Couplage à hautes performances de codes parallèles et distribués

Hamid-Reza Hamidi

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

N° attribué par la bibliothèque

THÈSE

pour obtenir le grade de

Docteur de l'Institut National Polytechnique de Grenoble (INPG)

Spécialité : "Informatique : Systèmes et Logiciels"

préparée au laboratoire ID (Informatique et Distribution)
dans le cadre de l'école doctorale
"Mathématiques, Sciences et Technologie de l'information, Informatique"

présentée et soutenue publiquement

par

Hamid-Reza HAMIDI

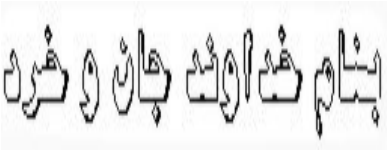
le 5 octobre 2005

Couplage à hautes performances de codes parallèles et distribués

---

JURY

M. Jacques Mossière	Président
M. Frédéric Desprez	Rapporteur
M. Bertrand Braunschweig	Rapporteur
Mme. Brigitte Plateau	Directeur de thèse
M. Thierry Gautier	Responsable de thèse


بنام خداوند جان و خرد

Remerciements

Cette thèse, bien que signée de mon seul nom, ne doit donc pas être attribuée à un travail solitaire : elle reflète ces années de travail mené ensemble ; de jour, de nuit, de week-end, de jours fériés... Je tiens à remercier ici tous ceux qui m'ont aidé, soutenu et encouragé pendant ma thèse.

Mes premiers remerciements vont bien entendu à mon jury. Je tiens tout d'abord à remercier Jacques Mossière pour m'avoir fait l'honneur de présider mon jury. Je remercie également chaleureusement Bertrand Braunschweig et Frédéric Desprez, tous deux rapporteurs, qui ont consacré une partie de leur temps précieux à relire ce manuscrit et à faire des commentaires constructifs. Et évidemment, n'oublions pas mes deux encadrants. Brigitte Plateau, figure de proue de tout le projet APACHE, pour son accueil. Thierry Gautier qui m'a fait confiance pendant ces ans, et qui m'a donné l'impression de savoir où j'allais y compris quand moi-même je ne le savais pas !

Je tiens à remercier tous le projet APACHE / MOAIS, Maxime et Laurent pour la correction du français de ce mémoire. Merci à Jean-Louis et Samir pour m'avoir aider à faire évoluer ma soutenance. Merci à Mauricio, Luiz-Angelo, Jesus-Alberto, Georges, Said, Nhien-An, Jean-Michel, Lucas, Ihab, Jonathan, Adrien, Corine et Gregory pour toutes les discussions mémorables dans le laboratoire. Merci aussi à tous les autres que j'oublie ici et qui ont contribué d'une façon ou d'une autre à cette thèse, comme mes amis iraniens pour les moments inoubliables qu'on a passé ensemble.

Je tiens évidemment à remercier mes parents, mes frères et mes soeurs, pour ce qu'ils sont et parce que rien ne serait si bien sans eux. Merci à Soroush, pour qui, chaque jour, je fais de mon mieux pour être à ses yeux un véritable héros.

Enfin, merci à celle qui a su me donner l'envie, la joie et la soif de m'évoluer. C'était il y a de cela une dizine d'années, à Gazvin, une petite stagiaire en informatique. J'étais bien bien moyen et dissipé ; elle a su me convaincre. Shokooh, pour cela, l'intégralité de mon travail te sera toujours et à jamais dédiée.

Table des matières

1

Introduction Générale

Sommaire

1.1 Couplage de codes

L'accroissement rapide de la puissance des calculateurs actuels et leur interconnexion en grappes et grilles de calcul à l'aide de réseaux rapides, permettent d'envisager, en mode de production, l'utilisation de plusieurs codes de calculs numériques couplés pour la simulation de phénomènes physiques plus complexes.

Pourquoi le couplage de codes ? La première question que l'on devrait se poser est : « a-t-on réellement besoin de coupler des codes de simulations et pourquoi ? » Pour répondre à cette question, il faut analyser les simulations que les chimistes, mécaniciens, physiciens, ..., souhaitent réaliser. Ces simulations sont tridimensionnelles avec une physique la plus complète possible pour bien représenter le phénomène que l'on cherche à modéliser. De plus, comme on souhaite un haut degré de fidélité dans la simulation du phénomène, un nombre d'inconnus très important (plusieurs centaines de millions) est nécessaire, ce qui oblige de paralléliser l'application pour obtenir des temps de simulations raisonnables.

Ainsi les simulations devenant hautement sophistiquées et complexes, une personne, ou une institution, ne peut développer le logiciel de manière isolée. En effet, dans de telles simulations, il faudrait avoir une expertise sur toute la chaîne, c'est-à-dire des domaines de la physique utilisée jusque dans l'implémentation efficace sur des architectures hétérogènes. Par exemple, dans le domaine de l'optimisation du profil d'une aile d'avion [64], il est nécessaire de considérer des simulations multi-physiques qui tiennent compte des interactions entre la structure de l'aile, l'écoulement de fluide autour de l'aile, et l'échauffement dû au frottement. De plus, dans certains cas, des aspects électromagnétiques peuvent être considérés.

Dans le but d'obtenir des résultats toujours plus précis, un nouveau type de simulation numérique, dont l'objectif est de simuler plusieurs physiques en même temps, est apparu. Ce type d'application est appelé *application de couplage de codes*. En effet, plusieurs codes (physiques) sont couplés ou interconnectés afin qu'ils communiquent pour réaliser la simulation.

