerné l'extension des modèles de composants logiciels pour le support du parallélisme. En effet, les simulations numériques utilisent des codes parallèles qui faut encapsuler correctement dans des composants. Ce travail, présenté en section 2.2, a permis de comprendre les relations entre la programmation répartie, apportée par les modèles de composants considérés, et la programmation parallèle utilisée au sein des codes de simulations.

L'étape suivante est d'appréhender ce qui exprimable au travers d'un port. Alors que le travail précédent a apporté la notion de port parallèle, la section 2.3 s'intéresse à la notion de port orienté donnée. Ainsi, les deux grands types de communications – explicite et implicite – seront supportés

au niveau des modèles de composants.

La section 2.4 aborde la compréhension des relations entre modèles de composants et la dynamicité. Par dynamicité, nous entendons les possibilités de création/destruction de composants ainsi que le support de bonnes pratiques de programmation. Nous avons commencé par nous intéresser au patron de conception maître-travailleur. Non seulement, ce travail touche à la définition des ports mais également aux notions exprimables dans les ADL.

Enfin, nous nous sommes ponctuellement intéressé aux liens entre les modèles de composants logiciels et la programmation par aspects. La section 2.5 présente le travail que nous avons mené concernant l'ajout dynamique de ports à un composant.

2.2 Un modèle d'entités parallèles distribués

2.2.1 Introduction

Notre objectif est de fournir un modèle de programmation fournissant un environnement adapté pour les applications de couplage de codes. Une analyse des applications rencontrées dans les ACI GRID RMI et Hydrogrid a permis d'exprimer clairement les besoins vis à vis du modèle de programmation. Ce dernier doit supporter aussi bien les environnements distribués, avec notamment des connexions dynamiques, que les environnement parallèles. En effet, les applications sont classiquement constituées d'un ensemble de programmes, potentiellement distribués. Ces programmes peuvent être parallèles. Le besoin de connexions dynamiques vient par exemple des fonctionnalités de visualisation et de pilotage d'applications [20].

Les programmes pouvant être parallèles ou séquentiels, les communications peuvent être de types parallèle-parallèle, parallèle-séquentiel, ou séquentiel-parallèle. Ainsi, bien que le cas critique pour les performances soit le type parallèle-parallèle, il faut également supporter les deux autres cas.

Enfin, les exceptions ayant démontrées leur utilité pour la structuration d'applications et étant présentes dans la plupart des modèles distribués, nous nous devons de les supporter lors de communications entre deux codes parallèles.

Analyse de l'existant

Les applications de couplage de code expriment en premier lieu un besoin de modèle distribué. Notre approche a donc été de partir d'un modèle distribué et de l'étendre avec des notions de parallélisme. Cette approche a déjà été utilisée pour résoudre ce problème à partir de modèles d'objets distribués. Ainsi, ParDis [41] et PaCO [65] visent à définir le concept d'objets parallèles distribués. Cependant, ils présentent les limites de modifier la norme sur laquelle ils reposent (CORBA), de fixer dans le modèle les types de distributions de données supportées, de ne pas permettre d'ordonnancement des communications et enfin de ne pas offrir de modèle de fonctionnement. En effet, il s'agissait de travaux exploratoires.

Quelques travaux concernent les modèles de composants et les applications de couplage de code. Le plus ambitieux est sûrement le Common Component Architecture (CCA) [12]. Il s'agit d'une version simplifiée de CCM pour le calcul haute performance. Les interfaces d'un composant sont décrites en Scientific IDL (SIDL) [46], un IDL étendu pour supporter nativement les types FORTRAN tels que les complexes et les tableaux multidimensionnels. La plupart des implémentations de CCA se restreignent à des machines parallèles et ne supporte par vraiment de communications parallèle-parallèle. En effet, les communications entre composants parallèles sont des appels de fonctions intra-processus qui supposent que le parallélisme des deux composants est le même (M=N) et qui, de plus, ne prend pas en compte la redistribution de données (redistribution identité). C'est le cas des implémentations distribuées (XCAT [48], SciRun2 [77], Legion-CCA [50]) qui se basent sur les Web Services [18] ou JAVA RMI. Seule DCA, une implémentation de CCA sur MPI, supportent les communications MxN. Cependant, elle modifie le langage SIDL et ne supporte qu'une seule forme de redistribution de données.

Un autre travail est mené autour de ProActive [11]. D'un modèle objet distribué actif supportant les communications de groupe, c'est dire séquentiel-parallèle, le projet implémente maintenant le modèle de composant hiérarchique Fractal au-dessus de ProActive. Cependant, cette évolution n'a pas eu d'impact sur le type de communication supporté mais le type des entité. Il s'agit maintenant d'un modèle de composant distribué actif supportant des communications de groupe.

2.2.2 Un modèle de programmation parallèle distribué

Notre démarche consiste à travailler à un niveau abstrait. Ainsi, notre solution, indépendante des technologies distribuées, peut par la suite

être projetée vers des technologies distribuées. Nous avons à peu près la même démarche que l'OMG avec son initiative *Model Driven Architecture* (MDA) [59]. Cependant, par manque de temps, nous n'avons pas défini de spécifications précises de l'opération de projection.

Notre démarche est motivée en particulier par deux observations. D'une part, le support du parallélisme semble relativement orthogonal à la nature de la technologie distribué (objet, composant, service, procédure, ...). D'autre part, les composants n'imposent de type de communication qu'au travers de leur type de port. Même si ces ports implémentent principalement le concept de RMI, d'autres formes de communication peuvent également être utilisées comme nous l'illustrons à la section 2.3.

Ainsi, nous avons défini un modèle de programmation parallèle et distribué abstrait. Ce modèle définit les concepts et les communications qui régissent des entités qui sont en premier lieu distribuées mais qui peuvent en plus être parallèles. Les détails de ce modèle sont présentés dans la thèse d'André Ribes [66] que j'ai co-encadré avec Thierry Priol. Le modèle est basé sur le concept d'entité parallèle distribuée, sur des modèles définissant les communications, la gestion des données distribuées et celles des exceptions entre de telles entités.

Entité parallèle distribuée

Un écueil fréquemment rencontré en informatique est la difficulté de séparer le modèle de l'implémentation. Ainsi, de nombreux aspects de la solution sont souvent implicites. En définissant une entité parallèle distribuée, nous avons cherché à rendre explicite tous les concepts et opérations s'y référant.

Le concept de base de notre modèle est donc une entité parallèle distribuée. Il s'agit d'un mélange entre une entité parallèle – programme s'exécutant sur plusieurs flots d'exécution, potentiellement sur plusieurs nœuds, disposant d'un état global et de mécanisme de communication entre les différents instances – et une entité distribuée – programme disposant d'une référence permettant de s'y connecter et pouvant communiquer avec d'autres entités distribuées. Afin de prendre en compte les propriétés non-fonctionnelles, la définition d'une entité parallèle distribuée inclut la notion de gestionnaire, entité dédiée au support des ces propriétés. Nous définission une *entité parallèle distribuée* comme une entité composée d'un ensemble d'entités distribuées réalisant les différents rôles de l'entité (fonctionnelles et non-fonctionnelles) et partageant un état global ainsi qu'un état spécifique aux entités réalisant la fonction parallèle.

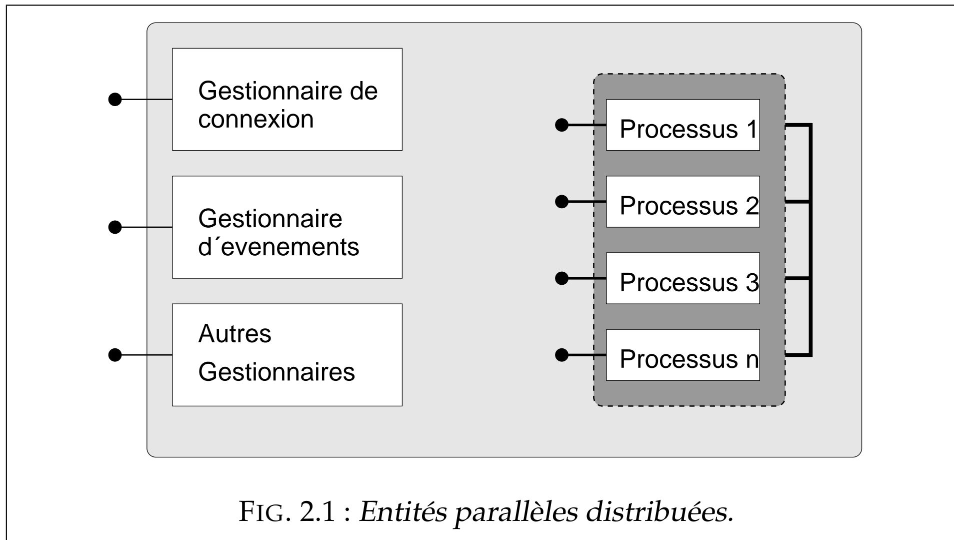

FIG. 2.1 : *Entités parallèles distribuées.*

La figure 2.1 présente une illustration d'une entité parallèle distribuée. Deux remarques importantes sont à noter. D'une part, une telle entité dispose de nombreuses références à des entités ayant des rôles fonctionnelles et non-fonctionnelles. D'autre part, le nombre et le type d'entité ayant des rôles non-fonctionnels sont non bornés. Ces deux points sont spécialisés dans le modèle de communication.

Modèle de communication

Le modèle de communication définit les opérations de connexions, de communications et de déconnexions entre entités parallèles distribuées. Comme nous allons le décrire, la connexion se fait entre des gestionnaires spécialisés. Afin, de permettre des interactions potentiellement efficaces et extensibles, le modèle autorise des communications directes entre les entités parallèles de deux entités parallèles distribuées. En général, les gestionnaires ne sont utilisés que durant les phases de connexions et déconnexions.

Afin de manipuler une entité de manière globale, il est nécessaire de pouvoir associer une référence à une entité parallèle distribuée. Notre choix a été de déléguer ce rôle à un gestionnaire particulier, l'`InputConnectionManager`. Ainsi, la référence de ce gestionnaire représente la référence de toute l'entité. Ce gestionnaire traite les demandes de connexions entrantes. De manière symétrique, un autre gestionnaire, l'`OutputConnectionManager` s'occupe des demandes de connexions sortantes. Ces deux gestionnaires sont appelés des

`ConnectionManagers`.

La connexion entre deux entités parallèles distribuées se passent entre deux gestionnaires, un `OutputConnectionManager` et un `InputConnectionManager`. La connexion consiste en l'échange des références des entités parallèles des deux entités. Les `ConnectionManagers` obtiennent et stockent ces références via l'état global de l'entité. Cet état a été au préalable correctement initialisé. Ainsi, après une connexion, toute entité membre d'une entité parallèle distribuée peut connaître la référence de toute entité faisant partie de l'entité distante. C'est cette propriété avec la possibilité d'avoir des liens directs qui est utilisée pour réaliser des communications de types MxN : nous retrouvons ainsi le modèle général de la problématique de la redistribution qui considère des ensembles de nœuds source et destination.

Modèle de gestion de données distribuées

Afin de permettre des communications efficaces entre codes parallèles, il est primordial de prendre en compte la redistribution de données. En effet, que l'opération de transfert soit un passage de message ou un appel de procédure ou de méthode, une donnée distribuée selon une distribution D1 sur l'entité parallèle source doit se retrouver sur l'entité parallèle destination selon une distribution D2.

Ce problème a été beaucoup étudié dans des environnements parallèles tel que HPF [26], Scalapack [85] ou encore les bibliothèques de redistribution comme par exemple Chaos [22]. Notre objectif n'est pas de définir une nouvelle distribution mais plutôt de chercher à intégrer les connaissances accumulées dans notre modèle. En particulier, nous visons à supporter n'importe quelle redistribution contrairement à ce que permettait HPF, ParDis ou PaCO. Ainsi, nous nous sommes focalisé sur les types des intervenants ainsi que sur leurs relations.

Notre analyse nous a conduit à identifier trois intervenants : l'entité parallèle chargée de réaliser les communications, un intervenant chargé de calculer les communications en fonction des redistributions et un intervenant chargé de calculer un ordonnancement des communications.

Cette claire séparation des rôles conduit à définir des interface entre l'entité parallèles et les deux autres intervenants. Ces interfaces étant génériques, elles ne restreignent ni le type des données ni les redistributions et ordonnancement possibles. Les avantages attendus sont d'implémenter ces intervenants sous forme de composants logiciels et donc de faciliter l'utilisation de fonctionnalités avancées. Les autres interfaces, comme par

exemple l'association d'une distribution à une donnée, ne sont pas définies car elles ne sont pas du ressort du modèle d'entité parallèle distribuée. Un court exemple est présenté en section 2.2.3.

Modèle de gestion d'exceptions parallèle

Les exceptions sont devenues essentielles pour une bonne programmation car elles offrent un modèle de haut niveau pour modéliser et réagir à des évènements exceptionnels ou à des erreurs. C'est pourquoi nous avons proposé une extension du modèle d'exception aux communications entre deux entités parallèles, appelées respectivement émetteur et récepteur ci-après. En effet, de manière un peu surprenante, le sujet ne semble pas avoir été traité, bien que quelques modèles existent pour des langages parallèles [35, 76, 36].

Nous avons étendu la notion d'exception à une communication parallèle-parallèle avec la définition d'une exception parallèle – ensemble d'exceptions levées par un ou plusieurs éléments d'une entité parallèle réceptrice. Un cas particulier – mais important – est l'exception SPMD qui est une exception parallèle où *tous* les éléments de l'entité parallèle réceptrice ont levé de manière *cohérente* la même exception.

Notre modèle de gestion d'exception pour des entités parallèles est constitué de trois parties : les types d'exception, le niveau de support du récepteur et de l'émetteur et enfin des règles qui permettent de déterminer comment une exception levée par un récepteur est reçue par l'émetteur.

Le modèle définit trois types d'exception. Le premier type est l'exception simple qui correspond à la levée d'une et d'une seule exception. Le second type est l'exception agrégée qui est une exception parallèle. Enfin, le troisième type est l'exception complexe. Il correspond à une exception agrégée pouvant contenir tout ou partie des données qui auraient due être transmises lors de la communication.

Les entités émettrice et réceptrice doivent indiquer le niveau d'exception qu'elles supportent. Toute entité supporte au minimum les exceptions simples (niveau 1). Le support d'exceptions agrégées (niveau 2) implique le support d'exceptions simples et celui d'exceptions complexes (niveau 3) celui d'exceptions agrégées.

Un cas particulier d'exceptions simples est l'exception SPMD. En effet, une exception SPMD est une exception standard mais avec potentiellement des données distribuées. Pour une entité parallèle émettrice, elle est reçue comme une exception simple. Coté récepteur, elle demande juste que *tous* les éléments de l'entité parallèle aient levé cette exception de manière cohé-

Niveau de supporté par l'émetteur	Type d'exception levée par le récepteur		
	Simple e	Sgrégée $\{ e_1, ..., e_n\}$	Complexe $\{\{ e_1, ..., e_n \},\{d_1, ... d_m\}\}$
Exception simple	e	e1	e1
Exception agrégé	e	$\{ e_1, ..., e_n\}$	$\{ e_1, ..., e_n\}$
Exception complexe	e	$\{ e_1, ..., e_n\}$	$\{\{ e_1, ..., e_n \},\{d_1, ... d_m\}\}$

Légende : e_i exception numéro i, d_i données distribuées numéro i.

TAB. 2.1 : *Règles de transformation d'exception.*

rente et que l'exception ait été déclarée de type SPMD.

Enfin, les règles de transformation d'exceptions définissent ce que reçoit une entité parallèle émettrice en fonction de ce qui s'est passé chez l'entité réceptrice. Le tableau 2.1 résume ces règles. Les exceptions de type agrégées doivent avoir un élément privilégié afin de pouvoir choisir une exception dans le cas d'émetteur ne supportant que des exceptions simples.

2.2.3 Projections du modèle d'entité parallèle distribuée

Nous avons projeté et implémenté le modèle d'entité parallèle distribuée sur deux technologies distribuées : les objets et les composants CORBA. Ces projections ont été réalisées manuellement alors que le modèle abstrait a été pensé pour pouvoir être projeté automatiquement.

PaCO++

PaCO++ [64] est une extension parallèle portable des objets CORBA. Successeur de PaCO [65], il est obtenu par projection du modèle abstrait d'entité parallèle distribuée sur les objets CORBA. PaCO++ a été développé au-dessus de la norme CORBA ; en ce sens il est portable sur toute implémentation de CORBA écrite en C++. PaCO++ ne définit que le fonctionnement des objets parallèles. Il repose sur des plugins sous forme de bibliothèques pour la gestion des données distribuées et l'ordonnancement des communications.

À ce jour, quatre plugins de redistribution et deux plugins d'ordonnancement sont disponibles. Alors que nous avons développé les plugins de redistribution *Identité*, *Redistribution bloc-cyclique 1D* et *Gabro* (une opération de collecte suivie d'une opération de diffusion sélective), le plugin *RedSym* a été développé par Aurélien Esnard du projet Scalapplix (LaBRI) [25]. De même nous avons développé le plugin d'ordonnancement

```
IDL:                                    XML:
typedef sequence<float> lseq;           Interface: Serveur
interface serveur {                      Operation: calcul
   void calcul(in lseq d);                Paramètre: d
};                                         distribution: bc(16)

C++:
class serveur_impl :: public PaCO_serveur {
public:
   ...
   void calcul(const PaCO::lseq& seq) {
     ...
     MPI_barrier(...);
     ...
   }
};
```

FIG. 2.2 : *Exemple de déclaration et d'implémentation d'un objet CORBA parallèle. L'argument de l'opération* `calcul` *est attendu avec une distribution bloc-cyclique(16). L'implémentation de l'objet se fait de manière presque habituelle si ce n'est que la classe d'héritage est spécifiée par PaCO++.*

direct (adapté pour les réseaux des grappes) et Frédéric Wagner du projet Algorille (LORIA) a développé un ordonnancement adapté aux réseaux longue distance [37]. Les deux plugins externes n'ont pas été développés spécifiquement pour PaCO++. Cependant, leur intégration dans PaCO++ a été possible dans le cadre de l'ARC RedGrid car les interfaces de PaCO++ étaient clairement définies.

Notre choix pour PaCO++ a été de considérer que le parallélisme ne fait pas partie de l'interface de l'objet. C'est une propriété non-fonctionnelle. Ainsi, lors d'un appel sur un objet, l'appelant n'a pas à savoir si l'objet distant est séquentiel ou parallèle. De même, un objet appelé ne sait pas si l'appelant est séquentiel ou parallèle.

Les figures 2.2 et 2.3 présentent un exemple de déclaration d'objet parallèle ainsi qu'un programme client réalisant un appel. La description de la distribution des données ne dépend que du plugin de redistribution et pas de PaCO++.

Le mécanisme de connexion du modèle abstrait est mis à profit pour transférer automatiquement les informations concernant le parallélisme.

```
// Initialisation des données
float* data = ...

// dislib est une instance du plugin de redistribution
// block-cyclic 1D associée à la donnée data (code
// d'initialisation omis).
// Configuration des paramètres de la distribution source
dislib->setBlocSize(32);
dislib->setVectorSize(2048);

// Invocation parallèle
serveur->calcul(data);
```

FIG. 2.3 : *Exemple de code client. La distribution étant différente de celle du serveur, une redistribution a lieu quel que soit le degré de parallélisme du client et du serveur si au moins un des deux est au moins égal à 2.*

Par exemple, c'est l'`OutputConnectionManager` qui détermine si la référence de l'objet qu'il obtient est celle d'un objet séquentiel ou d'un objet parallèle. Dans le second cas, sachant que c'est la référence d'un `InputConnectionManager`, il entame le protocole d'échange d'information concernant les références internes ainsi que des informations concernant les distributions des paramètres des opérations via une interface spécifique à PaCO++.

Les évaluations de PaCO++ ont confirmé que les objectifs de performance ont été atteints [66]. D'une part le débit agrégé est bien proportionnel au nombre de nœuds utilisés. D'autre part, la latence d'une invocation est celle d'un appel CORBA plus le coût d'une barrière. En effet, une barrière est nécessaire côté appelant pour garantir le synchronisme des appels conformément aux spécifications CORBA. En dernier lieu, l'utilisation de l'ordonnancement pour les réseaux longue distance permet d'éviter une perte de performance due à la saturation du réseau et/ou commutateur sans devoir modifier les codes.

GridCCM

De manière similaire à PaCO++ et aux objets CORBA, GridCCM [63] est une extension parallèle des composants CORBA. Il est basé sur la projection du modèle d'entité parallèle distribuée sur des composants CORBA.

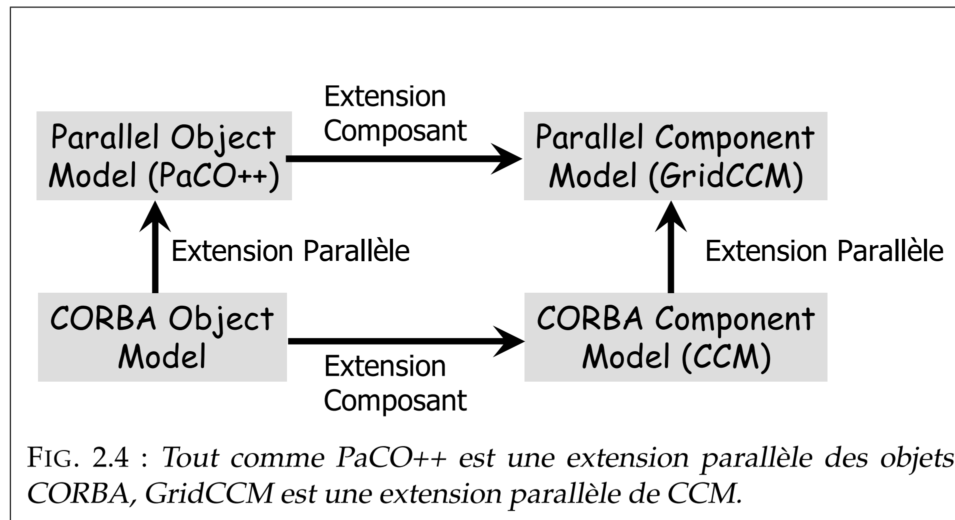

FIG. 2.4 : *Tout comme PaCO++ est une extension parallèle des objets CORBA, GridCCM est une extension parallèle de CCM.*

Ainsi, un composant parallèle est une collection de composants CCM, certains ayant les rôles de `ConnectionManagers` et les autres implémentant la partie fonctionnelle. Actuellement, nous avons choisi d'avoir le même degré de parallélisme pour tous les ports d'un composant parallèle. Cependant, cette limite arbitraire devrait être levée à l'avenir.