1.1.1 Exemples d'application de couplage de codes

Pour mieux connaître les caractéristiques attendues de la part du modèle de programmation pour ce type d'application, deux exemples typiques sont tout d'abord décrits. Ils serviront tout au long de cette thèse pour illustrer notre travail.

1.1.1.1 SIMBIO : un exemple de simulation numérique en chimie

L'application SIMBIO [13] couple plusieurs codes parallèles spécialisés dans deux domaines différents. A l'échelle atomique, la simulation est régie par la mécanique quantique. A l'échelle macroscopique, les équations du mouvement de Newton sont utilisées pour simuler le comportement de la structure moléculaire en utilisant une application parallèle de dynamique moléculaire [14]. Le solvant, par exemple un liquide, est modélisé comme des matériaux de continuum chargé. Le champ électrostatique est décrit par les équations de Poisson, avec des particules chargées. L'algorithme est basé sur une formulation intégrale des équations avec une implémentation parallèle. Les deux modèles interagissent réciproquement par une surface qui sépare le domaine des équations de Newton (MD) et

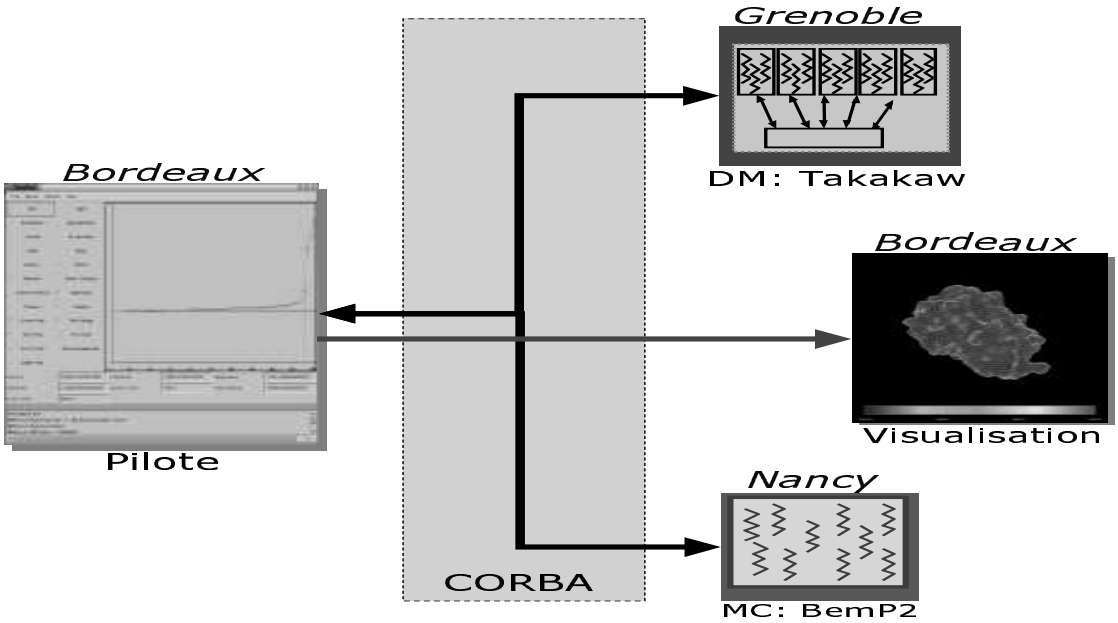


Figure 1.1 *La plate-forme d'application SIMBIO.*

le domaine des équations de Poisson (CM). La description complète de l'algorithme numérique pour le couplage est hors de portée de ce rapport [13]. Pour compléter la description de SIMBIO, notons que d'autres applications sont incluses dans le processus de simulation : l'une d'elles a besoin de stocker toutes les positions et les vitesses des atomes pour une analyse statistique de leur trajectoire ; les autres permettent de visualiser la structure moléculaire ainsi que la surface de séparation.

La figure 1.1 offre une vue schématique de l'application. L'application *pilote* contrôle la simulation (lancement, arrêt). Pour savoir quand arrêter la simulation, le code contrôleur reçoit régulièrement des informations sur la convergence des différents codes. Enfin, il peut dynamiquement changer les paramètres de la simulation. Par exemple, lors de la visualisation de la simulation, l'opérateur peut changer certains paramètres physiques pour les corriger. Les différents programmes de la simulation sont distribués sur des machines différentes.

1.1.1.2 Sappe : un exemple de réalité virtuelle

La réalité virtuelle (RV) est, par nature, hautement interactive et, par conséquent, les applications de RV ont des besoins matériels et logiciels spécifiques différents de ceux relatifs aux applications à hautes performances traditionnelles. Par exemple, un système de réalité virtuelle doit pouvoir recevoir, traiter et produire des données au moins dix fois par seconde (30 de préférence) pour conserver un niveau de performance correct.