Le modèle de composant CORBA reposant sur un modèle objet à l'exécution, nous avons choisi de baser GridCCM sur PaCO++. La figure 2.4 montre les relations entre ces modèles. Ainsi, GridCCM utilise directement PaCO++ pour la gestion des données distribuées et de l'ordonnancement des communications.

GridCCM maintient la vision de garder le parallélisme comme une possibilité d'implémentation pour un type de composant. Ainsi, un composant peut avoir une implémentation séquentielle et une autre implémentation parallèle. Idéalement, le langage CIDL [17] de CCM devrait être étendu pour exprimer les détails d'implémentation d'un composant parallèle. En attendant, nous avons gardé le même principe que pour PaCO++, à savoir l'utilisation d'un fichier XML auxiliaire pour spécifier le parallélisme. Ainsi, à partir d'un fichier IDL3 et d'un fichier XML, GridCCM génère les souches et les squelettes relatifs aux composants parallèles. Il génère également un fichier IDL2 et un fichier XML pour PaCO++.

La chaîne de compilation de GridCCM étant encore en cours de développement, deux études de faisabilité ont été réalisées concernant GridCCM. La première a été faire à partir OpenCCM, une implémentation JAVA [55], alors que la seconde fut réalisée à partir de MicoCCM, une im-

plémentation C++ [83]. Ces études ont démontré que le concept de composant parallèle était réalisable et que de bonnes performances pouvaient être atteintes. En effet, comme il n'existe pas encore de très bonne implémentation de CCM, contrairement aux objets CORBA, et que GridCCM repose sur PaCO++, nous n'avons pas mis une grande priorité à la finalisation de la chaîne de compilation de GridCCM.

Évaluation dans des applications

PaCO++ a été utilisé dans deux applications. La première est une application industrielle fourni par EADS dans le cadre de l'ACI GRID RMI. Il s'agit d'un couplage de deux simulations électromagnétiques. PaCO++ est utilisé pour encapsuler le transfert de deux vecteur via une redistribution bloc-cyclique.

La seconde application vient de l'ACI GRID HydroGrid et concerne le couplage d'une simulation de transport avec une simulation d'écoulement. Cette simulation, ainsi que l'utilisation de PaCO++, n'a pas été réalisée par nos soins mais par le projet Sage (IRISA). Ainsi, nous avons pu évaluer la pertinence et la difficulté d'utilisation de PaCO++.

Tout d'abord, il s'avère que PaCO++ permet de réaliser correctement le transfert de données entre deux codes parallèles. Cependant, la notion d'objets distribuées est difficile pour des non-spécialistes. Trop de détails sont apparents même si un certain nombre de détails est dû à la jeunesse de PaCO++. Ainsi, GridCCM apparaît beaucoup plus approprié, d'autant plus que le concept de composant est très proche des concepts manipulés par la communauté de la simulation numérique.

2.2.4 Discussion

Le modèle d'entité parallèle distribuée a été validé par PaCO++ et GridCCM. En particulier, les possibilités d'avoir un modèle abstrait, de déléguer la gestion des distribution et d'ordonnancement sont très appréciables. Elles permettent de dissocier les problèmes et d'intégrer assez facilement les résultats de la communauté comme nous l'avons montré.

Les projections du modèle abstrait n'ont été réalisées que sur la technologie CORBA. Des études préliminaires nous laissent penser que le modèle devrait s'appliquer également à des technologies services ou pair-à-pair. À court terme, nous envisageons définir des pairs parallèles pour permettre des communications parallèles dans des environnements tels que

JuxMem [3]. L'application du modèle abstrait aux Web Services serait également intéressante pour valider sa pertinence.

L'application du modèle abstrait à un modèle de composant est assez directe. Il nous manque de finir l'implémentation de GridCCM mais il ne devrait s'agir que d'implémentation. Les questions ouvertes concernent les relations entre les modèles de composants parallèles et hiérarchique. Est-il faisable d'avoir un modèle hiérarchique de composants parallèles ? A priori, oui, mais il reste à le définir précisément.

Un autre problème, apparu lors de l'évaluation de PaCO++, concerne le choix de la primitive de communication entre composant. Si les interfaces ont de bonnes propriétés, elles présentent l'inconvénient d'être peu compatibles avec une approche zéro-copie. En effet, dans un tel modèle, le programmeur ne peut pas spécifier où et comment recevoir les données. De plus, il faudrait une primitive permettant le zéro-copie et offrant une sémantique identique aussi bien lors des communications entre entités dans des processus différents qu'au sein d'un même processus. Nous pouvons aussi noter que les attributs actuellement utilisés par les modèles distribués pour spécifier le sens de passage des arguments (`in`, `out`, `inout`) n'apparaissent pas bien adaptés aux communications intra-processus.

2.3 Un modèle de composants avec partage de données

2.3.1 Des modèles de données

Jusqu'à présent, nous avons envisagé le problème où deux entités communiquent explicitement ; la communication pouvant être réalisée par du passage de contrôle (RPC, RMI, ...) ou par passage de données (évènements, Web Services, ...).

Un autre modèle de communication entre deux entités est possible : il s'agit du partage de données. Actuellement, c'est le modèle de communication privilégié entre threads au sein d'un même processus via la mémoire du processus. C'est également une méthode de communication standard entre processus d'une même machine via les segments partagées. Le partage de donnée est également utilisé entre machines d'un réseau local ou d'une grappe de machines ; on parle alors de Mémoire Virtuelle Partagée. Les implémentations des MVP sont possibles au niveau intergiciels (TreadMarks [42], Mome [39], DSM-PM2 [2], ...) ainsi qu'au niveau du système d'exploitation et particulier dans les système à image unique [52] (Kerri-

ghed [58], Mosix [9], etc).

Plus récemment, des travaux s'intéressent à l'implémentation du concept de partage de données au niveau des grilles. Par exemple, JuxMem [3] cherche à fournir un service de gestion de données en se basant sur les approches pair-à-pair et MVP.

Le partage de données apporte un concept différent du transfert de données car il est lié au concept de cohérence qui définit comment déterminer qui à le droit de lire et/ou d'écrire et quelle est la vue qu'à chaque entité d'une donnée. Ainsi, une tâche très complexe n'est plus à la charge du programmeur.

Il nous est paru intéressant d'étudier s'il était possible d'importer le concept de partage de données dans les modèles de composants logiciels. D'une part, cela permet d'unifier des modèles de programmation différents, et d'autre part, des applications de couplage de code exhibent parfois un patron d'accès aux données qui s'exprime mieux en partage de données qu'en transfert de données. C'est ainsi que l'équipe Sinetics d'EDF R&D se trouve confronté au problème de la gestion des données dans la plate-forme à base de composants Salome [84].

2.3.2 Un modèle de port orienté données

Profitant du développement de la plate-forme JuxMem dans le projet PARIS, nous avons lancé en collaboration avec Gabriel Antoniu l'étude des modifications à apporter à un modèle de composant pour qu'il supporte le partage de données. Le travail a été initié avec Hinde Bouziane (doctorante que je co-encadre avec Thierry Priol), et Mathieu Jan (doctorant co-encadré par Gabriel Antoniu et Luc Bougé). Il s'est poursuivi avec le co-encadrement du stage de fin d'étude d'ingénieur de Landry Breuil.

Similairement à la gestion de composants parallèles, nous avons d'abord défini un modèle abstrait de composant supportant le partage de données. Ensuite, le modèle a été appliqué au modèle de composant CORBA. Enfin, une implémentation supportant le partage de données via un système de fichier partagé (NFS) et JuxMem a été réalisée.

Un modèle abstrait de composants logiciels avec partage de données

Le modèle abstrait, se basant sur le principe que les composants inter-agissent via des ports, définit un nouveau type de port réalisant le partage de données. Ce type de port, que nous appellerons dans la suite un port de donnée, a en fait deux variantes : un port de type *partage* (*shares*) qui ex-

prime le fait qu'un composant met à disposition une donnée et un port de type *accède* (*accesses*) qui exprime le fait qu'un composant veut se connecter à une donnée fournie par un autre composant. Ainsi, l'assemblage de composants via des ports compatibles est respecté. Le type de la donnée attachée à un port nous apparaît devoir faire partie de la définition du port, ce qui permet d'avoir une vérification de compatibilité de type avant toute connexion.

La vision interne au composant des ports apparaît très similaire pour les deux variantes du port de données. Une interface permettant de lire et d'écrire une donnée est attendue. Afin de régler les problèmes de synchronisation, l'interface devrait aussi fournir des primitives de verrouillage de la donnée.

La différence entre les deux variantes est qu'un port de type *partage* doit offrir des fonctionnalités pour attacher le port à une donnée. Différents scénarios peuvent être envisagés à ce niveau ; la donnée ne devant pas forcément être présente en mémoire mais pouvant être géré par un système de fichier ou par un service de données tels que JuxMem. Ce point est traité dans la section suivante.

Application du modèle abstrait à CCM

Nous avons étendu le modèle de composant de CORBA à partir du modèle abstrait de port de donnée. Les modifications se situent dans le langage IDL3, dans la projection du langage en IDL2 et dans le Component Implementation Framework (CIF). Nous présentons brièvement ces modifications

La modification du langage IDL3 a consisté à ajouter deux nouveaux mot-clés – `shares` et `accesses` – décrivant les deux variantes de types de ports. Ces mots-clés sont suivis par le type de données qui leur est associé et par le nom du port. La figure 2.5 présente la déclaration de deux composants utilisant ces deux mots-clés.

Concernant la projection IDL3 en IDL2, les ports se projettent en opérations de connexion très similaire au couple de ports `provides/uses` : un port `shares` implique une opération fournissant une référence sur une donnée et un port `accesses` une opération permettant de lui passer une référence sur une donnée. Actuellement, le type de la référence est une chaîne de caractère considérée opaque. Si la donnée est stockée dans JuxMem, la référence est alors une référence liée à JuxMem (`JuxMem_Id`). Si un système de fichier partagé est utilisé, la référence peut être le chemin complet vers le fichier stockant la donnée.

```
typedef float[10] aType;

component F{
   shares aType portF1;
};

component A{
   accesses aType portA1;
};
```

FIG. 2.5 : *Le composant F exporte un accès à une donnée de type tableau de réelle ; le composant A peut accéder à tel type de données : les deux ports `portA1` et `portF1` sont donc composables.*

```
local interface CCM_portA1 {
// Accès à la donnée
   float*  get_pointer();
   unsigned get_size();
// Opérations de verrous
   void  acquire();
   void  acquireR();
   void  release();
};
```

FIG. 2.6 : *Interface d'accès interne à un port de type données.*

La dernière modification concerne le CIF. Elle se décompose en deux parties. Un port `accesses` étant implémenté par le contexte du composant, la seule modification est d'ajouter dans le contexte une opération permettant de retrouver une référence sur un objet CORBA local implémentant le port. Un exemple d'interface est présenté en figure 2.6. Un port `shares` également implique un tel objet local et donc une opération similaire dans son contexte. Ce choix est motivé d'une part par la volonté de rentre transparent à l'utilisateur la technologie utilisée pour stocker la donnée et d'autre part par la constatation que c'est du code pouvant être généré pour les cas considérés. Un port `shares` implique également une opération dans l'interface principale du composant. Cette opération doit retourner une référence sur la donnée. Elle est appelée lorsque le conteneur a besoin d'une telle référence, par exemple lorsque le port est connecté à un port de type `accesses`.

Une implémentation avec CCM et JuxMem

Une implémentation du concept de port de donnée a été réalisée sur l'implémentation MicoCCM [83]. Cette implémentation supporte deux systèmes de partage de la donnée : un système de fichier partagé (NFS) et JuxMem. L'approche retenue est la même que pour PaCO++ : s'agissant d'une extension à CCM, nous avons choisi de l'implémenter au-dessus de CCM.

Ainsi, notre implémentation est portable sur toute implémentation CCM C++.

2.3.3 Discussion

L'introduction d'un support de donnée partagée dans les modèles de composants apparaît très naturelle car elle exprime un type de relation entre composants. Or, les modèles de composant se focalisent justement sur l'expression des relations entre composants. Au regard des applications qui expriment des besoins d'accès à des données, nous sommes convaincu que c'est une piste intéressante à poursuivre. Cependant, les problèmes ouverts ne manquent pas.

Tout d'abord, concernant le choix de la technologie d'accès à la données, nous avons choisi de simplifier le problème en l'associant lors de la génération du code du composant. Notre compilateur ne supportant pas le langage CIDL, une option du compilateur contrôle ce choix. La suite immédiate est de modifier le langage CIDL pour y permettre d'exprimer ce choix.

À plus long terme, nous pouvons nous interroger sur la pertinence d'associer un composant à une technologie. En effet, du point du vue utilisateur, un port `accesses` est totalement ignorant de ce choix. Cela est moins vrai pour un port `shares` car l'utilisateur doit implémenter une opération retournant une référence sur la donnée, référence de type opaque dépendant de la technologie choisie.

Enfin, notre choix a été d'associer une donnée à un composant. Ce choix permet de ne pas remettre en cause le modèle de composant : il ne contient que des composants "standard". Cependant, il implique qu'une donnée ne peut exister sans composant *déployé*. Il n'est pas évident de savoir si cela va constituer un obstacle ou si cela représente une bonne propriété. Par exemple, si nous considérons une application qui produit des données puis les consomme ultérieurement, la question est de savoir comment représenter ces données entre leur production et leur consommation. Faut-il associer les composants au système de stockage des données pour n'avoir que des composants "standards" ? Cette approche suppose qu'il y ait toujours au moins un composant de déployé. Elle parait limitée car, par exemple, elle ne semble pas bien adaptée pour supporter le stockage sur disque. Ou faut-il rendre ce système invisible en associant les composants aux données ? Une possibilité serait d'avoir des composants "passifs" car constitués uniquement de données. Non seulement, il reste à définir de tels composants, mais il reste au préalable à déterminer si c'est bien des composants.

2.4 Vers un modèle de composant logiciel dynamique

2.4.1 De la dynamicité

Les modèles de composants logiciels se sont principalement attachés à décrire le comportement *statique* d'une application. Le comportement dynamique est souvent caché dans l'implémentation des composants et n'est donc pas visible. Quelques modèles de composants, tels Darwin [54], étudient les possibilités de support de la dynamicité dans les composants logiciels. Bien que proposant une piste intéressante d'intégration entre modèles de composant et langages de coordination [5], Darwin soufre d'inconvénients tels que le mélange de la définition et de la déclaration de composants, de l'hypothèse d'un environnement statique, et d'une définition trop approximative de la notion service.

La dynamicité dans les modèles de composants logiciels concerne la création (et la suppression) d'un composant durant l'exécution de l'application. Elle concerne également les connexions dynamiques ainsi que les relations mettant en jeu un nombre variable de composants. Un exemple de telles relations est la relation maître-travailleur. Elle constitue le point de départ de notre approche de l'étude des liens entre la dynamicité et les modèles de composants logiciels car c'est un cas simple mais très important d'applications dynamiques. Ce travail ayant débuté il y a un an avec le co-encadrement du doctorat d'Hinde Bouziane, la présentation en est volontairement brève.

2.4.2 Un modèle de composant supportant le paradigme maître-travailleurs

Les applications de type maître-travailleurs apparaissent comme un point de départ idéal pour l'étude de la dynamicité dans les modèles de composants pour de multiples raisons : le concept s'exprime facilement, c'est un type d'application très répandu, de nombreux environnements spécialisés visent à le supporter (des ASP à base de GridRPC [67] comme Nelsolve [16], Diet [15] ou Ninf [72] aux environnements de calcul global tels que SETI@Home [47], Boinc [78], XtremWeb [31]) et enfin c'est un exemple d'application de couplage de code. En effet, une application de simulation d'un couplage de réaction chimique-transport de l'ACI GRID HydroGrid est de ce type. Les réactions chimiques sont simulées conjointement par un seul code parallèle ; par contre la simulation du transport des éléments chimiques est indépendante et peut donc être réalisée par plusieurs exécutions indépendantes du code de transport.

```
Déf. composants          Déf. collection
 component Master {       Implementation collection Workers {
    uses I pM;              contains Worker
 };                         binding Workers.pC to Worker.pW
 component Worker {       }
    provides I pW;
 }                        Instantiation et assemblage
 collection Workers {     Master m;
    provides I pC;        Workers ws;
 };                       connect m.pM to ws.pC;
```

FIG. 2.7 : *Déclaration, définition et utilisation d'une collection de composants.*

Les modèles classiques de composants logiciels échouent à supporter de manière satisfaisante cette application car le nombre d'éléments chimiques est un paramètre de l'application. Une solution consiste à incorporer cette information dans un composant et donc à cacher des informations d'architecture (nombre de composant, connexion, ...) dans du code. Ce n'est pas une solution satisfaisante car elle rompt la propriété des composants logiciels qui est de rendre explicite les liens entre composants, propriété qui nous semble importante.

Notions de collections

Notre analyse du problème a conclu que les langages de description de composant (IDL3,...) ainsi que les langages de définition d'architecture (ADL) posaient problèmes. Les premier ne permettent de définir que des composants alors que nous avons besoin du concept de *collection* de composant. Les deuxièmes expriment une architecture concrète alors que nous cherchons à exprimer une architecture abstraite.

Notre proposition, en phase d'évaluation, vise à permettre la définition d'un type de *collection de composant* dans le langage de définition de composants. Comme illustré à la figure 2.7, une collection de `Worker` est un nouveau type. Une application de type maître-travailleur s'exprime alors en assemblant un composant de type `Master` à une collection de composants de type `Worker`.

Mise en œuvre des collections

L'application obtenue en reliant un composant à une collection de composant est incomplète. Il y manque en effet le mécanisme de transport des requêtes du maître vers les travailleurs. De très nombreux mécanismes et implémentations existent pour réaliser cette opération : d'une implémentation centralisée avec un ordonnancement de type tourniquet à l'utilisation d'environnement tels que DIET ou XtremWeb. L'objectif que nous poursuivons est de comprendre jusqu'à quel point ce choix concerne des propriétés non-fonctionnelles, principalement de performances. S'il s'agit d'un choix de configuration d'une propriété non-fonctionnelle, alors tout comme la sécurité, ce choix pourrait être réalisé lors du déploiement et/ou à l'exécution en fonction des services disponibles.

L'approche retenue actuellement consiste à transformer un ADL abstrait en un ADL concret. La transformation consiste à insérer des composants chargés du transport des requêtes entre un maître et des travailleurs. Dans un premier temps, nous avons utilisé des composants maisons fournissant des mécanismes de transport tourniquet ou aléatoire, centralisé ou hiérarchique. Ainsi, la définition d'une application de type maître-travailleurs peut ne plus incorporer des détails architecturaux relatifs à des propriétés non-fonctionnelles.

L'application obtenue est statique car le nombre de composant travailleur est fixé lors de cette transformation. Une première solution consiste à ajouter lors de la transformation des composants chargés du comportement dynamique de l'application. Par exemple, un composant peut surveiller la charge des composants travailleur et décider d'en ajouter ou d'en enlever en fonction de celle-ci. Cette voie nous semble fort prometteuse car elle nous permet de faire le lien avec les travaux concernant l'adaptation d'application à base de composants logiciels.

2.4.3 Discussion

Le support de la dynamicité dans les modèles de composants peut être envisagé à différents niveaux d'abstraction. Notre choix est d'obtenir un plus haut niveau d'abstraction en tentant de reléguer le mécanisme de transport de requêtes ainsi que le mécanisme de création/destruction des composants en tant que propriétés non-fonctionnelles de l'application. Nous avons été conduit à définir des ADL abstraits et des mécanismes de concrétisation de ces ADL. C'est une étape importante car elle permet de séparer les rôles et de proposer un modèle adapté aux concepteurs de composants et aux architectes de l'application.

Parmi les nombreuses questions ouvertes se trouve celle des collections de composants. Peuvent-elles être définies en tant que composites ? Une réponse affirmative permettrait de faire l'économie d'un concept. Nous comptons revisiter leur définition dans un modèle de composant hiérarchique, comme par exemple Fractal, après avoir étudier leurs propriétés dans le modèle de composant CORBA.

La mise en œuvre de collections dynamiques de composants demande de maîtriser des considérations liées au déploiement. Par exemple, il faut disposer de mécanismes de découverte de ressources et/ou de services, de déploiement d'application multi-intergiciel (si DIET est utilisé pour transporter les requêtes), etc. La section 4 sur le déploiement leur est en partie consacrée.

2.5 De la programmation par aspect pour des modèles de composants

2.5.1 De la programmation par aspects

Comme nous l'avons déjà vu précédemment, les modèles de composants logiciels ont en particulier l'objectif de définir l'architecture de l'application. Cependant, ils souffrent du même défaut que les modèles orienté objets en ce qui concerne le support de fonctionnalités transverses à la hiérarchie induite par la structuration en composants. Les travaux autour de la programmation par aspects (AOP) [44] visent à améliorer la modularité des fonctionnalités transverses en permettant une définition centralisée d'unités logicielles qui sont par la suite tissées dans le flot d'exécution d'une application.

Ainsi, les composants logiciels et les aspects visent la structuration d'une application mais selon deux axes orthogonaux. Il nous est donc paru important de maîtriser la programmation par aspect car un certain nombre de fonctionnalités parait plus du ressort des aspects que des composants. Il s'agit par exemple de logging, de la sécurité, d'une partie de la configuration, de la persistance, etc. Cependant, une forte motivation fut l'adaptabilité. Nous sommes convaincu qu'une exécution efficace d'une application sur une grille à besoin de s'adapter car d'une part la grille est par nature dynamique et d'autre part beaucoup d'applications ont un comportement dynamique. C'est ainsi que nous participons à l'ARC INRIA COA dirigé par Jean-Louis Pazat du projet PARIS. COA vise l'adaptation d'applications à base de composants en utilisant les aspects afin d'injecter le code relatif à

l'adaptation lors de la compilation. Notre participation concerne l'application de ces techniques au modèle de composant parallèle GridCCM.

Les travaux existants autour des modèles de composants et des aspects se divisent en principalement deux classes : ceux qui appliquent les aspects à l'intérieur d'un composant et ceux qui appliquent les aspects aux relations entre composants. Or, à notre connaissance, aucun travail n'a défini un aspect capable de relier l'intérieur et l'extérieur d'un composant. Cette fonctionnalité revient en fait à disposer d'aspects s'appliquant sur les ports d'un composant (ajout/retrait/modification).