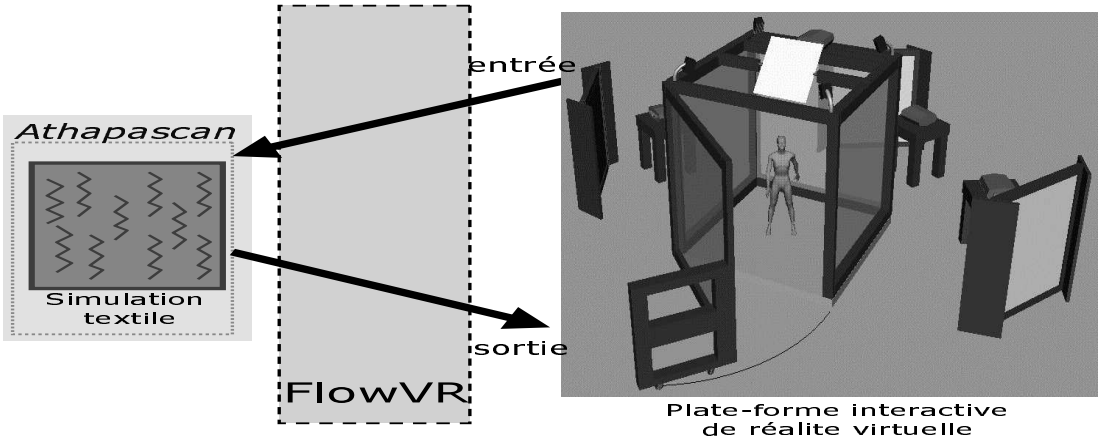


Figure 1.2 *La plate-forme d'application Sappe.*

L'application Sappe [86, 6] est une application de simulation de textiles décomposée en deux modules. Le premier concerne la simulation numérique du textile. Cette simulation est basée sur une discrétisation du tissu en un ensemble de particules. Ces particules sont connectées par des ressorts permettant de modéliser un comportement réaliste du tissu. Cet ensemble forme un maillage triangulaire. En informatique graphique, chacun de ces triangles est appelé facette. L'affichage de ces facettes va permettre la visualisation du textile. Ces calculs ont été parallélisés pour permettre une exécution sur une grappe de PCs. Le second module concerne l'interaction avec l'utilisateur : l'entrée interactive et la visualisation multi-écrans.

La figure 1.2 offre une vue schématique de l'application. Ce couplage entre simulation parallèle et visualisation parallèle permet d'exploiter efficacement chacun des codes parallèles sans dégrader trop fortement les performances de l'exécution globale, afin d'atteindre un rendu en temps réel. Le code de simulation de textile à fait l'objet de la thèse de F. Zara [86], tandis que l'environnement de réalité virtuelle est le travail de doctorat de J. Allard [6].

1.1.2 Modèle de conception

Du fait de la complexité des méthodes numériques, des structures de données de chacun des codes, et afin de tenir compte de l'évolution des besoins futurs, il n'est plus envisageable de développer des applications monolithiques, qui sont très difficiles à maintenir et à faire évoluer. Une approche basée sur la réutilisation des codes et de leurs compositions dans un langage de haut niveau est nécessaire. Dans une telle approche, un code constitue un *composant* [83] réutilisable et déployable qui

implémente une certaine fonctionnalité (pré-processing des données, solveur, adaptation du maillage, post-processing comme par exemple de la visualisation). Chaque composant nécessite des ressources spécifiques (puissance de calcul, capacité mémoire, interface graphique, carte d'entrée-sortie spécialisée, *etc*). Un composant qui demande une forte puissance de calcul peut être associé à un code parallèle.

Suivant cette approche, le processus de développement d'une nouvelle application consiste à définir ou à récupérer des composants élémentaires nécessaires à la simulation numérique envisagée. Pour un nouveau composant, il s'agit de définir son interface d'utilisation lui permettant d'une part, d'encapsuler son code et d'autre part, de pouvoir être réutilisé soit grâce à un langage (ou une bibliothèque) de composition, soit de manière *ad hoc*. L'application est définie par la description des interactions entre les différents composants. Cette description peut être statique (connue avant exécution) ou dynamique (un processus produit, au fur et à mesure du calcul, cette description). Grâce à cette description des interactions, il est possible de déployer et de configurer les composants sur l'architecture choisie : les composants sont placés sur les machines de l'architecture, les connexions entre composants s'établissent et l'exécution peut alors commencer. Dans le cadre d'une description dynamique des interactions entre composants, l'ensemble de ces étapes est faite à l'exécution et se rapproche des problèmes liés à l'ordonnancement en ligne des tâches d'un programme parallèle.

1.1.3 Architecture cible

Les applications de couplage exigent d'importantes ressources informatiques puisqu'elles sont composées de codes numériques qui requièrent chacun une grande puissance de traitement (calcul, stockage) pour que chaque physique soit simulée précisément. Il est alors très difficile d'obtenir la ressource informatique nécessaire sur un seul centre de calcul. En revanche, les *grilles informatiques* [35] peuvent offrir les ressources nécessaires à l'exécution de ce type d'application. En effet, une grille informatique est définie comme la mise en commun de ressources de différents centres de calcul interconnectées par des réseaux à longue distance et à très haut débit. Les différents codes de la simulation sont déployés sur les ressources de plusieurs sites. Grâce aux capacités des réseaux, les applications peuvent communiquer rapidement d'importantes quantités de données.

Cette architecture hétérogène est caractérisée essentiellement par un temps d'accès à une donnée non uniforme. Ce temps est d'autant plus important que les distances sont grandes et que le débit est faible.

1.2 Objectifs et contributions

Cette thèse s'intéresse aux problématiques liées au *couplage à hautes performances de codes parallèles et distribués*. L'obtention des performances repose sur la conception d'applications *distribuées* dont certains composants sont *parallèles* et dont les *communications* sont efficaces. Cette conception se base sur des techniques de composition de composants et de déploiement sur les grilles de calcul. Ce domaine de recherche est à l'intersection des domaines des *systèmes distribués* pour la gestion des ressources, du *calcul distribué* pour la conception d'applications réparties et du *calcul parallèle* pour les critères de performances qu'il s'agit de respecter.