Nous nous sommes donc intéressé à évaluer la faisabilité d'ajouter dynamiquement un port à un composant. Ce travail a été motivé et réalisé en étroite coopération avec Jean-Marc Menaud du projet OBASCO (École des Mines de Nantes). Nous disposions d'une solide expertise concernant les composants et CCM en particulier ; Jean-Marc Menaud avait l'expertise sur les aspects ainsi qu'Arachne un tisseur d'aspect dynamique. Ensemble, nous avons soumis un sujet à un appel d'offre de stagiaire internationaux INRIA (programme INRIA Internship). Gabriel Lopez, étudiant uruguayen, y répondit avec comme objectif d'ajouter dynamiquement via Arachne un port à un composant CCM.

2.5.2 Tissage dynamique de ports dans CCM

Le modèle de composant de CORBA repose à l'exécution sur un modèle objet. Ainsi, un port dans CCM est toujours réalisé via une interface d'objet. On peut distinguer deux groupes de ports suivant que l'utilisateur doit implémenter l'interface ou non. Le premier groupe contient les ports de type facette (`provides`), puit d'évènements (`consumes`) ainsi que les attributs (`attributes`). Le second groupe est constitué des ports de type réceptacle (`uses`) et source d'évènements (`emits` et `publishes`).

Concernant la vision interne d'un composant (modèle CIF de CCM), l'ajout d'un port du premier groupe consiste à ajouter son implémentation dans le processus. Ajouter un port du second groupe demande de charger l'implémentation du port mais également d'ajouter une opération dans l'interface du contexte du composant ; opération qui permet à l'implémentation du composant d'obtenir une référence vers un objet implémentant ce port.

L'insertion dynamique de code dans un processus est actuellement facilement réalisable grâce au support de la bibliothèque C de chargement dynamique de code (opération `dlopen`). Cependant, peu (ou pas ?) d'implémentation d'aspect permettent d'exprimer des aspects pour des pro-

grammes en `C++`. Arachne ne le pouvait pas car il ne supportait alors que des codes écrits en C ; le support C++ étant en cours. Heureusement, cette limitation ayant pu être contournée dans le prototype, la faisabilité de l'étude ne fut pas remise en cours.

Concernant la vision externe d'un composant, l'ajout de tout port demande également de modifier l'interface externe du composant comme le précise la projection IDL3 en IDL2 de CCM. Par exemple, pour une facette, une opération permettant à distance d'obtenir une référence sur ce port doit être ajoutée dans l'interface principale du composant. Une étude de l'architecture de CORBA nous a montré qu'une façon de réaliser ces modification est d'utiliser le mécanisme d'interception de requêtes – les intercepteurs portables – de CORBA [19]. Ce mécanisme permet en particulier de rediriger des appels et ainsi de permettre l'ajout ou l'interception d'opération dans une interface existante.

Notre prototype permettant d'ajouter une facette à un composant CCM utilise Arachne pour tisser dans les intercepteurs portables de requêtes le code d'indirection des appels associés au nouveau port. Leur implémentation, ainsi que l'implémentation du port, a été compilée en bibliothèques partagées (`shared object`) et est chargées dynamiquement. Les détails sont disponibles dans le rapport de stage de Gabriel Lopez [51].

Vers un langage d'aspect pour composants logiciels

La faisabilité de l'ajout dynamique de ports dans CCM démontrée via le prototype, la fin du stage de Gabriel Lopez fut consacrée à essayer de définir une langage d'aspect pour composants logiciels. L'originalité de ce langage est qu'il est construit au-dessus de trois primitives concernant l'ajout, la modification et la dérivation d'un port. Avant de les introduire brièvement, nous pouvons remarquer qu'il n'y a pas de primitive de retrait d'un port car c'est un opération implicite lors d'un dé-tissage d'un aspect.

La figure 2.8 présente la grammaire de la proposition. Elle s'inspire des langages d'aspect existant tel que AspectJ [80]. La primitive `add` permet d'ajouter un nouveau port. La primitive `modify` exprime la modification d'un port existant. La primitive `derive` permet de créer un nouveau port à partir de la définition d'un port existant.

Ce langage n'est encore qu'une version très préliminaire : le langage de point de jonction n'est pas encore pleinement défini ; le langage d'expression sur les identifiants n'existe pas et enfin seules des exemples assez simples sont disponibles [51]. Cependant, il montre qu'un langage d'aspect s'appliquant sur les ports de composant est faisable et implémentable.

```
aspect IDENTIFIER {

   add [receptacle|facet|attribute] TYPE IDENTIFIER
   to COMPONENTSER
   [{ (OO_JOINT_POINT + ADVICE) | (C++ CODE) }];

   modify [receptacle|facet] COMPONENTSET.PORTSET
   when JOINT_POINT
   [around|before|after|throwing [EXCEPTION_IDENTIFIER]]
   { C++ code };

   derive [receptacle|facet] COMPONENTSET.PORTSET
   (IDENTIFIER_EXPRESSION)
   { C++ code };
}
```

FIG. 2.8 : *Proposition de définition d'un langage d'aspect pour composants.*

2.5.3 Discussion

Le stage de Gabriel Lopez a été très fructueux et a permis de tisser des liens avec le projet OBASCO. Il a permis une avancée dans l'intégration des modèles de composants et la programmation par aspects. En effet, les modèles de composants logiciels possèdent de bonnes propriétés mais aussi quelques inconvénients. En particulier, ils ne savent pas traiter les fonctionnalités transverses. Ainsi, l'objectif de notre travail a été d'étudier les possibilités d'application des aspects aux modèles de composants.

Cependant, cet objectif est loin d'être atteint. Non seulement il reste à valider la proposition de langage d'aspect pour composants mais il nous est évident qu'il faut l'étendre. Une extension attendue est le support des connexions comme par exemple le changement de connexion ou la connexion d'un port nouvellement créé. Une deuxième extension concerne le support multi-langage. Un composant CCM peut en effet être implémenté en différents langages. Or, il peut être souhaitable de disposer d'un aspect qui s'applique sur toutes les implémentations du composants, comme par exemple un aspect de sécurité. Mais il peut être également souhaitable de disposer d'un aspect ne s'appliquant qu'à une implémentation d'un composant car dépendant très fortement de son implémentation.

Enfin, l'inverse – appliquer l'approche par composants logiciels à des

aspects – est également important et constitue une piste de recherche. Ainsi, les aspects pourraient bénéficier des propriétés de structuration et de réutilisabilité des composants logiciels. Cette thématique s'inscrit dans la vision multidimensionnelle de la programmation qui commence à se dessiner [73]. L'intégration des composants logicielles et des aspects dans un même modèle paraît envisageable de par leur orthogonalité de concept.

2.6 Bilan et perspectives

Notre premier axe de recherche est de définir un modèle de programmation adapté pour les simulations numériques distribuées, et en particulier pour les applications de couplage de codes. Les modèles de composants logiciels fournissent un point de départ fort pertinent.

Nous avons tout d'abord cherché à étendre les modèles de composants logiciels pour supporter les codes parallèles. Nous avons alors défini un modèle abstrait d'entité parallèle distribuée qui ne se restreint pas aux modèles de composants. Ce modèle, via les deux prototypes existant PaCO++ et GridCCM, a atteint ces objectifs. Le séjour post doctoral d'André Ribes chez EDF Synetics à permis d'évaluer les possibilités de transfert industriel de ce modèle dans la plate-forme Salome.

Nous nous sommes ensuite intéressés au partage de données entre composants qui peut être également vu comme l'accès d'un composant à des données externes. Nous avons proposé un modèle abstrait de port d'accès à une donnée, modèle qui a été validé avec une prototype étendant CCM. Ce travail a permis de relier les modèles de composants logiciels avec des services de gestion des données comme par exemple JuxMem. La justification de l'importance de cet axe de recherche a été apportée par des discussions avec l'équipe Sinetics d'EDF R&D qui rencontre ce problème de gestion de donnée dans Salome. En effet, Salome mélangent un modèle de composant logiciel et un modèle de flots de données, laissant mal définie la notion de donnée.

Un autre axe de recherche consiste à apporter des propriétés de dynamicité dans les modèles de composants logiciels. Un objectif à moyen terme est de comprendre les liaisons entre les modèles de flots de données et/ou de contrôle et les modèles de composants logiciels. Nous avons commencé à nous intéresser aux applications de type maître-travailleurs pour nous permette de définir la notion de collection. C'est une étape vers l'étude de la notion de service, concept très utilisé en programmation distribuée, et les modèles de composants logiciels.

Ces différentes pistes de recherche contribuent à notre axe de recherche à long terme qui serait la définition d'un modèle de composant logiciel masquant totalement la localisation, haute performance (notamment avec du zéro-copie) et fournissant les abstractions nécessaires pour le support de simulations numériques distribuées comme par exemple le parallélisme, la hiérarchie, le partage de données, la dynamicité et enfin un support pour l'adaptabilité.

Chapitre 3

Une plate-forme d'intégration d'exécutifs communicants

Sommaire

3.1 Introduction **39**
3.2 Des communications dans les grilles informatiques . . **41**
3.2.1 Des réseaux de communications dans les grilles informatiques 41
3.2.2 Des paradigmes de communications 43
3.2.3 Discussion 44
3.3 Un modèle de plate-forme d'intégration d'exécutifs . . **45**
3.3.1 Un modèle d'abstraction multi-paradigme 46
3.3.2 Architecture d'une plate-forme multi-paradigme 47
3.3.3 Abstractions de communications multi-paradigmes 49
3.3.4 Virtualisation des interfaces de communications . 53
3.3.5 Accès arbitré aux ressources 54
3.4 PadicoTM : une implémentation du modèle **55**
3.4.1 Évaluation des performance 57
3.5 Bilan et perspectives **58**

3.1 Introduction

Les réseaux d'interconnexion ont connu ces dernières années une forte accélération dans leur développement. Cela a été notamment le cas pour

les réseaux longue distance dont le débit a et continue de croître très rapidement. Les débits déjà atteints ont permis la mise en œuvre des grilles informatiques.

Les grilles sont constituées de technologies issues du parallélisme et du réparti. Les technologies issues du parallélisme se concentrent sur l'obtention de très haute performance et sur la portabilité des applications. C'est pourquoi, elles définissent des interfaces de programmation et non des protocoles réseaux. A contrario, les technologies venant du réparti ont comme contrainte très forte l'interopérabilité. C'est pourquoi, elles définissent principalement des protocoles. Des interfaces sont aussi définies mais cela ne s'avère pas une nécessité. Par exemple, aucune interface n'est définie pour les Web Services [18] qui n'en connaissent pas moins un engouement très fort.

Cette différence d'objectifs – haute performance pour le parallélisme et interopérabilité pour le réparti – pose le problème de la séparation (ou non) des technologies réseaux des intergiciels utilisés. Par exemple, est-ce que le choix d'une technologie (parallèle ou répartie) doit-elle restreindre les réseaux qu'une application peut utiliser ? Une telle limitation nous paraît aller contre l'idée des grilles qui prônent implicitement le découplage entre une application et les ressources utilisées pour son exécution.

Ainsi, quel que soit l'intergiciel ou l'exécutif utilisé, il doit être capable d'utiliser tout type de réseau. Cette propriété est déjà grandement présente dans la plupart des intergiciels du parallélisme mais elle constitue une option de configuration à la compilation alors que nous voulons que cela soit un choix dynamique. Les intergiciels du réparti se contentent pour la grande majorité de l'interface `socket` [68] qui est l'interface classique. L'utilisation de cette interface pour tous les intergiciels n'est pas une solution à cause de la limitation de performance générée sur les réseaux haute performance.

La propriété d'utiliser efficacement tout type de réseau de manière transparente est d'autant plus importante que le modèle de programmation pour grilles informatiques, que la section précédente a présenté, mélange des technologies du réparti et du parallélisme. Une motivation forte est de mieux structurer les applications. Cependant, cela ne doit pas constituer un obstacle dans l'obtention de haute performance ni d'interopérabilité. Ainsi, une application développée avec un tel modèle de programmation mais déployée sur une grappe avec un réseau haute performance doit avoir des performances équivalentes à la même application développée avec seulement une technologie parallèle. Cette contrainte implique que la technologie distribuée soit capable d'utiliser efficacement le réseau haute perfor-

mance.

Une autre propriété recherchée est que plusieurs intergiciels soient capables d'utiliser conjointement un réseau haute performance. Dans notre cas, il s'agit au minimum d'un intergiciel réparti et d'un intergiciel parallèle. Cette propriété dépend d'options globales du processus – comme le traitement associé au signal de réception d'un message (`SIGIO`) – mais surtout de la réception des messages : le nombre de processus léger de scrutation, leur fréquence, ... doivent être très minutieusement définis pour l'obtention de haute performance.

Enfin, comme nous l'avons déjà évoqué, la propriété de haute performance est fondamentale pour l'acceptation des grilles : beaucoup de simulations numériques sont limitées par la puissance des machines. La virtualisation nécessaire pour réaliser nos objectifs ne doit pas entraîner de surcoût significatif – en particulier sur les réseaux haute performance.

Nous nous sommes donc impliqué dans la recherche d'un mécanisme permettant de découpler les interfaces d'accès aux réseaux des réseaux eux-même afin de supporter une exécution efficace des modèles de programmation pour grilles informatiques. Ce travail a été réalisé lors du doctorat d'Alexandre Denis [24], que j'ai co-encadré avec Thierry Priol. Il a demandé de maîtriser avec beaucoup de minutie de nombreux détails technologiques qui ne sont pas présentés dans ce document. Le lecteur intéressé pourra se référer à [24].

Cette section continue par quelques rappels sur les réseaux constituant les grilles ainsi que les intergiciels utilisées. Nous présentons ensuite dans la section 3.3 l'architecture abstraite d'une plate-forme d'intégration d'exécutifs communicants pour la programmation des grilles. PadicoTM, une implémentation d'une telle architecture, est décrit dans la section 3.4. Le bilan est dressé dans la section 3.5.

3.2 Des communications dans les grilles informatiques

3.2.1 Des réseaux de communications dans les grilles informatiques

Les grilles informatiques sont constituées de trois grandes catégories de réseaux :

SAN (*System Area Network*, réseau haute performance) Ces réseaux sont principalement dédiés à la construction de grappes. Leur portée est

généralement de quelques mètres pour permettre une très haute performance en débit et surtout en latence. Les bibliothèques d'accès à ces réseaux ont pris l'habitude de court-circuiter le système d'exploitation pour réduire au maximum la latence. Elles proposent des interfaces spécifiques.

LAN (*Local Area Network*, réseau local) Il s'agit d'un réseau interne à une institution, d'une portée de quelques mètres à quelques centaines de mètres et géré par une seule personne ou une seule organisation. La technologie très souvent utilisée est Ethernet avec le protocole IP. Comme pour les SAN, la sécurité est implicite car tout le réseau est sous contrôle.

WAN (*Wide Area Network*, réseau longue distance) Ces réseaux ont une portée en kilomètres. Interconnectant en générale différentes organisations, ils sont en général considérés comme des réseaux ouverts et donc la question de la sécurité se pose. Cependant, des réseaux longue distance peuvent être privée comme par exemple la dorsale de Grid'5000 [14] ou TeraGrid [86]. Les technologies utilisées concernant le transport physique varient grandement mais c'est très fréquemment le protocole IP qui est utilisé.

Ainsi, les SAN demandent l'utilisation de protocoles et d'interfaces spécifiques pour l'obtention de tout leur potentiel. À contrario, les LAN et les WAN, imposant le protocole IP, sont généralement utilisés avec TCP/IP. Cependant, de nouveaux protocoles émergent parfois comme récemment le protocole SCTP [69], qui a de bonnes propriétés mais pas encore d'implémentation performante.

Pour faire face aux spécificités des WAN, des protocoles adaptés ont été développés, principalement au-dessus de TCP/IP. Ainsi, des méthodes de chiffrement telles que SSL permettent de transporter en sécurité des données, en attendant que les protocoles IPsec et DNSsec soient déployés. Concernant des réseaux à faible débit, des protocoles de compression adaptatifs sont apparus comme AdHoc [38]. Enfin, l'utilisation des réseaux longue distance haut débit s'avère plus difficile en raison du *slow-start* de TCP en cas de perte de paquet. Ces pertes ont été démontrées même sans congestion du réseau. Des protocoles *multisocket* ont été développés afin de recouvrir le *slow start* d'une connexion par d'autre connexion [34].

Ainsi, l'utilisation efficace d'un réseau dépend beaucoup du choix de l'interface d'accès au réseau ainsi que des protocoles à mis en œuvre. La transparence d'accès au réseau, généralement demandée par les programmeurs, n'est donc pas immédiate à obtenir.

3.2.2 Des paradigmes de communications

Les interfaces d'accès au réseau ont évoluées vers de plus en plus d'abstraction et une intégration de plus en plus forte vers les langages de programmation. Nous organisons ce bref survol des paradigmes de communication en séparant les paradigmes du parallélisme de ceux du réparti.

Paradigmes de communications parallèles

Les paradigmes de communications parallèles peuvent être rangés en trois catégories. Premièrement, nous trouvons tous les intergiciels orientés passages de messages sur une topologie connue qui fournissent surtout des opérations collectives telles que la diffusion, la réduction, etc. PVM [30] et MPI [32] sont les intergiciels les plus courants. Deuxièmement, des intergiciels fournissent une implémentation de la notion de mémoire virtuellement partagée comme par exemple TreadMarks [42], DSM-PM2 [2] ou encore Mome [39]. Troisièmement, des intergiciels visent à fournir non pas un environnement de programmation pour l'utilisateur mais une plate-forme générique de communication. Ils sont destinés soit à servir de base à des intergiciels de plus haut niveau ou à servir de fondation aux exécutifs des langages parallèles. Ces intergiciels supposent une topologie connue et relativement statique. Des exemples sont Fast Message [62], Madeleine [6], Ibis [75], et enfin Nexus [27].

Paradigmes de communications réparties

Les paradigmes de communications réparties sont plus variés et en un sens ont suivi une évolution assez liée à celle des langages de programmation. L'idée a été de masquer la répartition dans des constructions des langages de programmation.

Les interfaces de bas niveau, de type `socket`, fournissent un accès explicite au réseau. Elles représentent l'abstraction du réseau proposée par le système d'exploitation aux utilisateurs. Par analogie avec les langages de programmation, elles correspondraient aux langages assembleur. Il est à noter que le protocole HTTP, utilisée à travers des interfaces spécifiques, est de plus en plus souvent considéré comme le langage d'Internet.

Les langages procéduraux ont comme équivalent les appels de procédure à distance (*Remote Procedure Call*), popularisé par SUN [70] puis par l'Open Group [23]. C'est une étape importante car les RPC ont commencé à masquer les aspects réseaux. Mais ce n'est qu'avec les langages objets et

les appels de méthodes à distance (*Remote Method Invocation*) que les aspects réseaux ont pu être totalement supprimés : il n'y a plus de différence entre une invocation locale et une invocation distante. Les exemples les plus connus sont le modèle objet de CORBA [19] ainsi que les RMI JAVA [33].

Alors que les modèles objets distribués ont suscités beaucoup d'espoir, leur acceptation est fortement remise en cause par les architectures orienté services (SOA). Les Web Services représentent l'implémentation la plus connue. Ces modèles proposent une absence d'état et un prototypage vérifié dynamiquement et non plus statiquement. D'un point de vue de paradigme, c'est une variation du passage de message.

Le paradigme de pair-à-pair n'a pas d'équivalent évident dans les langages de programmation. Il constitue une variation intéressante des SOA car un client n'a plus besoin de connaître l'identité d'un serveur lui fournissant un service.

Enfin, il existe des paradigme incluant une gestion du temps comme le standard HLA (*High Level Architecture*). HLA définit des règles d'interactions, un modèle objet et une spécification d'interface.

Paradigme de communication pour grilles informatiques

Le chapitre précédent a montré que des modèles de programmation bien adaptés pour les grilles informatiques mélangent les paradigmes de communications du calcul parallèle et du calcul réparti. Ainsi, PaCO++ ou GridCCM font intervenir CORBA et au moins une technologie parallèle telle que MPI, PVM ou une MVP.

Des modèles de programmation plus avancés, comme par exemple le support des données dans des modèles de composant parallèles tel que ébauché dans la section 2.3, demandent au moins le support de plusieurs intergiciels – CORBA, MPI et JXTA par exemple. Si nous rajoutons le support de la dynamicité ou encore des mécanismes d'interopérabilité vis à vis des Web Services, une application peut se retrouver à devoir intégrer de nombreux intergiciels, venant du calcul parallèle et du calcul réparti. Comme les applications scientifiques séquentiels sont devenus multi-langages, les applications scientifiques distribuées sont en train de devenir multi-intergiciels.

3.2.3 Discussion

D'une part, les grilles informatiques offrent un vaste spectre de type de réseaux avec parfois des interfaces spécifiques, des méthodes d'utilisation

adaptées afin d'offrir de la sécurité ou une utilisation efficace des réseaux. D'autre part, les modèles de programmation sont variés, et peuvent être divisés en deux groupes de propriétés distinctes : haute performance pour le calcul parallèle et interopérabilité pour le calcul réparti. Les modèles de programmation pour grilles informatiques que nous avons présentés dans le chapitre précédent présentent en plus la particularité de combiner ces deux propriétés.

La solution retenue par les intergiciels afin de supporter le plus de types de réseaux est d'implémenter une interface de portabilité et d'implémenter cette interface sur le plus de types de réseaux possible. Si cette approche a un sens pour un intergiciel, elle constitue une duplication des efforts certaine car ce même travail doit être refait pour *chaque* intergiciel.

Une multitude de plates-formes génériques existent dans les calcul parallèle et réparti. Cependant, les calculs réparti et parallèle étant différents, l'utilisation d'une plate-forme d'un type de calcul dans l'autre conduit soit à des surcoûts significatifs soit à une manque de fonctionnalité.

Notre objectif est de permettre le découplage des intergiciels par rapport au réseau pour faciliter le déploiement et supporter plusieurs exécutifs simultanément sans perte de performance significative. Nos recherches ont été guidées par l'analyse suivante : une plate-forme de communication générique pour grilles informatiques se doit de supporter plusieurs paradigmes simultanément de part leur différence de principe. L'obtention d'une telle plate-forme est cruciale pour la validation des modèles de programmation présentés au chapitre précédent.

3.3 Un modèle de plate-forme d'intégration d'exécutifs communicants

Nous avons proposé un modèle de plate-forme de communication pour grilles informatiques. Le modèle gère plusieurs paradigmes de communication et autorise l'utilisation de ressources variées de façon transparente. Les principaux aspects qui ont guidés sa conception sont :

- une prise en charge d'une hiérarchie complète des réseaux, des SAN aux WAN en les exploitant suivant leur paradigme natif, et éventuellement à l'aide de méthodes spécifiques ;
- la possibilité d'utiliser des exécutifs variés quelque soit leur paradigme ;
- la possibilité d'utiliser plusieurs exécutifs simultanément, en combinaison arbitraire ;

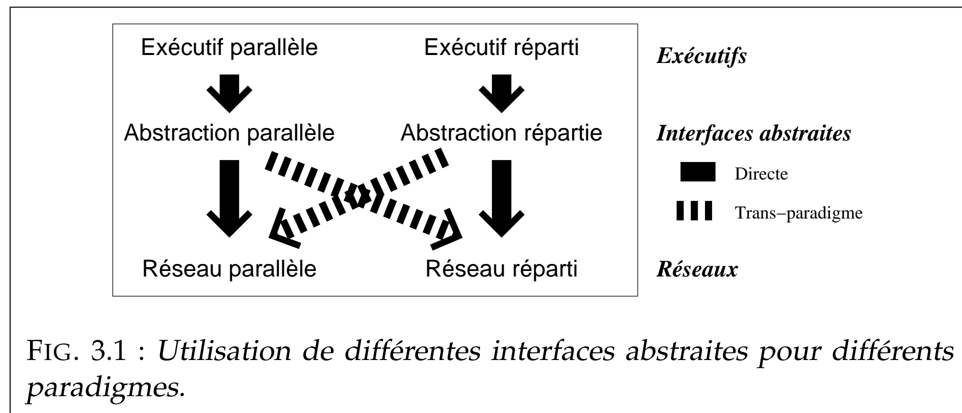

FIG. 3.1 : *Utilisation de différentes interfaces abstraites pour différents paradigmes.*

- la mise en commun des méthodes de communication : tous les réseaux et toutes les méthodes de communication sont mis à disposition de tous les exécutifs ;
- la possibilité d'avoir une implémentation haute performance.

Nous commençons par présenter dans la section 3.3.1 notre modèle d'abstraction multi-paradigme. L'architecture de notre modèle est décrit dans la section 3.3.2. Les trois niveaux constituant cette architecture sont introduits dans les section suivantes.

3.3.1 Un modèle d'abstraction multi-paradigme

L'intégration de plusieurs abstractions de communications est classiquement réalisée en utilisant une interface de fondation commune. Une fois cette interface définie, il ne reste plus qu'un portage des différentes abstractions sur cette interface. Or, nous avons montré qu'une telle interface était impossible à obtenir si nous voulions garder les propriétés de haute performante et de fonctionnalités non restreintes car les abstractions du calcul parallèle et du calcul réparti sont fondamentalement différentes. En effet, une interface unifiant les deux paradigmes reviendrait à prendre en compte les contraintes les plus fortes (celle du réparti qui impose une connexion dynamique point-à-point) et à renoncer à l'essentiel des optimisation possibles et souhaitables dans le cas où les contraintes sont sciemment relâchées (calcul parallèle).

Nous avons introduit le concept de plate-forme multi-paradigme qui est illustré en figure 3.1. Le principe fondateur est de ne faire que les compromis nécessaires. Ainsi, plutôt que de proposer des interfaces faisant des compromis, la plate-forme propose plusieurs interfaces, chaque interface

étant appropriée pour un paradigme. Il s'en suit qu'il faut réaliser plusieurs incarnations pour chaque abstraction. Par exemple, avec les paradigmes parallèle et réparti et les réseaux correspondant, il faut réaliser quatre incarnations : abstraction parallèle sur réseau parallèle, abstraction répartie sur réseau réparti, abstraction parallèle sur réseau réparti, abstraction répartie sur réseau parallèle. Les deux premières sont qualifiées de directes alors que les deux dernières sont des passerelles *trans-paradigmes*. Les incarnations directes ne réalisent pas de compromis et ne perdent donc pas les propriétés du paradigme concerné. Les incarnations trans-paradigmes ne réalisent que les compromis nécessaires : ces compromis ne sont fait en pratique que lors de l'utilisation d'une abstraction sur un réseau d'un autre paradigme. Les compromis sont alors inévitables.

Il a noter que le nombre de combinaison à implémenter est limité par le nombre de paradigme et non par le nombre d'exécutifs ou de réseaux. Ainsi, si nous considérons en plus des deux paradigmes réparti et parallèle à mémoire distribué, le paradigme de mémoire partagé, le nombre d'incarnations est a priori neuf. Cependant, il semble peu pertinent d'implémenter le paradigme mémoire partagée à ce niveau car c'est un travail complexe qui est déjà réalisé par les intergiciels de MVP ; intergiciels qui peuvent être construits au-dessus du paradigme parallèle sans perte de performance ou de fonctionnalité.

3.3.2 Architecture d'une plate-forme multi-paradigme

Nous avons proposé une architecture générale d'un modèle de gestion de communications d'une plate-forme pour grilles informatiques. Les caractéristiques principales du modèle sont d'être multi-paradigme, d'être multi-exécutif, d'être flexible, c'est à dire les éléments de l'architecture sont composables, et enfin d'être performant.

Le modèle est organisé en plusieurs niveaux d'abstraction, chaque niveau correspondant à un service. À un même niveau, plusieurs éléments peuvent exister mais pour des paradigmes, des méthodes de communications ou des implémentations différentes. La figure 3.2 montre un exemple d'empilement de ces niveaux.

Les niveaux considérés par notre modèle sont au nombre de sept, soit par niveau d'abstraction croissant :

Système C'est le niveau juste au-dessus du matériel. Il peut être fourni par le système d'exploitation ou par un pilote tel que MX pour Myrinet. Les interfaces sont souvent spécifiques.

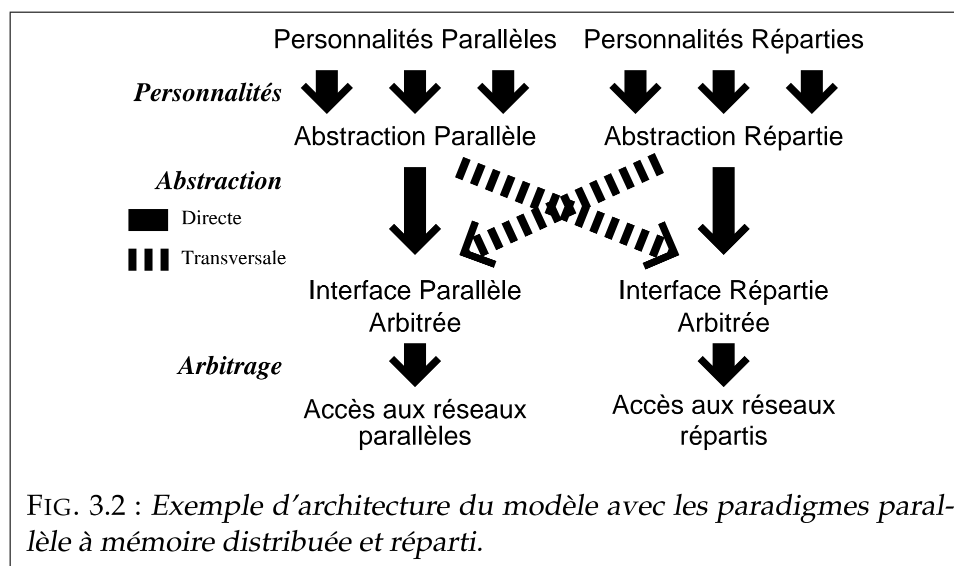

FIG. 3.2 : *Exemple d'architecture du modèle avec les paradigmes parallèle à mémoire distribuée et réparti.*

Portabilité, généricité Ce niveau a pour but de cacher la différence d'interface des pilotes lors de l'accès au réseau. La portabilité est assurer par des interfaces de même paradigme que les interfaces sous-jacentes.

Arbitrage L'arbitrage apporte la ré-entrance, le multiplexage et une harmonisation des choix globaux. Le but de l'arbitrage est de résoudre les conflits d'accès aux ressources qui surgissent lorsque plusieurs exécutifs demandent simultanément l'accès au réseau. L'interface d'un service arbitré est similaire à celle du niveau générique.

Adaptation d'abstraction Le niveau d'abstraction a pour but d'offrir chaque paradigme disponible sur toutes les ressources disponibles. C'est à ce niveau qu'à lieu la traduction trans-paradigme.

Virtualisation Le niveau de virtualisation adapte les interfaces abstraites aux différentes personnalités susceptibles d'être utilisées par les exécutifs.

Exécutif Les exécutifs implémentent un modèle de programmation de haut niveau comme par exemple CORBA, MPI, HLA, les Web Services et les JVM.

Application Le niveau applicatif regroupe les codes au-dessus du niveau des exécutifs. Il peut s'agir de code écrits par l'utilisateur final ou de bibliothèques comme par exemple Scalapack.

La plate-forme de communication que nous proposons englobe les niveaux arbitrage, adaptation d'abstraction et virtualisation. Ainsi, elle ne demande pas l'écriture de pilote spécifique – elle essaie d'utiliser au mieux les pilotes disponibles – ni elle ne demande de modifier les exécutifs existants – ce qui souhaitable au vu de l'effort demandé par l'écriture d'un exécutif. En fait, l'objectif peut être compris comme l'ajout d'une nouvelle fonctionnalités – le découplage entre les interfaces et les accès effectifs au réseau – de manière transparente.

3.3.3 Abstractions de communications multi-paradigmes

Le niveau d'adaptation d'abstraction est le cœur d'une plate-forme multi-paradigme. C'est ce niveau qui réalise les traductions directs entre un même paradigme et surtout les traductions d'un paradigme de communication à un autre.

Ce niveau utilise l'interface offerte par l'accès arbitré, selon le paradigme considéré, et propose au-dessus une interface générique du même paradigme ou d'un autre paradigme. Les interfaces offertes par cette couche sont totalement abstraites et génériques : pour un paradigme, l'interface, unifiée et unique, exprime les propriétés relatives à ce paradigme.

Les paradigmes que nous avons considérés sont le calcul réparti et le parallélisme à mémoire distribuée. Pour chacun de ces paradigmes, les propriétés d'adressage, de topologie, d'établissement et de clôture de connexion et d'échange de messages ont été analysées et définies. Pour le calcul réparti, l'adressage se fait par une adresse absolue (adressage IP par exemple). La topologie connue de chaque nœud est faite des liens point-à-point qui le relient à d'autres nœuds. Les connexions sont dynamiques de type client/serveur. L'échange de message est basé sur les flux. Concernant le calcul parallèle à mémoire distribué, l'adressage est local à l'intérieur du monde connu ; monde qui est organisé suivant une topologie plus ou moins complexe et surtout statique. Il n'y a pas alors de mécanisme de connexions entre nœuds. En plus d'échange de message en point à point, l'abstraction comprend des opérations collectives.

Adaptateur d'abstraction

La couche d'adaptation est constituée d'adaptateur reliant la couche d'accès arbitré à la couche de virtualisation. Nous avons recensé trois types d'adaptateur.

Adaptateur direct C'est le plus simple des adaptateurs qui offre une interface virtualisée du même paradigme que celui de l'accès arbitré. C'est par exemple le cas de l'adaptateur réparti-réparti.

Adaptateur croisé C'est un adaptateur qui réalise une conversion trans-paradigme comme par exemple l'adaptateur réparti-parallèle à mémoire distribuée. La complexité d'un tel adaptateur est variable suivant les propriétés respectives des paradigmes fournis et utilisés.

Adaptateur généralisé Il s'agit de méthodes de communication supplémentaire, en dehors de la collection principale. Il peut s'agir d'un adaptateur alternatif aux adaptateurs directs et croisés, proposant des méthodes de communications supplémentaires ou encore s'intercalant dans les assemblages.

Les adaptateurs généralisés sont décrits après une présentation succincte des abstractions réparti et parallèle à mémoire distribuée.

Abstraction du réparti

L'interface abstraite de cette abstraction a pour objectif d'être utilisée par une personnalité de type socket BSD, POSIX AIO ou encore X/Open XTI. En plus des caractéristiques de topologie et d'adressage que nous avons déjà évoquées, il faut prendre en compte les caractéristiques d'asynchronisme et de protocole. L'interface abstraite n'est constituée que d'opérations asynchrones, les opérations synchrones pouvant simplement être reconstruites au-dessus. La sémantique des opérations est modifiée suivant le protocole associé à la communication. Par exemple, le protocole DATAGRAM implique des messages avec des frontière bien délimitées alors que le protocole STREAM implique un flux.

Les opérations réalisant l'interface abstraite pour le réparti sont subdivisées en trois groupes : la gestion du lien (connexion/déconnexion), l'échange de données, et la scrutation et le contrôle de la synchronisation. L'interface manipule principalement trois types d'objets : des adresses, des

liens via leur descripteur et des observateurs. Un observateur est une entité qui centralise les notifications associées à un lien.

Abstraction parallèle à mémoire distribué

À la différence de l'abstraction du réparti, il n'existe vraiment de standard pour une interface générique et d'assez bas niveau. Il existe bien MPI mais c'est un standard de haut niveau. Nous nous sommes donc fortement inspiré de l'interface de Madeleine 2 [6].

Les caractéristiques de cette abstraction sont basées sur une topologie logique de clique, appelé circuit. Un circuit est un groupe de nœuds, ordonnés, sans doublon, numérotés et invariant dans le temps. Un circuit peut être créé à tout moment mais pour l'instant la modification dynamique d'un circuit n'est pas supportée. La construction et la destruction d'un circuit sont des opérations synchrones, ce qui convient bien au modèle de programmation SPMD. Un nœud peut appartenir à plusieurs circuits. Intuitivement, un circuit correspond à un réseau haute performance.

Afin de supporter efficacement les grappes de grappes, plusieurs circuits peuvent être combinés ensemble afin de créer un circuit. Les contraintes de construction de ces sous-circuits sont que chaque couple de nœud du circuit doit apparaître dans exactement un sous-circuit – les chemins multiples ne sont pas gérés – et chaque sous-circuit est géré par un seul adaptateur.

L'interface abstraite pour le parallélisme à mémoire distribuée est divisée en trois catégories d'opérations : l'administration du circuit, l'échange de message et les opération collections. L'échange de message à point à point est basé sur une interface très proche de celle de Madeleine 2, qui a montré qu'elle était générique et efficace. C'est une interface basée sur la construction incrémentale de message mais contrairement à Madeleine, nous assumons une réception par message actif. Les opérations collectives n'ont pas vocation à réaliser un jeu d'opérations collectives complet, mais uniquement d'exprimer des opérations susceptibles d'être optimisées par les niveaux inférieurs. Actuellement, nous avons seulement retenu les opérations de barrière et de diffusion.

Adaptateurs généralisés

Comme nous l'avons évoqué, il existe d'autres types d'adaptateur que les adaptateurs directs et croisés. Ces adaptateurs peuvent être des adaptateurs alternatifs, des intercepteurs ou des adaptateurs croisés abstraits.

Les adaptateurs alternatifs réalisent la même type d'adaptation qu'un adaptateur de la collection principale mais avec des variantes. Par exemple, dans le cas de l'adaptateur réparti/réparti, des adaptateurs alternatifs sont un adaptateur réalisant la compression ou mettant en œuvre un mécanisme de chiffrement. Les intercepteurs sont des adaptateurs qui utilisent et fournissent la même interface abstraite. Ils permettent de modifier les communications comme par exemple un intercepteurs de compression. Enfin, les adaptateurs croisés abstraits utilisent une interface abstraite en lieu et place d'une interface arbitrée. Ils sont similaires aux intercepteurs si ce n'est que l'interface abstraite utilisée n'est pas du même paradigme que celle fournie. Leur principale attrait est d'élargir les possibilités d'assemblage des adaptateurs. Par exemple, il est ainsi possible d'utiliser un intercepteur de compression sur les parties d'un circuit qui empruntent un réseau "lent" dans devoir développer une version spécifique de l'adaptateur de compression pour le parallélisme.

Les différents types d'adaptateur que nous avons identifiés sont la compression à la volée, les flux parallèles, la connexion passive, l'authentification, le chiffrement, le tunnel et un protocole à tolérance de perte.

Sélection d'assemblage

La richesse d'expressivité des adaptateurs mène au problème du choix d'un assemblage d'adaptateur pour une connexion. Afin que la communication puisse avoir lieu, il faut que le nœud émetteur et le nœud récepteur utilise le même assemblage.

Pour utiliser des assemblages complexes, il faut qu'une négociation ait lieu entre les nœuds ou que ces assemblages soient fixés à la configuration. Le déploiement a une très forte incidence dans ces choix. Par exemple, supposons qu'une application doivent être déployée sans que ses communications soient lisibles par une tierce personne. Les réseaux SAN peuvent être considérés comme privé et donc il n'y a pas besoin d'utiliser un adaptateur de chiffrement. Par contre sur tout réseau public, le chiffrement devient obligatoire : il faut correctement configurer les nœuds de part et d'autre des réseaux publics pour qu'ils utilisent un adaptateur de chiffrement pour les communications utilisant les réseaux publics.

Afin d'assurer une communication entre tout nœud, une stratégie par défaut est possible : les réseaux haute performance sont utilisés si possible – via leur interface arbitrée parallèle – et sinon l'interface arbitrée du réparti est utilisée. Cette stratégie n'utilise pas de méthodes de communication évoluées mais elle permet de communiquer sur la plupart des grappes

et grappes de grappes.

Nous aurons l'occasion de revenir sur ce point lors de la présentation de PadicoTM, une implémentation de ce modèle.

3.3.4 Virtualisation des interfaces de communications

La plate-forme multi-paradigme pour grilles informatiques a pour vocation de supporter tous les intergiciels. Elle est fondée sur le support des deux paradigmes retenus pour l'étude. Cependant, ils existent de nombreux intergiciels et également une multitude d'interface. Afin d'assurer la portabilité de tous les intergiciels sur notre plate-forme, il faut un mécanisme permettant de transformer les interfaces des adaptateurs en des interfaces standards.

Virtualisation par personnalités

Ce rôle est dévolu au niveau de virtualisation. Ainsi, les interfaces standards peuvent être construites au-dessus de l'interface abstraite la plus pertinente. Par exemple, l'interface `socket` est construite au-dessus de l'interface abstraite répartie alors que l'interface Madeleine est reconstruite audessus de l'interface parallèle à mémoire distribué.

La couche de virtualisation est ainsi constituée de codes d'adaptation, codes réalisant juste une adaptation d'interface et non de fonctionnalité. Cette couche de virtualisation étant en général assez mince, ces codes sont appelés personnalités.

Le principale avantage de l'utilisation de personnalité est que les exécutifs existants peuvent être utilisés sans aucune modification puisqu'ils ont accès à une interface qu'ils supportent déjà. Il est donc aisé de supporter les différentes versions d'un même exécutif ou de supporter de nouveaux exécutifs tant qu'ils utilisent des interfaces présentes sous forme de personnalité.

Transparence

La virtualisation par personnalité permet d'atteindre la transparence d'utilisation. L'ensemble formé par les adaptateurs et la personnalité se comporte du point de vue de l'utilisateur comme l'implémentation qu'il émule. Cependant, il faut également assurer la transparence de protocole. Dans le cas du réparti où l'interopérabilité est nécessaire, il faut également que l'ensemble adaptateur et personnalité génère le même message du

point de vue du réseau afin de pouvoir communiquer avec un processus non géré par la plate-forme.

La transparence de protocole demande la coopération de la personnalité pour déterminer quel assemblage d'adaptateurs est à utiliser pour la communication avec des nœuds non gérés par la plate-forme. Il faut donc également déterminer si un nœud distant est ou non géré par une telle plate-forme. Cette difficulté est assez facilement gérable durant la phase de déploiement si tous les nœuds de l'application sont déployés conjointement : les nœuds gérés par la plate-forme sont connus. Par défaut, tout autre nœud est considéré comme ne faisant pas partie de la plate-forme et l'assemblage par défaut est utilisé.

Il est à noter que cette solution s'applique uniquement dans la communication directe entre deux nœuds. Un exemple de cas difficile est le cas d'un nœud interne d'une machine parallèle ne disposant pas d'adresse IP et protégé par un pare-feu voulant communiquer avec le protocole SOAP avec un autre nœud non géré par la plate-forme. Il s'en suit qu'il faut mettre en place des nœuds passerelles réalisant les conversions nécessaires. Cette problématique dépasse notre cadre d'étude.

3.3.5 Accès arbitré aux ressources

Le niveau d'arbitrage a pour rôle principal d'assuré un accès concurrent haute performance aux réseaux. En effet, il y a un accès concurrent aux ressources réseaux : les intergiciel ou les applications peuvent être multithread et/ou la juxtaposition d'exécutifs génère souvent du multithread. De plus, le niveau d'arbitrage doit assurer une cohabitation entre exécutifs, notamment du point de vue des fonctionnalités globales au processus comme les interruptions. Enfin, il doit faire en sorte que les méthodes de réception des différents exécutifs coopèrent et ne rentrent pas en concurrence pour l'accès au réseau. Par exemple, il ne paraît pas performant de laisser chaque boucle de scrutation de chaque intergiciel accéder aux mécanismes de scrutation des réseau.

Méthodologie d'accès arbitré Le contrôle des interactions est basé sur la virtualisation des interfaces. Chaque exécutif accède aux ressources comme s'il était seul. Le niveau d'arbitrage assure que les interfaces fournies sont ré-entrantes. Les interfaces proposées sont souvent spécifiques à chaque technologie réseau. Le choix de chaque interface est guidée par un soucis d'offrir les meilleures propriétés concernant la performance, l'expressivité maximale du matériel, l'évolutivité, la complexité d'intégration et la porta-

bilité. Notre choix a été de considérer Madeleine 2 comme unique méthode d'accès aux réseaux haute performance et les interfaces proposées par le système d'exploitation pour les autres réseaux.