Notre contribution se situe dans deux modèles de programmation : le modèle « *appel de procédure à distance (RPC)* » et le modèle « *mémoire partagée* ». L'objectif est de contribuer à la conception d'un modèle de programmation pour les applications de couplage de codes parallèles et distribués. Les contributions apportées par ce travail de recherche sont les suivantes.

Modèle RPC. Nous avons montré comment exploiter automatiquement le parallélisme entre plusieurs invocations de méthodes du modèle RPC de CORBA. Un modèle de coût, pour une architecture homogène illustre les gains de notre approche vis-à-vis d'une utilisation d'une implantation standard de CORBA. Notre implantation se base et hérite des propriétés de moteur d'exécution KAAPI utilisé dans le langage ATHAPASCAN.

Modèle mémoire partagée. Nous avons donné une nouvelle définition des objets partagés du langage ATHAPASCAN afin de prendre en compte une distribution spatiale et temporelle d'un ensemble de données. Cette approche permet d'étendre le langage à ATHAPASCAN pour la coordination d'un ensemble de codes distribués et communicants.

1.3 Organisation du manuscrit

Ce document s'organise en cinq parties principales.

I. Contexte d'études. Cette partie présente le contexte de notre travail. Tout d'abord, dans le chapitre 2, nous présentons le domaine du calcul parallèle pour l'obtention des performances, à savoir les modèles de programmation parallèle et les outils ou les bibliothèques permettant le développement d'applications parallèles. Ensuite, nous présenterons les modèles et les outils pour le couplage de codes (chapitre 3). Une analyse des solutions existantes permet de mettre en évidence des lacunes importantes de ces propositions pour l'obtention de hautes performances des applications de couplage de codes. Le chapitre 4 présente la problématique du couplage de codes à hautes performances abordée dans cette thèse.

II. HOMA : Composition à hautes performances de services repartis. Dans cette partie, nous présentons HOMA qui est un compilateur IDL CORBA associé à un support exécutif permettant d'exploiter de manière transparente le parallélisme entre plusieurs invocations de méthodes d'un client vers un ensemble de serveurs. Le chapitre 5 présente les caractéristiques à hautes performances

de HOMA dans un modèle théorique de grappe homogène. L'implantation de HOMA au dessus de CORBA et du moteur exécutif KAAPI d'ATHAPASCAN fait l'objet du chapitre 6.

III. Modèle pour le couplage de codes. Dans cette partie, nous promouvons un modèle pour le couplage de codes basé sur un modèle de données partagées. Nous basons notre travail sur le modèle ATHAPASCAN, présenté dans le chapitre 3, et nous l'étendons afin de prendre en compte deux caractéristiques fondamentales des applications de couplage de codes à hautes performances : d'une part, la gestion des données distribuées provenant, de fait, des codes parallèles est présentée dans le chapitre 8 ; et d'autre part la gestion des séquences de données partagées provenant des communications inhérentes entre différents codes de ces applications (chapitre 7).

IV. Applications. L'ensemble des solutions présentées dans cette thèse est illustré à travers des mesures de temps sur différentes applications. Le chapitre 9 présente les résultats expérimentaux obtenus avec HOMA dans le cadre de l'application SIMBIO en simulation moléculaire ; ansi que les mesures expérimentales dans le cadre d'un couplage en simulation numérique dans le cadre d'une application de simulation de tissus interactives.

V. Conclusions. Cette partie présente nos conclusions sur le travail effectué ainsi que les perspectives de recherche possibles à cette thèse.

1.4 Publications

Cette thèse a conduit à des publications, en particulier autour du projet HOMA. Les recherches concernant l'extension d'ATHAPASCAN comme modèle pour le couplage de codes sont présentées pour la toute première fois dans ce manuscrit et font l'objet d'une rédaction d'article. Ces publications sont les suivantes.

Chapitres de livres

[1] T. GAUTIER, O. COULAUD, H.R. HAMIDI, chapitre « Couplage de codes parallèles », dans **Informatique répartie**. Editeurs M. Jemni, Y. Slimani et D. Trystram. ISBN 2746208571, Hermès Science Publications. Mai 2005.

Revue internationale

[2] T. GAUTIER, H.R. HAMIDI, « Re-scheduling invocations of services on RPC-based Grid », In International Journal Computer Languages, Systems and Structures (CLSS), Special Issue on SEMANTICS AND COST MODELS FOR HIGH-LEVEL PARALLEL PROGRAMMING, Elsevier press, à paraître en 2005.

Conférences avec comité de lecture

[3] T. GAUTIER, H.R. HAMIDI, « Automatic Re-scheduling of Dependencies in a RPC-based Grid », In Proceedings of the 18th annual ACM International Conference on Supercomputing (ICS'04), pages 89-94, Saint-Malo, France, ACM Press, June 2004.

[4] T. GAUTIER, H.R. HAMIDI, « High Performance Composition of Services with Data Dependencies on a Computational Grid », In Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'04), Volume II, pages 801-809, Las Vegas, USA, June 2004.

[5] T. GAUTIER, H.R. HAMIDI, « HOMA : un compilateur IDL optimisant les communications des données pour la composition d'invocations de méthodes CORBA », RENPAR'15, La Colle sur Loup, France, Octobre 2003.

Rapport de recherche

[6] T. GAUTIER, H.R. HAMIDI, « Homa : automatic re-scheduling of multiple invocations in CORBA », INRIA Rhône-Alpes, projet APACHE, Research report RR-5191, May 2004.

I

Contexte d’étude

2

Calcul parallèle et reparti

Sommaire

Le chapitre précédent a présenté les applications de couplage de codes. Ce type d'application requièrt beaucoup de ressources informatiques pour leur exécution. Les logiciels existants pour la programmation de ces applications présentent des fonctions de trop bas niveau pour programmer ces simulations qui sont composées de codes parallèles communicants dans un environnement distribué. Ce chapitre résume tout d'abord les architectures des machines cibles, puis les différentes abstractions que fournissent les modèles de programmation classiques pour les applications parallèles et pour les applications distribuées. Ce chapitre s'achève par une synthèse de ces logiciels. Le chapitre suivant présentera les modèles et les outils plus particulièrement dédiés au couplage de codes.

2.1 Architectures de machines

Le chapitre précédent a présenté les applications de couplage de code. Ce type d'application requièrt beaucoup de ressources informatiques pour leur exécution. Une possibilité consiste à utiliser les grappes ou grilles informatiques.

Des architectures spécifiques de machines et de réseaux ont été développées pour les calculs parallèles. Cependant il n'existe pas, à proprement parler, d'*architectures réparties*, car les systèmes répartis sont intrinsèquement un assemblage de diverses ressources hétérogènes. La caractéristique principale des systèmes répartis est l'hétérogénéité : des architectures, des systèmes d'exploitation, des logiciels, des réseaux, ou encore des politiques d'administration.

- **Super calculateur.** Habituellement, une machine parallèle (appelée communément *supercalculateur*) est une plate-forme homogène, dont tous les nœuds présentent la même architecture, à l'intérieur d'un seul domaine d'administration. L'ensemble des nœuds est ainsi vu comme une seule entité logique où tous les nœuds sont équivalents. Les supercalculateurs sont des machines caractérisées par un grand nombre de processeurs reliés par un réseau à haute performance - machines dites « massivement parallèles » -, ou des processeurs exploitant le parallélisme du calcul sur des vecteurs de nombre - on les qualifie alors de « vectorielles » - un système d'exploitation spécifique, et un prix exorbitant !
- **Grappes.** L'émergence des grappes de calcul est liée à la puissance sans cesse croissante des PCs, comblant le retard qui les séparait des puissances développées par les supercalculateurs. Les matériels réseaux à très haute performance tels que Myrinet [19] et SCI[1] [66] offrent une puissance du même ordre de grandeur que celle disponible directement sur le bus système des PCs (plusieurs Moctet/s). De simples PCs interconnectés par ces réseaux, même s'ils ne sont pas capables de rivaliser en performances pures avec les processeurs des supercalculateurs, les battent en ce qui concerne le rapport performance/prix et la puissance cumulée d'un grand nombre de PCs (plusieurs centaines à plusieurs milliers).

[1] *Scalable Coherent Interface*

- **Architectures réparties.** Un système réparti est constitué de machines diverses, stations de travail reliées par des réseaux LAN[2] et/ou WAN[3], comme par exemple des machines éparpillées sur internet. Nous distinguons les variantes suivantes :
  - **Système réparti local** ou *intranet* utilise un LAN et reste à l'intérieur d'un domaine d'administration. Typiquement, il regroupe des stations de travail et/ou des serveurs sur un unique site, à l'abri des préoccupations de sécurité.
  - **Système réparti à large échelle** utilise des WANs et des LANs et se trouve sur plusieurs domaines d'administration. Typiquement, il regroupe des machines de plusieurs institutions et/ou entreprises qui collaborent ou mettent en commun leurs infrastructures.
  - **Internet** est un système réparti utilisant un grand nombre de machines ayant un rôle équivalent - à la fois client et serveur. Une telle architecture est qualifié de système *pair-à-pair*. Les utilisations actuelles sont le partage de données et la mise à disposition du temps de calcul inutilisé des machines des particuliers.

La plupart des définitions concernant les architectures parallèles et réparties ne concernent que leur construction. Le terme **grille** informatique [38] est apparu à la fin des années 90. Nous citons ici une définition du point de vue des utilisateurs d'une grille :

Définition 1 *[78] Une **grille informatique** est un système distribué composé du partage de ressources informatiques appartenant à plusieurs organisations. Elle offre aux utilisateurs une vue cohérente de ses ressources.*

De point de vue de la puissance de calcul, une grille peut être assimilée à une architecture parallèle distribuée. En effet, certains nœuds la constituant sont des nœuds parallèles : supercalculateurs et grappes de PC. Ils sont reliés par des réseaux tels que des WAN à très haut débit. La nature des grilles est très hétérogène puisque les matériels et les systèmes d'exploitation qui les constituent peuvent être très disparates. En France, il existe un projet fédérateur d'une grille pour la recherche qui est appelé Grid5000 de l'ACI GRID.

La section suivante présente les modèles de programmation des applications parallèles et la section 2.3 présente les modèles de programmation pour les architectures distribuées.

2.2 Modèles de programmation parallèle

Dans cette section, nous décrivons les modèles de programmation utilisés pour programmer les applications parallèles. Cette façon de programmer est souvent intimement liée à l'architecture sous-jacente. La plupart du temps de tels systèmes parallèles, basés sur des flots d'instructions différents, exécutent en réalité un même programme. Ce modèle d'exécution, cas particulier de MIMD[4], a été baptisé SPMD[5] par Frederica Darema [28]. Il existe deux modèles de programmation courants en parallélisme, souvent de type SPMD : à mémoire distribuée ou à mémoire partagée.