Haute performance L'obtention de haute performance d'une telle plate-forme demande de prendre en compte toutes les opérations mises en œuvre lors des communications. Tout d'abord, les mécanismes de scrutation doivent être mis en commun afin d'éviter les scrutation inutiles : tout intergiciel peut potentiellement recevoir un message de tout réseau. Ensuite, il s'agit de contrôler la compétition et l'équité entre les différents réseaux qui demandent des fréquences de scrutation différentes (fonction des latentes) et les intergiciels qui peuvent vouloir scruter à différentes fréquences. La virtualisation permet de séparer ces deux mécanismes de scrutation : l'accès arbitré fourni des informations concernant l'état du réseau mais ne permet pas de déclencher les opérations de scrutations effectives. Ces dernières sont asynchrones et contrôlé par la plate-forme. Enfin, l'empilement de tous ces niveaux réclament des mécanismes de multiplexage efficaces pour l'obtention de performance.

3.4 PadicoTM : une implémentation du modèle

Cette section présente PadicoTM, une implémentation de l'architecture proposée dans la section précédente. Ne disposant pas alors d'un modèle de composant adapté à la réalisation d'une plate-forme haute performance, PadicoTM fournit un modèle très simple de composants logiciels : les modules. Un micro-noyau (Puk pour PadicoTM micro kernel) assure les fonctionnalités du framework (chargement, déchargement, gestion des dépendances, etc). Les modules peuvent être de type binaire, de package (assemblage de modules) ou encore virtuel (uniquement des attributs). PadicoTM est basé sur les bibliothèques Marcel [21] pour la gestion des threads et Madeleine [6] pour le support des réseaux haute performance.

Niveau d'arbitrage Sans entrer dans les détails de PadicoTM, le module TaskManager gère les aspects systèmes (multithreading, ordonnancement, synchronisation, ...). C'est ce module qui contient la boucle de gestion d'évènement. La couche arbitrée au réseau est implémentée par le module NetAccess qui se subdivise en trois modules : SysIO pour le réparti, MadIIO pour le parallélisme à mémoire distribuée et une partie principale pour gérer les interactions.

Niveau d'abstraction Les modules d'abstraction sont VLink pour le réparti et Circuit pour le parallélisme à mémoire distribuée. La couche d'abstraction contient d'autres modules comme en particulier le module NetSelector qui s'occupe du choix d'assemblage d'adaptateur lors de la première réception ou envoie d'un message. NetSelector fonctionne un peu comme un oracle : il consulte d'autres modules pour répondre. Actuellement, deux modules existent qui implémentent deux stratégies présentées, une politique basée sur la description via un fichier et la politique par défaut.

Niveau d'adaptation d'abstraction De nombreux adaptateurs sont implémentés dans PadicoTM comme Vlink/MadIO, MadIO/Vlink, Vlink/tunnel, Adhoc (compression adaptative), ParallelStreams (flux parallèle sur réseau WAN), VRP (protocole à tolérance de perte de message variable). Il est à noter que Adhoc a été développé par le projet Algorille du LORIA indépendamment de PadicoTM. Cependant, son intégration a été assez simple. L'étude et la première implémentation du module ParallelStreams a été réalisé par Joel Daniels, étudiant de l'Université du New Hampshire (NH, USA), lors de son stage dans le projet PARIS.

Niveau personnalités Nous avons implémenté quatre personnalités : l'interface socket BSD, l'interface POSIX AIO (Asynchronous Input/Output), la version 2.0 des Fast Message et l'interface Madeleine.

Exécutifs sur PadicoTM Grâce aux personnalités, de nombreux intergiciels et exécutifs ont pu être testés et fonctionnent sur PadicoTM. Parmi eux, nous trouvons MPICH (basé sur le portage de MPICH sur Madeleine réalisé par le projet Runtime du LaBRI), de nombreux implémentations de CORBA (omniORB 3.x et 4.x, MICO 2.x, TAO et ORBACUS), une MVP (MOME du projet PARIS), gSOAP, HLA Certi, ICE, JXTA, JuxMem (basé sur JXTA) ainsi que la JVM Kaffe.

La limitation actuelle de PadicoTM provient de son utilisation de la bibliothèque de thread Marcel qui demande de recompiler les exécutifs. Cependant, des travaux sont en cours dans le projet Runtime (LaBRI) pour rendre la bibliothèque Marcel compatible avec l'ABI (Application Binary Interface) des threads sous Linux. Cette compatibilité permettrait enfin à PadicoTM d'utiliser les exécutifs sans recompilation. Outre une plus grande souplesse d'utilisation – et une transparence toujours plus grande – le plus grand avantage serait de supporter les exécutifs uniquement disponibles sous forme de binaire comme les machines virtuelles JAVA.

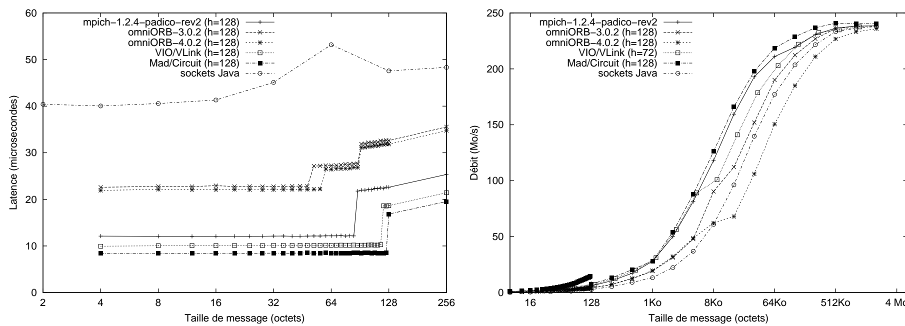

FIG. 3.3 : *Comparaison de MadIO et Madeleine sur Myrinet-2000 (à gauche : latence ; à droite : débit). Les processeurs sont des Pentium III à 1 GHz.*

3.4.1 Évaluation des performance

L'analyse détaillée des performances de PadicoTM est présentée dans la thèse d'Alexandre Denis. Nous nous contenterons ici d'en dégager les grandes tendances.

Comme le montre la figure 3.3, les hautes performance sont au rendez-vous. Les exécutifs évalués sont capable d'avoir des latences faibles et d'exploiter totalement la bande passante du réseau. La figure 3.4 détaille les coûts des différentes couches logicielles : le surcoût apporté par PadicoTM est négligeable.

Les résultats sur réseaux longue distance sont conformes aux résultats déjà publiés. L'utilisation de flux parallèle permet de totalement utiliser la bande passante offerte. C'est une technique particulièrement intéressante pour les environnements multithread qui présentent des débits peu importants sur ces réseaux. De même, l'utilisation du module de compression à la volée améliore le débit.

Ainsi, il est possible de mettre en œuvre une plate-forme de découplage entre les exécutifs et les réseaux afin d'offrir la transparence d'accès au réseau sans introduire de surcoût significatif. Ce résultat n'a été possible non seulement car le modèle s'y prêtait mais également par l'utilisation de stratégie d'implémentation n'introduisant pas de recopie de donnée à tous les niveaux.

Il est important de noter que ces résultats ont été obtenus sans recompilation des exécutifs pour des réseaux différents (Myrinet-2000, SCI, Ethernet, ...) et pour des méthodes de communications différentes. C'était un objectif important pour aller vers la transparence d'utilisation des grilles et de

Kaffe
30
omniORB
8.2 – 9.7
MPICH
3.7 – 4.2
VLink
2 – 2.5
Circuit
0.4 – 0.5
NetAccess
SysIO
4 – 6
MadIO
0.25
sockets TCP/IP
Ethernet–100
72
Madeleine
Myrinet–2000
5.5 – 7.8

FIG. 3.4 : *Coût logiciel et matériel estimé de chaque module lors de l'envoi de messages ; les temps sont donnés en microsecondes, sous la forme d'un intervalle de valeurs constatées selon les conditions. Ces temps comprennent le coût d'émission et de réception additionnés. Les processeurs sont des Pentium III à 1 GHz*

leurs ressources réseaux hétérogènes.

3.5 Bilan et perspectives

L'architecture pour une plate-forme d'intégration d'exécutifs communicant que nous avons proposée est fondée sur la reconnaissance de l'existence de plusieurs paradigmes de communications. Les couches d'arbitrage et de personnalité sont quant à elles plus classiques. Elles sont cependant vitales pour supporter les exécutifs existant sans modification et offrir de haute performance en exploitant au mieux les interfaces réseaux existantes.

Cette architecture a été pensée pour être implémentée au niveau utilisateur. Cependant, au vu du rôle de cette plate-forme, la question se pose si elle n'aurait sa place au sein des systèmes d'exploitation. Cela serait le cas si les réseaux haute performance cesserait de cour-circuiter les systèmes d'exploitation pour réduire au maximum la latence.

La mise en œuvre de la plate-forme PadicoTM a permis d'expérimenter les modèles de programmation présentés dans le chapitre précédent, en particulier PaCO++. Alors que sans PadicoTM, il faut s'assurer que les intergiciels puissent coopérer – ce qui est difficile car nous avons remarqué que des erreurs peu fréquentes mais fatales se produisent – l'utilisation de PadicoTM supprime ce problème.

L'utilisation de PadicoTM n'est pas d'un abord facile. Le premier incon-

vénient vient de la recompilation de tous les exécutifs et applicatifs induits par l'utilisation de Marcel. Il sera résolu lorsque Marcel implémentera l'ABI apparue récemment sous Linux. Le second inconvénient provient de l'utilisation de modules alors que la plupart des utilisateurs sont habitués à manipuler un exécutable. Les modules existent pour deux raisons. D'une part, les lanceurs pour réseaux haute-performance demandent de lancer tous les processus de manière synchrone et statique. L'introduction de la dynamicité dans les lanceurs permettrait non seulement de simplifier le lancement mais surtout de supporter les applications dynamiques et les architectures dynamiques. Cette dynamicité devra être supportée car elle est assez intrinsèque aux grilles et aux applications visées. Cependant, elle est du ressort de Madeleine et non de PadicoTM.

D'autre part, nous avions la volonté de concevoir PadicoTM avec une architecture orientée composant car nous sommes convaincu des avantages qui en découlent. Cependant, nous devons attendre le développement et la diffusion à grande échelle d'un modèle de composant supportant la haute performance. Le modèle de composant Fractal apparaît comme un modèle possible notamment lorsqu'une implémentation en `C` sera disponible.

Nous pouvons remarquer qu'une plate-forme comme PadicoTM ne fait que permettre un accès efficace au réseau. Il faut encore que les intergiciels et les applications soient correctement implémentées pour obtenir une application haute performance. Ainsi, sur toutes les implémentations CORBA que nous avons testés, une seule permet d'obtenir de haute performance car elle implémente en particulier une stratégie de zéro-copie.

La recherche de haute-performance avec plus intergiciels basés sur les paradigmes de communication RPC ou RMI nous a montré une limitation. Actuellement, les données reçues lors d'un appel RPC ou RMI sont allouées par l'intergiciel sans possibilité d'intervention de l'utilisateur. Une limitation apparaît lorsque certains paramètres de l'appel doivent être insérés dans une donnée locale, comme cela se produit lors de redistribution de donnée. Il s'en suite une copie de l'utilisateur. Il faudrait que l'utilisateur ait un moyen de contrôler l'allocation de certains paramètres. Cependant, ce contrôle ne suffirait pas à enlever la copie. En effet, il reste un problème concernant la sémantique d'un appel distribué. Classiquement, les paramètres ne survivent pas à la fin de la méthode invoquée. Or, il est assez fréquent que des paramètres doivent survivre à la fin de l'invocation d'une opération. Dans ce cas, une copie est quand même indispensable.

Si l'architecture proposée et son implémentation permettent une grande souplesse et dynamicité dans la construction et la modification des assemblages, il manque la liaison de ces fonctions avec un environnement d'adap-

tation. Les adaptations peuvent concerner le choix du réseau et/ou des paramètres en fonction de la variation des caractéristiques du réseau et/ou des pannes. Mais, elles peuvent aussi tenir compte de la priorité des communications. Par exemple, les communications reliées au service de log peuvent être moins prioritaires que celles faisant progresser le calcul. La définition de telles politiques ainsi que leur mise en œuvre ne semble pas très évidente à court terme.

Chapitre 4

Déploiement automatique d'applications sur grilles informatiques

Sommaire

4.1 Introduction . . . 62
4.1.1 Services offerts par les grilles informatiques . . . 63
4.1.2 Applications et grilles informatiques . . . 63
4.2 Vers un déploiement automatique . . . 64
4.2.1 Une architecture de déploiement automatique . . . 64
4.2.2 Descriptions spécifiques des applications . . . 66
4.2.3 GADe : une description générique d'applications 70
4.2.4 ADAGE : une implémentation de l'architecture . . . 74
4.2.5 Discussion . . . 75
4.3 Une description des réseaux pour le déploiement . . . 76
4.3.1 Introduction . . . 76
4.3.2 Description des ressources réseaux . . . 76
4.3.3 Un modèle de description des topologies réseaux 77
4.3.4 Mise en œuvre . . . 78
4.3.5 Discussion . . . 79
4.4 Utilisation de Gamma pour l'exécution de workflow . . . 80
4.4.1 Des workflow scientifiques . . . 80
4.4.2 Un modèle d'exécution basé sur Gamma . . . 81
4.4.3 Discussion . . . 82
4.5 Bilan et perspectives . . . 83

4.1 Introduction

L'analogie des grilles avec le circuit de distribution de l'électricité correspond à un certain un graal de l'informatique. Il s'agit d'exprimer un calcul sans dépendance avec le matériel sur lequel il va être exécuté. Dans le chapitre 2, un modèle de programmation basé sur les composants logiciels a été présenté. Sans être totalement indépendant du matériel – un composant doit être compilé pour chaque architecture visée, la démarche est d'augmenter le niveau d'abstraction. Le chapitre suivant a présenté notre démarche pour rendre les composants – et plus généralement toute application – indépendants des technologies réseaux tout en permettant de haute performance. Ce chapitre est consacré au déploiement, c'est à dire à la phase associant une application aux ressources.

Les intervenants principaux de ce chapitre sont au nombre de trois : la description de l'application, la description des ressources et l'appariement d'une application à des ressources. Il s'agit pour un utilisateur ayant implémenté son application de la décrire de manière adéquate afin de la soumettre à un système d'exécution de grille. Ce système doit ainsi accéder à la description des ressources afin d'en choisir certaines pour les associer à l'application. Les phases suivantes comprennent le transfert des fichiers, la configuration et l'exécution proprement dite de l'application.

Nous avons dans un premier temps doublement restreint notre cadre d'étude. D'une part, nous avons considéré des applications statiques, c'est à dire des applications ne cherchant pas à allouer des ressources à l'exécution. Toutes les ressources nécessaires peuvent ainsi être connues et allouées avant que l'exécution commence. D'autre part, nous considérons des applications fermées, c'est à dire ne cherchant pas à communiquer avec d'autres applications ou services. Ces restrictions ont été motivées par le soucis de définir des objectifs raisonnables. Cette activité a été menée pendant le doctorat de Sébastien Lacour [49], que j'ai co-encadré avec Thierry Priol. Tout comme pour les modèles de composants, la dynamicité de l'application ou des dépendances de services sont des objectifs ultérieurs. Ainsi, ce chapitre se termine par une étude de l'utilisation du modèle Gamma pour la modélisation de l'exécution de workflow. Elle a été réalisée par Zolt Németh lors de son post-doctorat co-encadré par Thierry Priol et moi-même.

L'objectif de ce chapitre est de déployer et d'exécuter le plus automatiquement possible une application parallèle et/ou distribuée sur une grille. Avant de présenter nos travaux, nous présentons une analyse de l'état des lieux.

4.1.1 Services offerts par les grilles informatiques

Les intergiciels de grilles constituent un sujet de recherche extrêmement actif. Les services de base que de tels intergiciels doivent fournir sont déjà bien identifiés. Ainsi, il est admis qu'un intergiciel de grille doit gérer la sécurité (Authentification, Autorisation, Accouting), la description des ressources, le transfert de fichier (exécutables et données) et le lancement de processus à distance [29]. Ces services et l'architecture des grilles sont standardisés par le Global Grid Forum [28]. Les exemples d'intergiciels les plus connus sont Globus Toolkit, Unicore, et EGEE.

En ce qui concerne nos préoccupations, nous ignorerons la sécurité, non car elle n'est pas importante – elle est même vitale – mais car notre problématique en est relativement indépendante. Nous supposons que les services de description de ressources, de transfert de fichiers et de lancement de processus sont sécurisés.

Concernant les services offerts, nous supposons qu'il est possible de faire de la co-réservation de ressources. Bien que la plupart des intergiciels n'offrent actuellement pas cette fonctionnalité, les travaux en cours sur le thème laissent penser qu'elle sera bientôt commune [57].

4.1.2 Applications et grilles informatiques

Les grilles informatiques ont pour vocation d'exécuter tous les types d'applications. Cependant, nous pouvons constater que la plupart se focalisent sur un seul type d'utilisation à la fois et ont tendance à ignorer d'autres utilisations de la grille. Ainsi, de manière synthétique, nous pouvons distinguer deux sortes d'utilisation, la première ne permet à un utilisateur que d'appeler des applications pré-installées alors que la deuxième laisse un utilisateur déployer et exécuter ses propres applications.

La première approche est basée sur le modèle de l'appel de services. En effet, les grilles mettent à disposition des ressources, ces ressources peuvent être des services. Trois tendances différentes peuvent être notées bien que leurs différences semblent plus technologiques que fondamentales. Tout d'abord, les travaux autour du GridRPC [67] visent à permettre l'invocation d'une opération sans connaître sa localisation mais en ayant des méta-informations permettant d'estimer le coût d'invocation afin de choisir automatiquement la meilleure instance du service. Ensuite, les travaux autour des Web Services mettent en avant la programmation par "composition" de services sans état. Enfin, le calcul global, en général basé sur les technologies pair-à-pair, semble viser le même type de programmation, bien que les travaux actuels soient surtout basés sur le paradigme maître-travailleur.

Ces trois tendances ont un commun qu'elles assument que les programmes sont déjà déployés et disponibles sous forme de services. Leurs problématiques principales sont la description des services, notamment en terme de méta-informations et de mises en places d'ontologie, la sécurité et le passage à l'échelle.

La deuxième approche consiste à voir une grille comme un service de gestion de ressources de calcul, c'est à dire comme un service de batch. C'est une vue principalement issue de l'utilisation des machines parallèles où de nombreux utilisateurs veulent exécuter leur application sur un nombre limité de ressources. Ainsi, les services offerts par la grille permettent de consulter l'état des ressources, de soumettre des tâches, de transférer des fichiers, etc. La conséquence est que le déploiement d'une application sur une grille est une tâche ardue : pour déployer une application, il faut en connaître sa structure, rechercher des ressources adaptées, et calculer un placement. Les phases de lancement et de transfert de fichiers sont raisonnablement automatique à condition d'écrire les fichiers requis.

4.2 Vers un déploiement automatique

Le succès des grilles dépend en grande partie de la simplicité de leur utilisation. Dans la vision originelle des grilles, un utilisateur n'a qu'à soumettre son application. C'est au système de grille de déterminer les ressources à affecter à son exécution. Afin de réaliser cette vision de simplicité d'utilisation – et surtout de déployer simplement les modèles de programmation et de support réseaux décrits dans les chapitres 2 et 3, nous avons démarré une activité de recherche visant à définir un modèle permettant d'automatiser le processus de déploiement.

4.2.1 Une architecture de déploiement automatique

La première étape a consisté à définir l'architecture générale afin de déterminer les éléments entrant en jeu dans la phase de déploiement ainsi que leur relations. Nous nous sommes fixés des contraintes pour simplifier le problème mais qui sont néanmoins compatibles avec les applications envisagées. La contrainte principale a été de ne viser que des applications *statiques*, c'est à dire des applications dont tous les constituants sont connus et déployables lors du lancement de l'application. De plus, nous ne nous intéressons pas non plus à la dynamicité des ressources durant l'exécution de l'application comme la tolérance aux pannes ou la variation

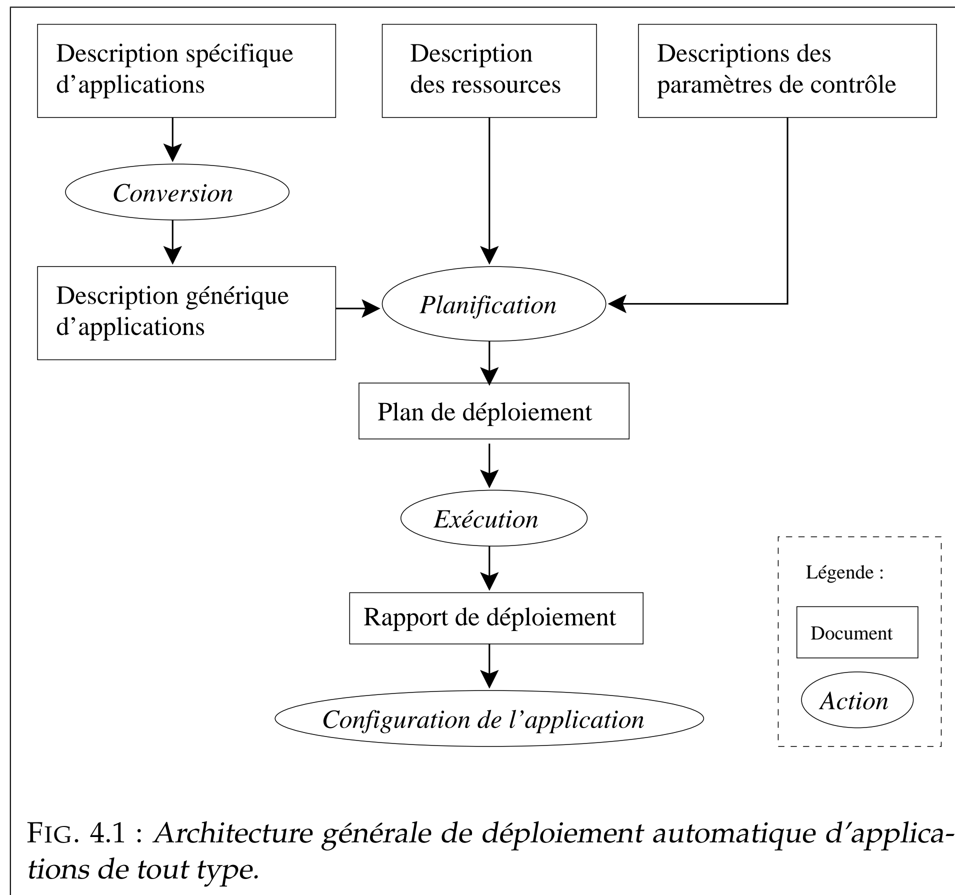

FIG. 4.1 : *Architecture générale de déploiement automatique d'applications de tout type.*

des débits réseaux. Cependant, afin de supporter les applications multi-intergicielles comme GridCCM, nous supposons *a priori* tout type d'applications. Concernant l'accès aux ressources, nous supposons que la grille offre des services d'information et de co-allocation de ressources.