[2] *Local Area Network*
[3] *Wide Area Network*
[4] *Multiple Instruction Multiple Data*
[5] *Single Instruction Multiple Data*

2.2.1 Parallélisme à mémoire partagée

Dans ce modèle de parallélisme à mémoire partagée [74], la machine parallèle est vue comme un ensemble de processeurs qui accèdent à la même mémoire virtuellement partagée (MVP). La mémoire peut être physiquement partagée par plusieurs processeurs (SMP[6]), ou une mémoire physiquement distribuée avec l'illusion d'une mémoire partagée assurée par des mécanismes matériels (UMA[7], NUMA[8], CC-NUMA[9]) ou logiciels (DSM[10] [74]).

Avec ce modèle, l'entité de base est un fil d'exécution ou « *thread* .» Les échanges de données entre les différents fils se font simplement par des lectures/écritures en mémoire. Ceci amène le principal aspect de la programmation parallèle à mémoire partagée : la « cohérence ». Il s'agit de spécifier dans quelle mesure les différents fils d'exécution voient les modifications apportées par les autres sur une même zone de mémoire partagée. En effet, il est rare que tous les threads voient réellement la même mémoire ; dans le cas de DSM ou de NUMA, la mémoire partagée est ajoutée au-dessus d'une mémoire physiquement distribuée. Même avec un SMP, les processeurs ont des caches qui peuvent mener à l'existence, à un moment donné, de plusieurs versions d'une donnée. Les garanties sur la propagation des actions sur la mémoire sont spécifiées par un *modèle de cohérence* [29].

La façon de gérer la propagation des modifications de façon à respecter un modèle de cohérence s'appelle un *protocole de cohérence*. La parallélisation des applications n'étant pas toujours chose aisée, des langages et compilateurs spécialisés ont vu le jour pour faciliter la programmation parallèle à mémoire partagée. Le parallélisme est géré directement dans le langage ; le compilateur génère un code parallèle en suivant les indications données par le programmeur.

2.2.2 Parallélisme à mémoire distribuée

Avec le modèle de parallélisme à mémoire distribuée, la machine parallèle est vue comme un ensemble de nœuds équivalents les uns aux autres ; chaque nœud est constitué d'un ou plusieurs processeurs et de sa mémoire associée. Les mémoires des différents nœuds sont physiquement distribuées ; pour accéder aux données d'un autre nœud, il est nécessaire de réaliser une communication. Les deux paradigmes de communication principaux sont l'*échange de message* et l'appel de *procédure à distance* ou RPC[11].

En « échange de message », les communications peuvent avoir lieu en *point à point* - un émetteur, un récepteur - ou peuvent être *collectives*, c'est-à-dire impliquer plusieurs nœuds. Lors d'un échange de message en point à point, l'émetteur envoie les données, le récepteur les reçoit explicitement. Les opérations collectives sont de trois types : synchronisation, communications, et calculs. La synchronisation est essentiellement une barrière, où tous les nœuds attendent tous les autres. Les communications collectives sont : la diffusion (*broadcast*) - la même donnée est envoyée par un nœud à tous les autres - ; dispersion/regroupement (*scatter/gather*) - dispersion et regroupement de blocs d'un vecteur entre un nœud et tous les autres - ; ou d'autres schémas plus évolués tels que des opérations « tous vers tous » (*all-to-all*). Les calculs collectifs sont typiquement la réduction (*reduce*) qui

[6] *Symetric Multi-Processors*
[7] *Uniform Memory Access*
[8] *Non Uniform Memory Access*
[9] *Cache Coherent - Non Uniform Memory Access*
[10] *Distributed Shared Memory* ou, en français, MVP
[11] *Remote Procedure Call*

permet par exemple de récupérer sur un nœud la somme d'un vecteur distribué sur tous les nœuds. L'exemple essentiel d'un tel modèle est MPI [31].

En RPC, l'opération de base est l'appel d'une procédure à distance, c'est-à-dire qu'un nœud peut appeler une procédure sur un autre nœud. L'appel peut contenir des paramètres qui seront communiqués à la procédure qui peut également fournir des valeurs de retour communiquées à l'appelant. L'intérêt du modèle de programmation par RPC est qu'en plus de transférer des données, il transfère également le flot de contrôle du nœud appelant au nœud appelé.

Il existe un modèle à mi-chemin entre l'échange de message et le RPC qui est appelé « message actif » (*active messages*) [61]. Dans un message actif, l'envoi se fait de la même façon qu'en « échange de message ». La réception quant à elle se fait par un traitant (handler) appelé quand une arrivée de message est détectée ; la réception se fait ensuite à l'intérieur du traitant, en utilisant une méthode similaire à celle employée en échange de message. Les messages actifs sont cependant rarement un modèle de programmation applicatif, mais plutôt un modèle mis en œuvre par des bibliothèques de communication destinées à être utilisées par des exécutifs de plus haut niveau. KAAPI [44] utilise les messages actifs comme primitives de communication entre processus.

2.3 Modèles de programmation distribuée

Dans cette section, nous décrivons les modèles de programmation utilisés pour programmer les systèmes répartis. Dans la programmation de tels systèmes, l'accent est souvent mis sur la robustesse et l'interopérabilité, parfois au détriment de la performance. La programmation répartie est faite essentiellement selon trois modèles : le modèle client-serveur, le modèle à agents et les systèmes pair-à-pair.