La figure 4.1 synthétise notre proposition d'architecture générale de déploiement automatique. Elle est constituée de documents et d'actions. Les documents en entrée sont la description de l'application à déployer dans le formalisme du modèle de programmation de l'application, la description des ressources (ou une référence vers le service d'information) et enfin des paramètres de contrôle, qui permettent d'agir sur la planification en spécifiant par exemple quelle(s) métrique(s) l'algorithme de placement doit prendre en compte (temps de calcul, coût de calcul, etc). Afin de rendre l'algorithme de planification indépendant du type d'application, la description de l'application est convertie en une description générique. Notre pro-

position de description générique ainsi que la conversion d'une description spécifique en cette description générique est abordée dans la section 4.2.3.

Toutes les informations sont alors disponibles pour le cœur du processus de déploiement, c'est à dire le planificateur qui calcule un plan de déploiement. C'est durant cette phase que toutes les décisions doivent être prises comme la sélection les ressources de calcul, de communication, de stockage, le placement des constituants de l'application, la sélection des implémentation des constituants de l'application, etc.

Le plan de déploiement peut alors être exécuté : il s'agit d'effectuer les ordres qu'il contient comme le transfert des fichiers et le lancement des processus. Un rapport de déploiement, contenant par exemple l'identifiant des processus lancés, est produit afin de permettre par exemple à d'autres outils de contrôler l'exécution de l'application ou d'annuler son exécution.

Le dernier élément de l'architecture générale est la phase de configuration de l'application. En effet, de nombreux modèles de programmation réclament de configurer l'application par rapport à l'environnement. Les implémentations MPI [32] pour la grille demandent de connaître la topologie réseaux afin d'optimiser les opérations collectives ; les modèles de composants ayant un langage de description d'architecture demandent de créer les composants, de configurer les attributs des composants et enfin de connecter les composants.

Nos travaux se sont concentrés sur les descriptions spécifiques et génériques des applications, sur la description des ressources réseaux et sur la réalisation d'un prototype. L'objectif était de disposer de suffisamment d'éléments pour juger de la pertinence de l'approche. Une autre motivation est que cette architecture découple les différentes étapes du déploiement et en particulier isole le planificateur. Elle permet ainsi d'intégrer des algorithmes de planification mis au point par d'autres équipes à moindre coût. Cette section se poursuit d'abord par la présentation de nos travaux sur les descriptions spécifiques et génériques d'applications et d'ADAGE, notre prototype implémentant cette architecture. Elle se termine par une discussion sur ces travaux. Nos travaux sur la description des ressources réseaux sont présentés à la section 4.3 principalement car ils sont indépendants de cette architecture bien que motivés par elle.

4.2.2 Descriptions spécifiques des applications

Notre proposition d'architecture de déploiement prend en entrée une description de l'application à déployer. Cette description doit contenir deux types d'information. Premièrement, elle doit informer sur la structure lo-

gique de l'application (nombre de constituant, connexions éventuelles entre eux, etc.). Par exemple, pour les applications à base de composants il s'agit d'un ADL. Deuxièmement, elle doit également contenir des informations décrivant les implémentations comme la localisation des binaires, les contraintes d'exécution (CPU, OS, quantité mémoire, etc).

Il nous paraît important de pas inclure dans la description de l'application des informations relatives à l'exécution. Ce type d'information est pris en charge dans notre modèle par les paramètres de contrôle. Cette séparation permet de bien forcer le découplage entre les applications et les ressources. Ainsi, le développeur d'une application spécifie la structure de son application ainsi que les contraintes opérationnelles de son implémentation dans la description de l'application. La personne qui exécute l'application spécifie ses contraintes d'exécution dans les paramètres de contrôles sans devoir modifier la description de l'application.

La description d'une application est spécifiée dans un formalisme dépendant du modèle de programmation. Les modèles de composants sont généralement dotés d'un langage de description d'architecture (ADL). Ce n'est pas le cas pour les applications MPI. En général, la description d'application MPI – quand elle est disponible – est sous forme textuelle voir informelle. Pour une application MPICH-G2, sa description se retrouve dans le fichier RSL qui alors mélange des informations de structure de l'application avec une exécution particulière de l'application.

Les applications visées par GridCCM utilisant MPI, nous nous sommes intéressé à définir d'abord une description pour MPI puis pour GridCCM. Nous en présentons ci-après les principes. La description complète est disponible [49].

MPI

Afin de décrire une application MPI, un certain nombre d'informations est utile. Ces informations peuvent être rangées en deux catégories. La première catégorie regroupe les informations relatives aux programmes qui constituent l'application comme la liste des programmes[1], leur localisation, les contraintes de processeur et de systèmes d'exploitation, les dépendances vis-à-vis d'autres bibliothèques, etc. La deuxième catégorie concerne le nombre d'instances à déployer, la description des groupes de processus qui seront amenés à communiquer intensivement au sein de communicateur

[1]Le plus souvent une application MPI est constituée d'un seul programme (SPMD) mais la norme définit également le cas de plusieurs programmes (MPMD)

```
<MPI_Application>
  <programs>
    <program id="master_prg">
      <binaries>
        <binary vendor_impl="MPICH-G2" bin_type="exec">
          <location>ftp://ftp.exe-store.org/master.exe</location>
          <targets><!-- CPU and OS constraints--></targets>
        <binary>
      </binaries>
    </program>
    <program id="worker_prg">...</program>
  </programs>
  <application>
    <world_size var="n">n>8+1 && n=i**2+1 && n<1025</world_size>
    <program_instances>
      <program_instance id="the_master" ref_id="master_prg">
        <cardinality var="x">x=1</cardinality>
      </program_instance>
      <program_instance id="the_worker" ref_id="worker_prg" />
    </program_instances>
    <partitioning>
      <group id="worker_group" ref_id="the_worker">
      <cardinality><max_value>4</max_value></cardinality>
      <partitioning>
        <group id="worker_subgroup1">
          <size<max_value>2</max_value></size>
          <cardinality>
            <min_value>1</min_value><max_value>1</max_value>
          </cardinality>
        <group>
        <group id="worker_subgroup2">...<group>
      </partitioning>
      </group>
    </partitioning>
    <topologies>
      <topology ref_id="worker_subgroup1" type="cartesian">
        <dimensions>2</dimensions>
        <list_of_sizes>4,3</list_of_sizes>
      </topology>
    </topologies>
  </application>
</MPI_Application>
```

FIG. 4.2 : *Exemple de description d'application MPI avec un seul processus* `master`, *un nombre carré de processus* `worker` *divisé en au plus 4 groupes d'exactement 2 processus.*

MPI, et la description des schémas de communication entre processus par le biais de la topologie virtuelle au sein des communicateurs.

Notre modèle de description d'applications MPI s'inspire du format de description des applications CCM et du format OSD[2] [74] pour la présentation des informations. La figure 4.2 en présente un exemple. Cette description contient deux parties. La première partie contient les informations relatives aux programmes. Elle représente les *types* des programmes qui peuvent être déployés. La seconde partie contient les informations de la deuxième catégorie introduit ci-dessus. Elle représente la structure logique de l'application MPI.

Nous avons retenu trois types d'information pour décrire la structure de l'application. Tout d'abord, il s'agit de décrire le nombre total de processus ainsi que les instances des programmes avec également des contraintes de cardinalité. Ensuite, les différents types de partitionnement possibles sont décrits. Enfin, les contraintes topologiques sont précisées. Elles permettent d'exploiter le fait que MPI permet de décrire des topologies virtuelles cartésiennes ou quelconques.

GridCCM

La définition du modèle GridCCM nous a amené à définir un modèle pour décrire les applications GridCCM. Le travail a donc consisté à étendre le modèle CCM pour des implémentations de composants parallèles sans encoder dans le modèle un liste de technologies parallèles. Ainsi, une description GridCCM diffère d'une application CCM uniquement via la spécification des implémentations dans la description des composants (fichier `csd`). La description d'une implémentation séquentielle reste la même que dans CCM, via la balise `implementation`. La description d'une implémentation parallèle est réalisée via la balise `GridCCM_implementation` comme illustré à la figure 4.3.

La balise `GridCCM_implementation` regroupe la description des trois éléments constituant un composant GridCCM : la description du programme parallèle et la description des deux gestionnaires de connections. En référençant des fichiers externes, la description d'une application GridCCM est ainsi totalement découplée de la description du programme parallèle. Dans le cas d'une implémentation MPI, il suffit de générer un fichier de description de programme MPI comme introduit précédemment. L'évolution du formalisme MPI ou l'utilisation d'une autre technologie pa-

[2]Open Software Description

```
<softpkg>
  ...
  <!-- Une implémentation séquentielle d'un composant GridCCM -->
  <!-- conforme aux spécification de CCM -->
  <implementation >
  </implementation >

  <!-- Une implémentation parallèle de ce composant GridCCM -->
  <!-- utilisant MPI -->
  <GridCCM_implementation type="MPI" id="impl_mpi">
    <functional_program>
      <location>http://www.store.org/FFT.mpi</location>
    </functional_program>

    <connection_managers>
    <input_connection_managers>
      <location>http://www.store.org/input-FFT.csd</location>
    </input_connection_managers>
    <output_connection_managers>
      <location>http://www.store.org/output-FFT.csd</location>
    </output_connection_managers>
    </connection_managers>

  </GridCCM_implementation >

</softpkg>
```

FIG. 4.3 : *Exemple de description de l'implémentation d'un composant GridCCM (extrait d'un fichier* `csd`*).*

rallèle n'impacte pas la description d'une application GridCCM.

Les gestionnaires de connexions sont implémentés comme des composants séquentiels. Il est ainsi possible de les décrire à l'aide du formalisme proposé par CCM (fichier `csd`).

4.2.3 GADe : une description générique d'applications

La section 4.2.1 a présenté notre proposition d'architecture générale pour le déploiement automatique d'applications. Afin de supporter tout type d'application – et en particulier les applications multi-modèles de programmation comme GridCCM, la solution retenue est de convertir les descriptions spécifiques d'application en une description générique. La conversion dans un même formalisme de la description d'applications MPI

et CCM a deux avantages. D'une part, elle représente une solution simple pour supporter des applications multi-modèles à condition d'avoir une opération d'agrégation simple des différentes descriptions génériques obtenues. D'autre part, elle permet de découpler le planificateur du type d'application. Les algorithmes de planification efficaces étant très difficiles à mettre au point, il y a tout intérêt à pouvoir les réutiliser pour tout type d'application. Notre postulat implicite est que la difficulté de réalisation d'un planificateur est fondamentale – calculer un plongement d'un graphe valué représentant l'application dans un graphe valué représentant les ressources de la grille. Elle ne dépendrait donc pas du type d'application.

Modèle de description générique d'applications

Le modèle de description générique – nommé GADE (***G**eneric **A**pplication **De**scription*) – calque le modèle d'exécution des systèmes d'exploitation modernes : il se fonde sur le principe que toutes les applications se résument à un ensemble de processus, quels que soient leurs types. Ces processus peuvent contenir des DLL. De plus, il peut y avoir des contraintes de localisation entre processus ou entre DLL afin de partager un même banc mémoire, un même système de fichier local, etc.

GADe définit quatre entités : les processus, les codes à charger (les DLL), les groupes de processus et enfin les connexions entre groupes de processus. Les processus, les codes à charger et les groupes de processus sont organisés via une relation d'inclusion : les groupes de processus contiennent des processus qui peuvent contenir des codes à charger.

Un groupe de processus représente un ensemble de processus qui doivent s'exécuter au sein d'un même système d'exploitation, typiquement un nœud d'une grappe. Un processus a la même signification que dans les systèmes d'exploitation. Enfin, un code à charger représente une entité qui peut être placée dans un processus comme une DLL ou une classe JAVA. Ces trois entités ont une cardinalité associée : elle décrit le nombre de fois qu'il faut dupliquer l'élément. Par exemple, un processus avec une cardinalité de quatre représentera quatre fois le même processus à l'exécution.

Un groupe de processus et un processus ont une double signification. D'une part, ils représentent des constituants de l'application à placer et d'autre part ils représentent un conteneur. Ainsi, les groupes de processus contiennent de processus et les processus peuvent héberger des codes à charger. Les processus et les codes à charger étant des éléments concrets, la description de leur implémentation en terme de processeur, systèmes d'exploitation, dépendances de bibliothèques font parties de la description gé-

```
<proc_groups>
 <proc_group id="A">
   <cardinality var="n">n=2*i && i>1</cardinality>
   <processes>
     <process id="process_1">
       <cardinality>1</cardinality>
       <implementations> ...</implementations>
       <codes_to_load>
         <code_to_load id="thread_1">
           <cardinality>1</cardinality>
           <implementations> ...</implementations>
         </code_to_load>
       </codes_to_load>
     </process>
   </processes>
 </proc_group>
</proc_groups>
```

FIG. 4.4 : *Description générique d'une application représentant un groupe de processus contenant un seul processus qui à son tour contient un code à charger. La cardinalité du groupe de processus peut être une expression : dans ce cas elle décrit qu'il faut répliquer un nombre pair de fois ce processus. C'est au planificateur – éventuellement aider des paramètres de contrôle – d'instantier* `n`.

nérique. La figure 4.4 illustre ces trois entités.

La quatrième entité de GADe est la connexion (orientée) entre groupes de processus. Elle exprime des contraintes de communications. Ainsi, le planificateur devra s'assurer ou mettre en œuvre des mécanismes pour que la communication soit possible entre ces groupes de processus. Comme illustré en figure 4.5, la connection peut s'appliquer à plusieurs groupes de processus en distinguant pour chacun d'entre eux le rôle – émetteur, récepteur ou les deux. Cette information est pertinente vis à vis des pare-feux. L'information est portée par les groupes de processus et non par les processus car comme tous les processus d'un groupe sont co-localisés sur une même machine, si l'un d'entre eux peut communiquer alors les autres également. Ainsi, le modèle permet de compacter l'information. Nous avons choisi de ne pas distinguer entre les différents types de réseaux car nous assumons le découplage entre les intergiciels et les réseaux comme présenté au chapitre précédent. Cependant, les connexions peuvent porter des

```
<connections>
 <connection id="cnx_10">
   <ref_id role="source">A</ref_id>
   <ref_id role="destination">B</ref_id>
   <ref_id role="both">C</ref_id>
 </connection>
</connections>
```

FIG. 4.5 : *La description des communications entre groupe de processus s'exprime via le concept des connections. Le sens des communications est indiqués. Ainsi, le groupe de processus d'identifiant* `B` *peut recevoir des communications de* `A` *et* `C`*.*

contraintes sur les propriétés réseaux comme par exemple la latence ou le débit.

Conversion d'une description spécifique vers une description générique

L'introduction d'une description générique d'application amène la problématique de la conversion d'une description spécifique en une description générique. Cette description doit être réalisée pour chaque type d'application.

MPI La conversion d'une description MPI en une description générique est simple si nous ne prenons pas en compte les balises de partitionnement. En effet, une application MPI constituée de n processus (dans la description spécifique) se traduit en un groupe de processus de même cardinalité contenant un unique processus. Tout processus MPI pouvant envoyer et recevoir un message de tout autre processus, l'élément connection de la description générique référence le groupe de processus avec un rôle de source et de destinataire.

Les informations de partitionnement des applications MPI sont actuellement pris en compte lors de la phase de conversion. Ceci n'est pas satisfaisant car d'une part ces informations devraient être gérées par le planificateur et d'autre part le convertisseur n'a pas accès aux ressources pour décider de manière pertinente.

CCM La génération d'une description générique application CCM est assez directe – le modèle générique ayant été motivé par GridCCM – mais est

assez longue à décrire. En résumé, les composants sont projetés vers des processus ou des codes à charger. Dans ce dernier cas, le binaire du processus est implicitement un serveur de composant. Les contraintes de colocalisations entre composants sont supportées en les regroupant au sein d'un même groupe de processus (`host colocation`) ou d'un même processus (`process colocation`). Enfin, les connections entre composants se projettent directement sur la balise connection de la description générique.

GridCCM La conversion d'une application GridCCM ne pose pas de problème particulier. Elle peut être vue comme l'application de la conversion CCM pour la description de l'assemblage de composants et de la conversion MPI pour l'implémentation de composants parallèles MPI. Seules les contraintes de localisation intra-processus demandent une attention particulière car elles demandent d'une part que la description de l'implémentation parallèle soit sous forme de DLL et d'autre part que le lancement de ces processus soit bien défini. Ainsi, la co-localisation au sein d'un même processus d'un programme MPI et d'un programme PVM n'est pas évident sans une technologie telle que PadicoTM.

4.2.4 ADAGE : une implémentation de l'architecture

ADAGE (*Automatic Deployment of Applications in a Grid Environment*) [49] est notre prototype de recherche qui implémente notre proposition d'architecture générale pour le déploiement automatique d'applications sur des grilles informatiques. En particulier, il support le modèle de description générique d'application GADe.

ADAGE supporte des descriptions d'applications MPI (MPICH1-P4 et MPICH-G2), CCM et JXTA [82]. Le support d'application GridCCM est en cours de finalisation. Les convertisseurs représentent environs 500 lignes de code C++ pour MPI et JXTA et quasiment 1400 lignes pour CCM. Cette différence est due au plus grand nombre de balises des DTD et de cas à considérer pour CCM. L'expérience a montré que c'était du code assez simple à réaliser : le support initial de JXTA dans ADAGE a été réalisé en deux jours par Mathieu Jan sans connaissance préalable d'ADAGE. Concernant la description des ressources, ADAGE supporte soit un fichier XML accesible au travers une URL soit le système d'information de Globus (MDS2). Dans tous les cas, une limitation actuelle d'ADAGE est qu'il construit en mémoire la description complète des ressources de la grille.

Deux algorithmes de placement ont été implémentés, le premier calculant une placement avec une stratégie de tourniquet et le deuxième utilisant une fonction aléatoire de placement. Les deux algorithmes vérifient les contraintes de processeurs et de systèmes d'exploitation. Lors de l'exécution du plan de déploiement, ADAGE lance les processus via GT2 ou via les commandes rsh/ssh. Par manque de temps, le transfert des fichiers n'a pas encore été implémenté.

Concernant le passage à l'échelle d'ADAGE, des expériences préliminaires ont été réalisés avec le déploiement de 12 000 pairs JXTA sur 160 machines de Grid5000 à Orsay et de 580 pairs sur autant de machines sur 6 sites de Grid5000 (Bordeaux : 44 machines, Orsay : 185 machines, Toulouse : 50 machines, Sophia : 100 machines, Lyon : 40 machines et Rennes : 161 machines).

4.2.5 Discussion

Nos travaux sur le déploiement d'applications sur des grilles informatiques ont mis en évidence qu'il était possible de séparer la description des applications de la description des ressources. De plus, ils ont montré que la conversion en une description générique avait de bonnes propriétés comme le support très facile de nouveaux modèles de programmation ainsi que le support de modèles multi-intergiciels comme GridCCM.

Cependant, nous pouvons noter quelques limites. D'une part, la généralité de la description générique du point de vue des contraintes réseaux n'est pas encore montré comme nous l'avons vu avec la conversion des applications MPI. D'autre part, le modèle actuel prend mal en compte les possibilité de choix multiples. Par exemple, la description générique ne permet pas d'exprimer le fait qu'un composant est disponible sous forme de processus et de codes à charger. Cependant, ces limitations devraient être levées en raffinant le modèle générique.

Deux travaux importants restent encore à réaliser. Premièrement, il s'agit de réaliser un ou des planificateurs efficaces. Cette tâche étant connue pour être très complexe, nous avons commencé à collaborer avec des équipes spécialisées dans le placement. Deuxièmement, notre architecture ne supporte actuellement que des applications statiques. C'est pourquoi Boris Daix vient de débuter un doctorat sur le déploiement d'applications dynamiques. Il est co-encadré par Christine Morin et moi-même. Son travail devrait également aborder les relations entre le déploiement et les systèmes d'exploitation. En effet, il est fort probable qu'une partie des fonctionnalités nécessaires pour le support du déploiement dynamique devrait être fournie

par les services de base des grilles informatiques.

4.3 Une description des réseaux pour le déploiement

4.3.1 Introduction

Comme nous venons de le voir, une description pertinente des ressources est très importante pour un placement de l'application satisfaisant les contraintes d'exécution. La description des nœuds d'une grille a été l'objet de nombreuses attentions et est assez bien maîtrisée actuellement. Ces descriptions concernent le type de processeur, la mémoire, les disques durs, le système d'exploitation, etc. La description des réseaux n'a pas vraiment connu la même attention alors que de nombreuses contraintes peuvent s'y référer. Par exemple, un utilisateur peut demander une grappe distante d'au plus 100 ms d'une machine de visualisation. Il est donc important que la description des ressources d'une grille décrivent les informations statiques (ou quasi-statiques) concernant les réseaux comme la topologie mais également les informations dynamiques comme le débit ou la latence courante.

4.3.2 Description des ressources réseaux

La description des réseaux a fait l'objet de travaux comme par exemple RSD, Remos, GridLabMDS, ENV, GridG, RGIS, TopoMon, etc. Cependant, ces travaux présentent tous des limitations comme le non-support des réseaux non IP, la restriction à des réseaux hiérarchiques, la non prise en compte des pare-feux, et souvent une faible capacité à supporter de grands système car les descriptions sont au niveau du lien. Or, ces propriétés apparaissent importantes car elles sont présentes dans les grilles.

Réseaux non IP Les machines parallèles et les grappes sont souvent équipées de réseaux locaux haute performance (Myrinet, SCI, Infiniband, ...). Ces réseaux sont en général exploités avec un protocole spécifique afin d'offrir le plus de performance.

Réseaux non-hiérarchiques Les grilles sont parfois construites au dessus d'un réseau longue distance privé à la grille comme le réseau TeraGrid aux USA, ou le réseau Renater en France dont une partie du trafic est réservée aux communications Grid'5000. Or, avec la présence d'Internet, le réseau global de la grille n'est plus hiérarchique. De plus, au niveau des grappes de grappes, il peut exister différentes

routes entre des machines, typiquement via le réseau Ethernet et via un réseau haute performance.

Pare-feu, NAT et liens asymétriques La plupart des ressources d'une grille sont protégées par des pare-feux. Ainsi, il est important de connaître leur localisation ainsi que leur politique de filtrage pour déterminer si une application sera capable de communiquer ou s'il faut mettre en place des tunnels entre certains nœuds de la grille. Il en va de même pour les réseaux utilisant des adresses privées qui accèdent aux machines distantes via du NAT (Network Address Translation). Enfin, les liens présentent souvent des caractéristiques non symétriques au niveau des débits, et des latences mais aussi en ce qui concerne les politiques de filtrage des pare-feux. Il faut donc distinguer le sens de la communication.

Nous avons donc entrepris de définir un modèle de description de réseau supportant en particulier ces propriétés. D'une part, l'existence d'un tel modèle est indispensable pour envisager le déploiement d'applications parallèles distribués. D'autre part, nous pouvons nous concentrer sur la description de la topologie car des travaux comme ceux menés par le Network Measurement Working Group (NMWG) [53] du Global Grid Forum a pour objectif d'identifier et de développer des mécanismes standards pour décrire et publier les métriques réseaux utiles dans une grille.

4.3.3 Un modèle de description des topologies réseaux

L'objectif étant d'avoir une description des réseaux des grilles afin de sélectionner les ressources pour une application, nous avons en fait besoin d'une description fonctionnelle du réseau. Le modèle représente donc une topologie logique du réseau, comme la plupart des travaux du domaine. Cependant, le modèle regroupe ensemble les machines avec des caractéristiques réseaux semblables.

Le modèle étant fonctionnel, il ne cherche pas à représenter tous les commutateurs et routeurs présents dans le réseau. En fait, il se contente de décrire que des machines sont connectées par un même réseau et que ces machines partagent globalement les même caractéristiques vis à vis du réseau. Cet ensemble de caractéristiques constitue un groupe réseau.

Graphe de groupe réseau

La topologie est décrite par un graphe orienté acyclique. Les nœuds du graphe correspondent à des groupes réseau tels que définient ci-dessus ou

à des machines. Les arêtes orientées du graphe symbolisent l'inclusion. Un groupe réseau peut avoir des groupes réseaux fils. Les arcs sont orientés du parent vers un fils. Autrement dit, un groupe réseau fils représente un sous-réseau du groupe réseau parent.

Pour des réseaux hiérarchiques, le graphe obtenu est un arbre avec une racine implicitement définie comme étant le réseau englobant tous les réseaux de la grille. Ce réseau est Internet dans la plupart des grilles.

Pour les réseaux non-hiérarchiques, le graphe devient un demi-treillis – il n'y a pas de minimum – avec plusieurs racines (ou maximum). Afin de rendre le graphe facilement manipulable, nous introduisons un nœud spécial : le nœud racine. Ce nœud, en fait toujours présent dans notre modèle, représente la grille. Ces fils sont les réseaux indépendants de la grille, qui par construction sont les maximums du graphe.

La différence principale entre les autres travaux et notre modèle réside dans la nature des objets qui sont associés aux nœuds et aux arêtes. Les autres travaux associent les machines aux nœuds et les réseaux aux arêtes.

Propriétés réseau

Les propriétés d'un groupe réseau, comme par exemple les bibliothèques réseaux disponibles et les caractéristiques de performance, peuvent être spécifiées comme des attributs des nœuds correspondant du graphe. Si un nœud ne définit pas une propriété, celui-ci hérite par défaut de la valeur de son groupe parent.

Liens asymétriques, pare-feux et NAT

Pour supporter en particulier les liens asymétriques, des exceptions peuvent être déclarées dans un groupe réseau. Les exceptions remplacent totalement ou partiellement les propriétés du groupe réseau. Ainsi, des exceptions peuvent gérer les ports ouverts en entrées et/ou en sortie.

Pour décrire des machines accessible via du NAT, un premier groupe réseau inclut les machines avec leur adresse privée ainsi qu'une ou plusieurs passerelles NAT. Ces passerelles via une association dans un second groupe réseau sont responsables des translations d'adresse et de l'acheminement des messages.

4.3.4 Mise en œuvre

Nous avons mis en œuvre ce modèle dans l'intergiciel de grille Globus Toolkit 2. Cette mise en œuvre fut simple à réaliser : notre choix a

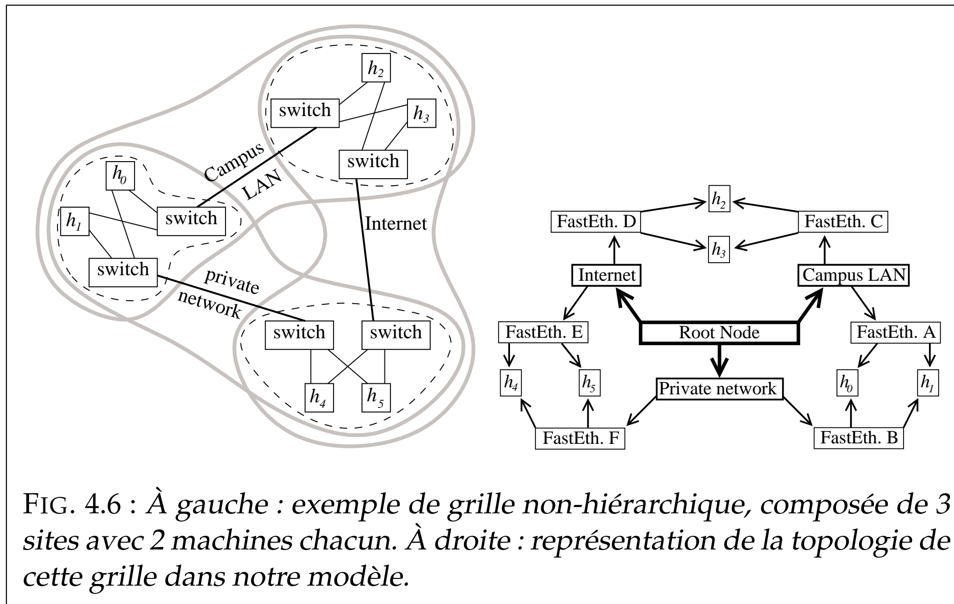

FIG. 4.6 : *À gauche : exemple de grille non-hiérarchique, composée de 3 sites avec 2 machines chacun. À droite : représentation de la topologie de cette grille dans notre modèle.*

été d'étendre le service d'information de Globus (MDS2) en manipulant ses fichiers XML. En fait, le graphe est décrit via un ensemble distribué de fichiers XML dont leur relation suivent les relations du graphe. Chaque groupe réseau est stocké dans un fichier XML géré par MDS2.

4.3.5 Discussion

Dans la perspective où l'utilisateur recherche une exécution efficace de ses applications, il apparaît important de bien maîtriser la description des grilles. Ainsi, un appariement pertinent peut être réaliser entre les contraintes applicatives et les possibilités des ressources. Nous avons proposé un modèle de description de la topologie des réseaux présents dans les grilles informatiques. Cette description est complémentaire des travaux du GGF. Leur intégration dans Globus2 et la description de la grille basée sur VTHD que nous avions à ce moment là a démontré la faisabilité et la pertinence du modèle. En plus d'une certaine simplification, un bénéfice attendu de notre modèle est une meilleure extensibilité car le regroupement des informations conduit à moins d'arêtes. Une étape future consiste à développer des algorithmes de planification prenant en compte ces informations.

4.4 Utilisation de Gamma pour l'exécution de workflow

L'accueil de Zsolt Németh lors d'un post-docorat dans le projet PARIS nous a permis de nous intéresser à l'utilisation du modèle Gamma pour la modélisation de l'exécution de workflows. Ce travail fut motivé par la recherche du passage à l'échelle des modèles de déploiement et une première incursion dans l'aspect dynamique du déploiement.

L'exécution de workflow est un cadre d'étude idéal car c'est un domaine assez bien connu mais sans solution totalement décentralisée. Gamma apporte un modèle bien défini (γ calcul), totalement décentralisé et sans notion d'ordre. Gamma (General Abstract Model for Multiset Manipulation) [7, 8] est un modèle abstrait visant à relâcher la sur-séquentialisation des algorithmes. Il est basé sur la métaphore chimique : des molécules représentent des données et des réactions chimiques des opérations sur ces données. Pour formaliser ce modèle, Gamma utilise un multi-ensemble. Les éléments du multi-ensemble représentent des molécules-données auxquelles sont appliquées des réactions-opérations. Ces réactions sont atomiques, locales et indépendantes. Ainsi, un programme Gamma est intrinsèquement parallèle sans contrôle centralisé et sans ordre d'exécution.

4.4.1 Des workflow scientifiques

Notre travail a volontairement été restreint à l'exécution de workflows scientifiques dans un environnement à très grande échelle et hétérogène tel qu'une grille informatique. L'état de l'art sur les workflows découpe en trois phases le cycle de vie d'un workflow. Tout d'abord, l'application est décomposée en un ensemble d'activités reliées entre elles par des dépendances de contrôle et/ou de données : c'est le workflow abstrait. Ensuite, des ressources sont associées à ces activités pour obtenir un workflow concret. Enfin, ce workflow concret est exécuté.

Contrairement à l'état de l'art qui sépare les phases d'association d'activités et d'exécution proprement dire, nous souhaitons réunir ces deux phases afin de pouvoir supporter des systèmes d'une part dynamiques et volatiles et d'autre part trop grand pour espérer en avoir une connaissance globale. Ainsi, le choix de la ressource peut être être réalisé au dernier moment, être modifié et ce même pendant l'exécution d'une tâche du workflow. Dans la suite, nous employons le mot exécution pour désigner à la fois l'opération d'association et de contrôle de l'exécution.

L'exécution d'un workflow demande de coordonner les activités du

workflow avec les ressources du système. Une telle coordination comprends la découverte et la sélection de ressources, de services et de données, la gestion des dépendances de contrôle et des données des activités et si possible la détection et la tolérance de défaillance. Ces éléments doivent être gérés dans un environnement à large échelle, hautement dynamique et défaillant.

De manière plus précise, les objectifs de notre travail étaient de trouver un modèle déclaratif de haut niveau fournissant un canevas pour la coordination de l'exécution de workflows avec (1) un très haut niveau d'autonomie, (2) la possibilité pour les activités du workflow de modifier l'environnement, (3) la non-centralisation de l'exécution du workflow, (4) la prise de décision sur une connaissance partielle de l'environnement et (5) le support de structures de contrôle avancées dans le workflow.

4.4.2 Un modèle d'exécution basé sur Gamma

Les similitudes entre les besoins exprimés et les possibilités de Gamma nous ont amené à étudier s'il était possible d'utiliser Gamma pour formaliser l'exécution de workflow. Nous décrivons très brièvement le modèle proposé. Les détails de ce travail ont été publiés dans [60, 61].

Gamma réalisant un modèle chimique, nous avons défini une analogie chimique à l'exécution de workflow. Les activités du workflows ainsi que les ressources sont modélisées à l'aide de molécules (les éléments du multiensemble). Les activités sont des molécules réactives contrairement aux ressources qui sont des molécules passives. La forme générale de la solution décrivant une ressource est $\langle a_1 : v_1, a_2 : v_2, ...a_n : v_n \rangle$ où chaque $a_i : v_i$ est une paire d'attribut-valeur. Le choix des attributs dépend des propriétés des ressources que l'on souhaite modéliser. Par exemple, $\langle id : R1,\ type : comp,\ proc : 16,\ OS : Linux,\ mem : 4,\ installed : equsolver, ... \rangle$ modélise une ressource de calcul identifiée par `R1` avec 16 processeurs, Linux comme système d'exploitation, 4 Go de mémoire et un logiciel de résolution d'équation installé. La valeur des attributs étant sans unité, il convient d'établir des conventions, comme par exemple d'exprimer les quantités de mémoire en Go.

Dans notre modèle, un activité est une unité d'exécution atomique qui peut retourner des résultats. L'exécution proprement dite de l'activité n'est pas pertinente pour notre modèle. C'est pourquoi une activité A prêtre à être exécutée est représentée symboliquement par **execute** A **on** $resources$ **using** $parameters$ où **execute** est une procédure externe, dont le comportement en terme de γ-calcul serait

execute A **on** $resource$ **using** $parameters \rightarrow \langle A : result, ...\rangle, \langle id : resource, ...\rangle, ...$ L'exécution d'une activité produit des résultats et rend à nouveau disponible les ressources utilisées durant le calcul.

Les dépendances entre une activité et une ressource sont exprimées suivant la forme générale $\gamma\langle a_1 : v_1, a_2 : v_2, ...a_n : v_n\rangle$.**execute** A **on** v_i où les a_i représentent des propriétés requises. Pour indiquer la non pertinence de certains attributs, nous introduisons le symbole ω comme symbole universel de mise en correspondance. Par exemple, une activité A demandant 4 processeurs et 128 Mo de mémoire et ne posant pas de contraintes sur les autres propriétés comme par exemple le système d'exploitation est représentée par $\gamma\langle id : r, procno : 4, memory : 128, \omega\rangle$.**execute** A **on** r.

La dépendances entre activités sont représentées par une relation binaire sur l'ensemble $A \times A$ où $A = \{A_1, A_2, ...A_k\}$ est la décomposition du workflow en k activités. Les dépendances sont exprimées par un paramètre dans l'entête de la γ-abstraction. Par exemple, si l'activité A_i dépend de l'activité A_j, alors la γ-expression correspondante est $\gamma\langle A_j : x, \omega\rangle$.**execute** A_i **using** x.

Nous avons défini les constructions de séquencement, de fusion synchronisée et de parallèle split. Ces constructions sont celles généralement définies par les modèles de workflows. Cependant, nous avons également défini des constructions plus avancées comme des conditions, le split, le choix exclusif, la fusion, et la boucle d'itération [60]. Il est également possible d'exprimer *simultanément* des dépendances de données et de contrôle afin d'exprimer la capture atomique.

4.4.3 Discussion

Le modèle que nous avons défini permet de modéliser et de contrôler des workflows avec des constructions très avancées dans un même formalisme. Une propriété importante de ce modèle est qu'il permet de concrétiser à la volée le workflow et de supporter des remises en causes. En effet, une erreur d'exécution est modélisée par l'annulation d'une réaction. L'apparition d'une nouvelle ressource est un simple ajout dans le multi-ensemble alors que la disparition d'une ressource est le retrait d'un élément du multi-ensemble, en dé-associant les éventuelles molécules activités liées à cette ressource.

Si la modélisation apparaît prometteuse, sa mise en œuvre repose principalement sur la capacité à gérer efficacement un multi-ensemble à large échelle. En plus des propriétés de persistance qu'un tel multi-ensemble doit avoir, il apparaît important d'avoir une stratégie d'appariement efficace.

Une première piste consisterait à continuer avec l'analogie chimique. Ainsi, le support d'exécution peut être un ensemble de machines, chacune exécutant localement ses réactions. Afin d'assurer la diffusion de l'information, un mouvement brownien peut être simulé en échangeant aléatoirement des molécules entre ces machines. Une autre piste serait de disposer d'algorithme de recherche à large échelle, non nécessairement exhaustif, pour rechercher des molécules adéquates. En tout état de cause, nous pouvons remarquer que le problème est plus simple à traiter car le nombre d'opération à supporter à l'exécution a très fortement diminué.

4.5 Bilan et perspectives

Nos travaux concernant le déploiement d'application sur les grilles de calcul a comporté trois volets. Premièrement, nous avons proposé une architecture générale pour le déploiement automatique d'applications. Ainsi, les différents intervenants et leur rôle ont pu être clairement identifiés. En particulier, nous avons insisté sur le découplage entre la description des applications et la description des contraintes à appliquer lors d'une exécution. Cette architecture, actuellement restreinte à des applications statiques, a été validée via un prototype et le support de plusieurs modèles de programmation.

Deuxièmement, nous avons proposé un modèle réaliste de description de la topologie des réseaux des grilles informatiques. La disponibilité d'un tel modèle est essentielle pour envisager de mettre au point des algorithmes de placement pour les grilles informatiquess. En effet, les applications que nous cherchons à déployer ont de fortes contraintes par rapport au réseau.

Troisièmement, nous avons exploré la pertinence du modèle gamma pour l'exécution de workflow. Les résultats obtenus sont très encourageants. Il est possible d'avoir un modèle très bien défini ne nécessitant pas d'état centralisé pour s'exécuter en milieu distribué. Ces travaux ont été rangés dans le chapitre du déploiement pour deux raisons. D'une part, l'exécution d'un plan de déploiement a toujours paru pouvoir s'exprimer facilement en terme de workflow. Il apparaît que gamma semble un bien meilleur candidat, notamment pour gérer toutes les évènements qui peuvent survenir comme les pannes ou les erreurs. D'autre part, afin de supporter des applications dynamiques, nous sommes à la recherche d'un modèle d'exécution non centralisé.

Ces travaux permettent d'envisager de nombreuses perspectives. Tout d'abord, l'architecture générale de déploiement automatique doit être éten-

due afin de supporter les applications dynamiques. Ce travail, actuellement en cours de réalisation par Boris Daix, demandera de s'intéresser à la gestion de l'état d'une application déployée. Une question également ouverte concerne la répartition des rôles entre le système et un logiciel de déploiement. Quelle connaissance doit avoir un système d'exploitation comme les grilles informatiques des modèles de programmation des applications ? Idéalement, il ne devrait en être découplé. Une piste que nous pensons explorer est l'utilisation du modèle générique comme frontière entre les systèmes d'exploitation des grilles et les outils de déploiement.

Une autre perspective fort importante concerne le développement d'algorithmes de planification efficace. Ces algorithmes devront en plus réaliser de l'ordonnancement lorsque nous disposerons d'un modèle pour les applications dynamiques. En plus de la réalisation proprement dite de ces algorithmes, l'interface entre ces algorithmes et les systèmes d'information des grille n'est pas encore stabilisée. Si nous supposons des grilles de grande taille, il apparaît trop coûteux de chercher à déterminer l'état global de la grille. Une alternative prometteuse est apportée par les algorithmes de recherche développés par la communauté pair à pair, et en particulier ceux supportant la recherche sur des intervalles. Ces algorithmes pourraient également se relever pertinent dans le support d'exécution d'application gamma afin d'accélérer l'appariement entre molécules.

Chapitre 5

Conclusion et perspectives

Sommaire

5.1 Modèle de programmation 85
5.2 Support réseau 86
5.3 Déploiement 86
5.4 Perspectives 87

5.1 Modèle de programmation

La programmation se révèle être un art d'autant plus difficile que la complexité des applications est sans cesse croissante. Il apparaît donc nécessaire d'accompagner cette évolution avec une évolution des modèles de programmation. Par rapport à la programmation des grilles informatiques, les modèles de composants offrent de nombreux avantages. Ils représentent un état de l'art des modèles de programmation ; ils offrent un niveau d'abstraction proche du niveau fonctionnel, ce qui semble bien convenir aux utilisateurs comme l'a montré le projet ACI GRID HydroGRID. Mais surtout ces modèles permettent d'exprimer les relations entre codes sans faire intervenir des concepts de ressources.

Nos travaux sur les modèles de composants logiciels s'inscrivent dans la démarche d'abstraire toujours plus la programmation. Ainsi, le composant manipulé lors de la phase d'assemblage est abstrait. Une implémentation séquentielle peut être choisie lors du déploiement tout aussi bien qu'une implémentation parallèle. Notre proposition de modèle de composants parallèles distribuées garantie que les communications entre deux composants parallèles seront bien parallèles. Le modèle a été validé via des

implémentations ainsi qu'au travers de projets nationaux. Le transfert technologique réalisé avec EDF R&D autour de ce modèle montre l'intérêt de l'approche.

L'étape suivante consistait à lancer l'étude de l'adéquation des modèles de composants pour décrire l'architecture spatiale et temporelle des applications. Les résultats préliminaires laissent présager de bonnes propriétés bien qu'il ne soit pas encore totalement établi s'il faut explicitement différencier ces deux dimensions ou s'il faut s'appuyer sur l'état des composants pour gérer l'aspect temporel.

Enfin, nous avons commencé par explorer une troisième dimension de la programmation, c'est à dire la programmation de propriétés transverses à la structure de l'application. Des travaux préliminaires sur les modèles de composants et de programmation orientée aspects menés avec l'École des Mines de Nantes ont permis de montrer qu'il était possible d'avoir des aspects qui dynamiquement modifient les ports d'un composant. Ainsi, le lien entre l'implémentation d'un composant et l'extérieur d'un composant a pu être établi.

5.2 Support réseau

Le modèle de programmation proposé impose deux contraintes majeures sur les réseaux. D'une part, il s'agit de virtualiser les réseaux constituant les grilles mais sans perte de performance. D'autre part, il s'agit de supporter plusieurs intergiciels communicant simultanément au sein d'un même processus. Le modèle proposé repose sur la reconnaissance des propriétés non conciliable entre le paradigme réparti et le paradigme parallèle, aussi bien au niveau des interfaces d'accès au réseau que des intergiciels. La validation du modèle a comblé nos espérances car nous avons pu montré que des intergiciels comme CORBA pouvaient être compétitifs avec des intergiciels MPI sur des réseaux haute performance. Par conséquence, nous avons pu démontré qu'une application décrite avec notre modèle de composants parallèles distribuées pouvait s'exécuter sur différents types d'architectures (grappe, grappe de PC, grappes interconnecté par un réseau longue distance, etc) avec un surcoût logiciel négligeable.

5.3 Déploiement

Le troisième axe de mes recherches concernait le déploiement d'applications sur des grilles informatiques. D'une part, le déploiement manuel sur

une grille informatique est extrêmement difficile, et d'autre part, le déploiement constitue une étape cruciale car elle contrôle l'affectation des entités applicatives à des ressources. Nous avons ainsi montré la complémentarité entre les environnements de grilles et les modèles de déploiement des modèles de programmation. Afin de supporter la grande diversité des applications, notre modèle de déploiement propose de réaliser le placement indépendant du type de l'application. Ce découplage permet ainsi d'avoir des algorithmes de placement générique.

5.4 Perspectives

Les travaux réalisés ces dernières années ont permis d'atteindre un premier palier offrant un modèle complet pour la définition, l'implémentation, le déploiement et l'exécution d'applications de couplage de codes sur des grilles informatiques sous des contraintes de haute performance mais sans notion de dynamicité. Nous avions fait l'hypothèse d'application statique ainsi que de la non volatilité des ressources une fois celles-ci sélectionnées.