2.3.1 Modèle client-serveur

Le modèle de programmation des systèmes répartis le plus fréquent est le client-serveur. C'est un modèle asymétrique où un nœud joue le rôle de serveur et un autre nœud joue le rôle de client. Le serveur attend les requêtes venant des clients. Un client envoie sa requête accompagnée de ses arguments, ce qui déclenche éventuellement une action sur le serveur, puis le client reçoit le résultat de sa requête. Le client-serveur est le mode de fonctionnement typique des RPC, des RMI[12] où le serveur est alors un objet. Il est utilisé dans CORBA[13] [67] ou les Web Services [71].

2.3.2 Modèle basé sur des agents

Dans une fédération, une entité centrale, appelée *agent / fédérateur* régit le déroulement du calcul et toutes les interactions à l'intérieur du système. Des nœuds « clients » appelés *membres*, pouvant rejoindre ou quitter la fédération en se connectant ou déconnectant du fédérateur, apportent leur contribution au calcul. Le fédérateur et les membres de la fédération communiquent en échangeant des messages. Ce modèle est par exemple utilisé dans l'architecture du simulateur HLA [3] du DMSO (Defense Modeling and Simulation Office), dans des programmes de calcul sur *internet* tels

[12] *Remote Method Invocation* - appel de méthode à distance
[13] *Common Object Request Broker Architecture*

que SETI@home [2] ou Folding@home [1], ou dans les environnement de Metacomputing comme Netsolve [25], Ninf [80] et DIET [21].

2.3.3 Système pair-à-pair

Un système pair-à-pair est un système réparti où tous les nœuds, appelés alors « pairs » sont logiquement équivalents. Tous les pairs jouent à la fois le rôle de client et de serveur ; ils peuvent se connecter de façon arbitraire les uns aux autres (d'où le nom « pair-à-pair ») et pas uniquement à un fédérateur. Les exemples de système pair-à-pair sont généralement des services de partage de données. Il existe également des exemples de système pair-à-pair pour le calcul tel que XtremWeb [53]. Les caractéristiques principales de ces systèmes sont, d'une part, la grande volatilité des nœuds de calcul, ils peuvent se déconnecter à tout moment sans prévenir, et, d'autre part, l'hétérogénéité des performances, aussi bien pour les machines que pour les liens réseaux.

2.4 Modélisation d'une exécution concurrente

Différentes approches existent dans la littérature pour décrire les applications concurrentes. Dans cette section, nous nous intéressons à la modélisation d'une exécution d'applications concurrentes par flot de données et de contrôle. Le graphe de flot de données est constitué par des nœuds *tâches*, des nœuds *versions* (qui représentent les données en mémoire partagée) et des arêtes qui représentent les accès des tâches sur les versions.

Fil d'exécution (tâches). Un fil d'exécution est une suite logique d'actions résultant de l'exécution d'un programme. Une application est caractérisée par les tâches et les données partagées qu'elle crée. Une tâche, lors de son exécution, peut créer de nouvelles tâches ou de nouvelles données partagées. La liste des données partagées qui pourront être accédées par une tâche est spécifiée dans les paramètres de la tâche lors de sa création.

Versions de données partagées. Afin de pouvoir tracer le flot de données concernant les accès aux données partagées, chaque donnée est considérée comme une succession de versions. Ce sont ces versions, représentant les différentes valeurs prises par la donnée au cours de l'exécution, qui sont accédées par les tâches. La suite des versions représente donc la suite des valeurs référencées par la donnée partagée ; chaque version est une sorte de variable à assignation unique.

Lorsqu'une tâche prend pour paramètre une référence sur une donnée en mémoire partagée, c'est une référence sur l'une des versions associées à la donnée partagée qui lui sera retournée lors de l'exécution. Cette version, ordonnée par chaînage, identifie la valeur qui sera retournée lors de l'accès puisque le chaînage spécifie la série des écritures qui seront vues par cette lecture.

Lors de l'exécution d'une application, les tâches générées, les données partagées et les droits d'accès sur ces données constituent un graphe de flot de données.

Définition 2 *[42] Le* `graphe de flot de données` *associé à une exécution est le graphe* $G = (V, E)$ *tel que les tâches* (V_t) *et les versions* (V_v) *forment l'ensemble* $V = (V_t \cup V_v)$ *des nœuds, et les accès des tâches sur les versions forment l'ensemble* E *des arêtes. Ce graphe est biparti,* $E \subset (V_t \times V_v) \cup (V_v \times V_t)$.

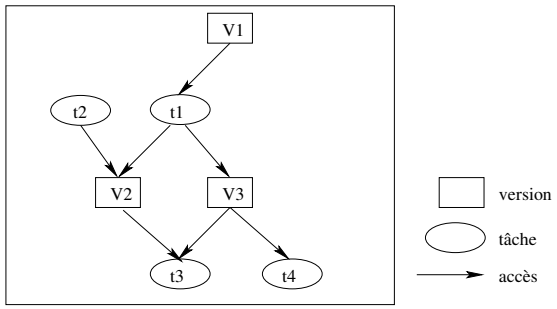


Figure 2.1 ***Graphe de flot de données.*** *Les quatre tâches t_1, t_2, t_3 et t_4 accèdent les trois versions v_1, v_2 et v_3 associées à trois données différentes. La tâche t_2 accède en écriture à v_2 qui est elle-même accédée en lecture par t_3 : il y a donc une contrainte de précédence entre ces deux tâches : t_2 doit être exécutée avant t_3 . Il est à noter que les tâches t_1 et t_2 peuvent être exécutées en parallèle, l'accès concurrent en écriture sera géré par la mémoire virtuelle partagée afin de respecter un modèle de cohérence. De même pour t_3 et t_4.*

La signification d'une arête est la suivante : soient $t \in V_t$ une tâche et $v \in V_v$ une version, l'arête $(t, v) \in E$ traduit un droit d'accès en écriture de la tâche t sur la version v, et l'arête (v, t) un droit d'accès en lecture de la tâche t sur la version v. Un exemple de graphe de flot de données traduisant les accès de 4 tâches sur 3 versions en mémoire partagée est donné dans la figure 2.1.