Le palier suivant est constitué de deux types d'objectifs. Premièrement, il s'agit de consolider les résultats obtenus. Deuxièmement, il s'agit d'étendre les modèles et les implémentations en relâchant des hypothèses.

Tout d'abord, si des démonstrateurs ont été construits afin de valider nos propositions, il reste à réaliser des campagnes de tests avec plus d'applications afin d'évaluer les limites du modèles. Ainsi, il est attendu que la gestion de la redistribution des données a probablement quelques limites dans le cas où plusieurs redistributions sont possibles ou des contraintes comme la quantité de mémoire sont à prendre à compte. Il s'agirait de mettre en place un mécanisme pour optimiser globalement les communications en fonction des différents paramètres comme par exemple les performances des réseaux locaux du coté émetteurs et récepteurs, du réseau longue distance, des quantités de mémoire. Au lieu de se contenter de redistribution soit coté émetteur ou récepteur soit entre les deux, il est fort probable qu'une communication efficace doit pouvoir utiliser simultanément tous les réseaux. Une difficulté est de concevoir la fonction d'ordonnancement.

De même, la liaison entre PadicoTM et ADAGE n'a pas encore été réalisée. Actuellement, ADAGE n'a pas pleinement connaissance des intergiciels constituant une application et ne peut donc pas générer des fichiers de configuration pour PadicoTM. Cette étape demande à revoir les interfaces par rapport aux algorithmes de placement ainsi que le plan de déploiement.

Concernant les objectifs à plus long terme, ils concernent des problématiques de modélisation, de conception et de déploiement. Concernant la modélisation, plusieurs pistes me paraissent intéressantes. Premièrement, les modèles de composants sont principalement basés sur une description spatiale d'architectures. Peu de travaux ont été menés pour en plus prendre en compte la dimension temporelle. La plupart du temps, le contrôle d'une application est caché au sein d'un composant "pilote". La fin de la thèse de Hinde Bouziane consiste à proposer un modèle de composant intégrant un modèle de workflow. C'est une première étape. Troisièmement, une question ouverte concerne le niveau du support du parallélisme dans les modèles de composants. Peut on décrire une application parallèle en terme de composant ou non ? Doit on se contenter d'encapsuler un code parallèle dans un composant, gérer le parallélisme au niveau des appels entre composant, ou peut on – par exemple – trouver un modèle qui permette de déterminer le nombre (et la forme ?) des composants en fonction des ressources disponible ? Bref, est-ce que cela a un sens de parler d'algorithme parallèle à niveau d'abstraction proposé par les composants logiciels ? Une première voie de réflexion consiste à étudier la pertinence et les possibilités de supporter des communications collectives entre composants.

En ce qui concerne la conception, deux points me paraissent prometteurs. Tout d'abord, les possibilités d'expression des types de ports des composant reste encore à étudier. Par exemple. l'appel de méthode est parfois trop compliqué quand c'est des communications de type passages de messages que l'algorithme expose. En plus de proposer un niveau d'abstraction pertinent, il s'agit de permettre des mises en œuvre efficace. Ainsi, il est connu que le modèle d'appel de méthodes actuel impose une recopie à l'arriver pour de nombreux algorithmes car le programmeur ne peut spécifier où les données doivent être reçues. Un deuxième objectif serait de dissocier les modèles de composants des protocoles de communications implicitement imposés par le modèle de composant. En effet, il paraît peu probable qu'un seul modèle de composants finisse par s'imposer, ne serait-ce que pour des raisons de concurrence économique [71]. Les modèles de composants ne s'intéressant qu'à la composition, l'objectif est de composer des composants écrits avec différents modèles. Cet objectif est d'autant plus sensé que les applications regroupent actuellement des parties de codes écrites avec des langages différents.

Par rapport à la problématique du placement, le point le plus crucial actuellement est d'avoir des modèles pour la gestion dynamique des applications. C´est un vaste sujet car on trouve des questions relatives à l'interface par rapport à la description des ressources (statiques et dynamique), à la

projection d'une application sur des ressources avec le respect de propriétés de qualité de services des applications, etc. Ces questions sont d'autant plus sensible que le support d'applications multi-modèles de programmation me paraît une propriété incontournable à moyen terme. Ainsi, il reste à déterminer par exemple si les services du système d'exploitation doit avoir connaissance du type de l'application ou si comme pour ADAGE, il est possible de définir un modèle générique. Une autre question concerne l'utilisation de technologies pair-à-pair pour gérer les ressources et en particulier pour effectuer des recherches de ressources adaptés au déploiement en cours. Ces travaux ont débuté en collaboration avec des équipes expertes de l'ordonnancement, des systèmes d'exploitation et des systèmes pair-à-pair. Deux autres travaux à réaliser en collaboration me paraissent très prometteurs et importants. Le premier consiste à intégrer des modèles d'adaptabilité afin de gérer la dynamicité de l'application et la volatilité des ressources. Le second travail à mener consiste à établir un lien plus formel entre un modèle d'aspect supportant des aspects aussi bien entre composants, à l'intérieur d'un composant, qu'au niveau de ses ports, et un modèle de composant parallèle et dynamique.

Enfin, à long terme, mon objectif est de continuer à étudier les abstractions nécessaires pour permettre une description simple d'applications de plus en plus complexes tout en permettant une exécution efficace quelques soient les propriétés de l'infrastructure d'exécution.

Bibliographie

[1] Marco Aldinucci, Massimo Coppola, Marco Danelutto, Marco Vanneschi, and Corrado Zoccolo. *Grid Computing: Software environments and Tools*, chapter ASSIST as a Research Framework for High-Performance Grid Programming Environments. Springer Verlag, 2004.

[2] Gabriel Antoniu and Luc Bougé. DSM-PM2: A portable implementation platform for multithreaded DSM consistency protocols. In *Proc. 6th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS '01)*, volume 2026 of *Lect. Notes in Comp. Science*, pages 55–70, San Francisco, April 2001. Held in conjunction with IPDPS 2001. IEEE TCPP, Springer-Verlag. Available as INRIA Research Report RR-4108.

[3] Gabriel Antoniu, Luc Bougé, and Mathieu Jan. Juxmem: An adaptive supportive platform for data sharing on the grid. *Scalable Computing: Practice and Experience*, 6(3):45–55, September 2005.

[4] Gabriel Antoniu, Luc Bougé, and Sébastien Lacour. Making a DSM consistency protocol hierarchy-aware: an efficient synchronization scheme. In *Proc. Workshop on Distributed Shared Memory on Clusters (DSM 2003)*, pages 516–523, Tokyo, May 2003. Held in conjunction with the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID 2003), IEEE TFCC.

[5] G. Arbab and G. A. Papadopoulos. *Advances in Computers*, volume 46, chapter Coordination Models and Languages. Academic Press, 1998.

[6] Olivier Aumage, Luc Bougé, Jean-François Méhaut, and Raymond Namyst. Madeleine II: A portable and efficient communication library for high-performance cluster computing. *Parallel Computing*, 28(4):607–626, April 2002.

[7] Jean-Pierre Banâtre, Pascal Fradet, and Daniel Le Metayer. Gamma and the chemical reaction model: Fifteen years after. In *Multiset Processing*, volume 2235 of *LNCS*, pages 17–44. Springer-Verlag, 2001.

[8] Jean-Pierre Banâtre, Pascal Fradet, and Yann Radenac. Principles of chemical programming. In S. Abdennadher and C. Ringeissen, editors, *Proceedings of the 5th International Workshop on Rule-Based Programming (RULE 2004)*, volume 124 of *ENTCS*, pages 133–147. Elsevier, June 2005.

[9] Amnon Barak. The Mosix Homepage. `http://www.mosix.org/`.

[10] L. Barroca, J. Hall, and P. Hall. *Software Architectures: Advances and Applications*, chapter An Introduction and History of Software Architectures, Components, and Reuse. Springer Verlag, 1999.

[11] Francoise Baude, Denis Caromel, and Matthieu Morel. From distributed objects to hierarchical grid components. In *International Symposium on Distributed Objects and Applications (DOA)*, number 2888 in LNCS, pages 1226–1242, Catania, Italie, November 2003. Springer Verlag.

[12] 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*, nov 2005. ACTS Collection special issue.

[13] Éric Bruneton, Thierry Coupaye, and Jean-Bernard Stéfani. The fractal component model. Draft Specification version 2.0-3, The ObjectWeb Consortium, February 2004.

[14] Franck Cappello, Frederic Desprez, Michel Dayde, Emmanuel Jeannot, Yvon Jegou, Stephane Lanteri, Nouredine Melab, Raymond Namyst, Pascale Primet, Olivier Richard, Eddy Caron, Julien Leduc, and Guillaume Mornet. Grid'5000: A large scale, reconfigurable, controlable and monitorable grid platform. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing (Grid'2005)*, Seattle, Washington, USA, nov 2005.

[15] E. Caron, F. Desprez, F. Lombard, J.M. Nicod, M. Quinson, and F. Suter. A Scalable Approach to Network Enabled Servers. In B. Monien and R. Feldmann, editors, *Proceedings of the 8th International EuroPar Conference*, volume 2400 of *Lecture Notes in Computer Science*, pages 907–910, Paderborn, Germany, August 2002. Springer-Verlag.

[16] H. Casanova and J. Dongarra. NetSolve: A Network-Enabled Server for Solving Computational Science Problems. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(3):212–223, 1997.

[17] CORBA components. OMG Document formal/02-06-65, Object Management Group, June 2002. Version 3.0.

[18] Ethan Cerami. *Web Services Essentials*. O'Reilly & Associates, 1ère edition, February 2002.

[19] Common object request broker architecture: Core specification. OMG Document formal/04-03-12, Object Management Group, March 2004. Version 3.0.3.

[20] Olivier Coulaud, Michael Dussère, and Aurélien Esnard. Toward a computational steering environment based on CORBA. In G.R. Joubert, W.E. Nagel, F.J. Peters, and W.V. Walter, editors, *Parallel Computing: Software Technology, Algorithms, Architectures and Applications*, volume 13 of *Advances in Parallel Computing*, pages 151–158. Elsevier, 2004.

[21] V. Danjean, R. Namyst, and R. Russell. Integrating kernel activations in a multithreaded runtime system on Linux. In *Parallel and Distributed Processing. Proc. 4th Workshop on Runtime Systems for Parallel Programming (RTSPP '00)*, volume 1800 of *Lect. Notes in Comp. Science*, pages 1160–1167, Cancun, Mexico, May 2000. Held in conjunction with IPDPS 2000. IEEE TCPP and ACM, Springer-Verlag.

[22] Raja Das, Joel Saltz, and Alan Sussman. Applying the chaos/parti library to irregular problems in computational chemistry and computational aerodynamics. In *In Proceedings of the 1993 Scalable Parallel Libraries Conference*, pages 45–56. IEEE Computer Society Press, oct 1993.

[23] DCE 1.1: Remote procedure call. Document C706, The Open Group, August 1997.

[24] Alexandre Denis. *Contribution à la conception d'une plate-forme haute performance d'intégration d'exécutifs communicants pour la programmation des grilles de calcul*. Thèse de doctorat, Université de Rennes 1, IRISA, Rennes, France, December 2003.

[25] Aurélien Esnard. *Analyse, conception et réalisation d'un environnement pour le pilotage et la visualisation en ligne de simulations numériques parallèles*. PhD thesis, Université de Bordeaux, dec 2005.

[26] High Performance Fortran Forum. *High Performance Fortran Language Specification*. Rice University, Houston, Texas, October 1996. Version 2.0.

[27] I. Foster, C. Kesselman, and S. Tuecke. The Nexus approach to integrating multithreading and communication. *Journal on Parallel and Distributed Computing*, 37(1):70–82, 1996.

[28] Ian Foster, Carl Kesselman, Jeffrey M. Nick, and Steven Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration. Draft, Global Grid Forum (GGF), June 2002.

[29] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *Supercomputer Applications*, 15(3):200–222, March 2001.

[30] 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*. MIT Press, November 1994.

[31] C. Germain, V. Néri, G. Fedak, and F. Cappello. XtremWeb: building an experimental platform for Global Computing. In *Grid'2000*, December 2000.

[32] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambridge, MA, USA, 2ème edition, November 1999.

[33] William Grosso. *Java RMI*. O'Reilly & Associates, 1ère edition, October 2001.

[34] Thomas J. Hacker, Brian D. Athey, and Brian Noble. The end-to-end performance effects of parallel TCP sockets on a lossy wide-area network. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium table of contents*, page 314, Washington, DC, USA, 2002. IEEE Computer Society.

[35] Ernst A. Heinz. Sequential and parallel exception handling in modula-3*. In P. Schulthess, editor, *Advances in Modular Languages: Proceedings of the Joint Modular Languages Conference*, pages 31–49, Ulm, Germany, September 1994.

[36] Valérie Issarny. An exception handling model for parallel programming and its verification. In *Proc of the ACM SIGSOFT'91 Conference on Software for Critical Systems*, pages 92–100, New Orleans, Louisianna, USA, December 1991.

[37] E. Jeannot and F. Wagner. Two fast efficient message scheduling algorithms for data redistribution through a backbone. In *18th International Parallel and Distributed Processing Symposium*, page 3, 2004.

[38] Emmanuel Jeannot, Bjorn Knutsson, and Mats Bjorkmann. Adaptive online data compression. In *IEEE High Performance Distributed Computing (HPDC'11)*, Edinburgh, Scotland, July 2002.

[39] Yvon Jégou. Implementation of page management in Mome, a user-level DSM. In *Proc. Intl. Workshop on Distributed Shared Memory on Clusters (DSM 2003)*, pages 479–486, Tokyo, Japan, May 2003. Held in conjunction with CCGrid 2003. IEEE TFCC.

[40] Nicholas T. Karonis, Brian Toonen, and Ian Foster. MPICH-G2: a grid-enabled implementation of the message passing interface. *Journal of Parallel and Distributed Computing (JPDC)*, 63(5):551–563, 2003.

[41] K. Keahey and D. Gannon. PARDIS: A Parallel Approach to CORBA. In *Supercomputing'97*, San Jose, CA, November 1997. ACM/IEEE.

[42] P. Keleher, D. Dwarkadas, A. Cox, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on standard workstations and operating systems. In *Proc. 1994 Winter Usenix Conference*, pages 115–131, January 1994.

[43] Rainer Keller, Bettina Krammer, Matthias S. Mueller, Michael M. Resch, and Edgar Gabriel. MPI development tools and applications

for the grid. In *Workshop on Grid Applications and Programming Tools*, Seattle, WA, USA, June 2003. présenté à GGF8.

[44] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier, and J. Irwin. *Aspect Oriented Programming*. Springer Verlag, June 1997.

[45] Thilo Kielmann, Rutger F. H. Hofman, Henri E. Bal, Aske Plaat, and Raoul Bhoedjang. MagPIe: MPI's collective communication operations for clustered wide area systems. In *1999 ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP'99)*, pages 131–140, Atlanta, GA, USA, May 1999.

[46] 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.

[47] E. Korpela, D. Werthimer, D. Anderson, J. Cobb, and M. Leboisky. SETI@home-massively distributed computing for SETI. In *IEEE Computer Society*, pages 78–83, Los Alamitos, CA, USA, February 2001.

[48] Sriram Krishnan and Dennis Gannon. XCAT3: A framework for CCA components as OGSA services. In *9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS)*, pages 90–97, Santa Fe, NM, USA, April 2004.

[49] Sébastien Lacour. *Contribution à l'automatisation du déploiement d'applications sur des grilles de calcul*. Thèse de doctorat, Université de Rennes 1, IRISA, Rennes, France, December 2005.

[50] Michael J. Lewis, Adam J. Ferrari, Marty A. Humphrey, John F. Karpovich, Mark M. Morgan, Anand Natrajan, Anh Nguyen-Tuong, Glenn S. Wasson, and Andrew S. Grimshaw. Support for extensibility and site autonomy in the legion grid system object model. *Journal of Parallel and Distributed Computing*, 63(5):525–538, 2003.

[51] Gabriel Lopez. Integrating aspects and components: A practical study of adding a port to a component. Rapport de stage, mai 2005.

[52] Renaud Lottiaux, Benoit Boissinot, Pascal Gallard, Geoffroy Vallée, and Christine Morin. Openmosix, openssi and kerrighed: A comparative study. In *Proceeding of IEEE International Symposium on Cluster Computing and the Grid (CCGrid '05)*, Cardiff, UK, May 2005.

[53] Bruce Lowekamp, Brian Tierney, Les Cottrell, Richard Hughes-Jones, Thilo Kielmann, and Martin Swany. A hierarchy of network performance characteristics for grid applications and services. Proposed recommendation, Network Measurement Working Group (NMWG), Global Grid Forum (GGF), January 2004.

[54] Jeff Magee, Naranker Dulay, and Jeff Kramer. A Constructive Development Environment for Parallel and Distributed Programs. In *Proceedings of the International Workshop on Configurable Distributed Systems*, pages 4–14, Pittsburgh, US, March 1994.

[55] R. Marvie, P. Merle, J.-M. Geib, and M. Vadet. OpenCCM : une plateforme ouverte pour composants CORBA. In *Actes de la seconde Conférence Française sur les Systèmes d'Exploitation (CFSE'2)*, Paris, France, Avril 2001.

[56] M. D. McIlroy. Mass Produced Software Components. In P. Naur and B. Randell, editors, *Software Engineering*, pages 138–155, Brussels, Belgium, 1969. Scientific Affairs Division, NATO.

[57] H.H. Mohamed and D.H.J. Epema. The design and implementation of the koala co-allocating grid scheduler. In *Proc. of the European Grid Conference*, volume 3470, pages 640–650, Amsterdam, feb 2005.

[58] Christine Morin, Pascal Gallard, Renaud Lottiaux, and Geoffroy Vallée. Towards an efficient single system image cluster operating system. *Future Generation Computer Systems*, 20(2), January 2004.

[59] Jishnu Mukerji and Joaquin Miller. Mda guide v1.0.1. OMG Document omg/03-06-01, Object Management Group, June 2003. Overview and guide to OMG's architecture.

[60] Zsolt Németh, Christian Pérez, and Thierry Priol. Workflow enactment based on a chemical metaphor. In *The 3rd IEEE International Conference on Software Engineering and Formal Methods*, September 2005.

[61] Zsolt Németh, Christian Pérez, and Thierry Priol. Distributed workflow coordination: Molecules and reactions. In *Workshop on Nature Inspired Distributed Computing*, Rhodes, Greece, 2006. IEEE Society Press. To appear.

[62] S. Pakin, V. Karamcheti, and A. Chien. Fast Messages: Efficient, portable communication for workstation clusters and MPPs. *IEEE Concurrency*, 5(2):60–73, April 1997.

[63] Christian Pérez, Thierry Priol, and André Ribes. A parallel CORBA component model for numerical code coupling. *The International Journal of High Performance Computing Applications (IJHPCA)*, 17(4):417–429, 2003.

[64] Christian Pérez, Thierry Priol, and André Ribes. PaCO++: A parallel object model for high performance distributed systems. In *Distributed Object and Component-based Software Systems Minitrack in the Software Technology Track of the 37th Hawaii International Conference on System Sciences (HICSS-37)*, Big Island, Hawaii, USA, January 2004. IEEE Computer Society Press.

[65] C. René and T. Priol. MPI code encapsulating using parallel CORBA object. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*, pages 3–10, Redondo Beach, CA, August 1999.

[66] André Ribes. *Contribution à la conception d'un modèle de programmation parallèle et distribué et sa mise en œuvre au sein de plates-formes orientées objet et composant*. Thèse de doctorat, Université de Rennes 1, IRISA, Rennes, France, December 2004.

[67] K. Seymour, H. Nakada, S. Matsuoka, J. Dongarra, C.A. Lee, and H. Casanova. Overview of GridRPC: A Remote Procedure Call API for Grid Computing. In *Grid Computing - GRID 2002*, pages 274 – 278, 2002.

[68] W. Richard Stevens. *Unix Network Programming: The Sockets Networking API*, volume 1. Addison-Wesley Professional, 3ème edition, October 2003.

[69] R. Stewart, Q. Xie, , K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. *RFC 2960 - Stream Control Transmission Protocol*. The Internet Society, oct 2000.

[70] Sun Microsystems. Rpc: Remote procedure call protocol specification: Version 2. RFC 1057, June 1988.

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

[72] Y. Tanaka, H. Nakada, S. Sekiguchi, T. Suzumura, and S. Matsuoka. Ninf-G: A Reference Implementation of RPC-based Programming Middleware for Grid Computing. *J. Grid Compututing*, 1(1):41–51, 2003.

[73] Peri L. Tarr, Harold Ossher, William H. Harrison, and Stanley M. Sutton Jr. N degrees of separation: Multi-dimensional separation of concerns. In *International Conference on Software Engineering*, pages 107–119, 1999.

[74] Arthur van Hoff, Hadi Partovi, and Tom Thai. The open software description format (OSD). Submitted note, W3C, August 1997.

[75] Rob V. van Nieuwpoort, Jason Maassen, Gosia Wrzesinska, Rutger Hofman, Ceriel Jacobs, Thilo Kielmann, and Henri E. Bal. Ibis: a flexible and efficient Java based grid programming environment. *Concurrency and Computation: Practice and Experience*, 17(7-8):1079–1107, June 2005.

[76] Joel Winstead and David Evans. Structured exception semantics for concurrent loops. In *In Fourth Workshop on Parallel/High-Performance Object-Oriented Scientific Computing*, Tampa Bay, October 2001.

[77] Keming Zhang, Kostadin Damevski, Venkatanand Venkatachalapathy, and Steven G. Parker. SCIRun2: A CCA framework for high performance computing. In *9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS)*, pages 72–79, Santa Fe, NM, USA, April 2004.

[78] Berkeley Open Infrastructure for Network Computing. `http://boinc.berkeley.edu/`, 2002.

[79] Asian-pacific grid. `http://www.apgrid.org/`.

[80] AspectJ language site. `http://eclipse.org/aspectj/`.

[81] Egee - enabling grids for e-science. `http://public.eu-egee.org`.

[82] The JXTA (juxtapose) project. `http://www.jxta.org`.

[83] MICO, an OpenSource CORBA implementation. `http://www.mico.org`.

[84] The Salome HomePage. `http://www.salome-platform.org/`.

[85] Scalapack. `http://www.netlib.org/scalapack/scalapack_home.html`.

[86] Teragrid. `http://www.TeraGrid.org/`.