2.5 Outils et bibliothèques

Dans les deux sections précédentes, nous avons étudié les modèles de programmation parallèle et distribuée. Cette section s'intéresse à résumer certains des outils et des bibliothèques de programmation, basés sur les modèles présentés, qui offrent les fonctionnalités de bas niveaux pour la conception d'applications de couplage de code.

2.5.1 Processus légers : threads POSIX

Un processus léger (ou thread) est une abstraction permettant d'exprimer des multiples flots d'exécution indépendants au sein d'un processus système (e.g. UNIX, appelé processus lourd). Les différents flots d'exécution partagent le contexte de mémoire du processus lourd, leur gestion est donc moins coûteuse que celle des processus système. En effet, la gestion des processus légers nécessite seulement la manipulation de leur contexte d'exécution, tandis que les opérations sur les processus lourds demandent aussi la gestion de zones mémoire et des ressources (fichiers ouverts, verrous, *etc.*) allouées par le système. Par exemple, un changement de contexte pour un processus léger est de 10 à 1000 fois plus rapide que pour un processus lourd.

Les opérations usuellement disponibles pour la programmation avec des processus légers sont la création (sur le même processeur ou sur un autre processeur), la destruction (souvent l'autodestruction), la communication entre thread (par message, appel de procédure à distance, *etc.*) et la synchronisation (par sémaphore, verrou pour l'exclusion mutuelle, barrière de synchronisation, partage de

données, *etc.*). La norme POSIX [63] définit un standard pour les opérations manipulant des processus légers et des implantations sont proposées par la plupart des grands constructeurs d'ordinateurs.

Bien qu'en ce qui concerne l'interface de programmation les bibliothèques de processus légers offrent souvent des fonctionnalités similaires, leurs mises en œuvre diffèrent sensiblement. Sur les systèmes d'exploitation, nous pouvons rencontrer les fonctionnalités suivantes :

- certaines bibliothèques s'appuient sur des « processus légers de niveau système », qui sont alors des unités d'ordonnancement du système d'exploitation sous-jacent au même titre que les processus. D'autres reposent sur des « processus légers de niveau utilisateur », qui sont plus « légers » que les précédents. Ils ne sont pas connus du système d'exploitation mais seulement des processus de l'environnement de programmation parallèle.
- La multiprogrammation des processus légers peut être implantée avec différentes formes de préemption : préemption uniquement quand le processus légers est bloqué (en attente d'un message ou d'un sémaphore), *etc.*
- La politique d'ordonnancement des processus légers sur chaque processeur peut varier : premier arrivé-premier servi, utilisation de priorité, *etc.* Lorsqu'un environnement de programmation parallèle utilise conjointement une bibliothèque de processus légers et une bibliothèque de communication (comme MPI [31]), il doit y avoir compatibilité (plus ou moins forte) entre les deux bibliothèques.
  - si la bibliothèque de communication est « *thread-safe* », elle peut être utilisée de façon concurrente par plusieurs processus légers.
  - Si une opération de communication est « *thread-synchronous* », elle peut bloquer le processus légers qui l'exécute sans bloquer les autres processus légers dans le même contexte.
  - Si la bibliothèque de communication est « *thread-aware* », elle connaît la multiprogrammation des processus légers.

2.5.2 Echange de messages : MPI

La version originale du standard MPI [31] a été créée par le Message Passing Interface Forum (MPIF) et la première version publique a été diffusée en 1994. Dans l'approche adoptée pour la programmation parallèle par la bibliothèque MPI, un ensemble de processus exécutent un programme écrit dans un langage séquentiel dans lequel ont été rajoutés des appels aux différentes fonctions contenues dans la bibliothèque permettant l'envoi et la réception de messages.

Au sein d'un calcul décrit avec MPI, plusieurs processus communiquent ensemble via ces appels aux routines de la bibliothèque. Dans la plupart des implantations de MPI, un nombre fixé de processus est créé à l'initialisation, et, généralement, un seul processus est créé par processeur. Le modèle de programmation parallèle MPI est souvent référencé en tant que SPMD dans lequel tous les processus exécutent le même programme, à distinguer des modèles MPMD dans lesquels les processus peuvent exécuter des programmes distincts.

Les processus peuvent utiliser des opérations de communication point à point pour envoyer un message depuis un processus nommé vers un autre. Un groupe de processus peut également employer des opérations de communications collectives bien utiles pour les opérations de diffusion ou de réduction ou de distribution ou redistribution de données. La bibliothèque MPI permet également l'utilisation de communications asynchrones.

MPI supporte la programmation modulaire. Un mécanisme appelé « communicateur » permet au

programmeur MPI de définir des modules qui encapsulent des structures de communications internes. L'approche adoptée par MPI pour la programmation parallèle est donc entièrement basée sur l'échange de messages entre les processus. De nombreuses routines sont mises à la disposition du développeur pour effectuer différents types d'envois et de réceptions de messages (synchrones, asynchrones, collectifs, point à point) ainsi que des opérations globales (barrière, diffusion, réduction). Par contre MPI ne spécifie pas :

- les opérations relatives à l'utilisation d'une mémoire partagée ;
- les opérations nécessitant des appels systèmes non prévus lors de la spécification du standard MPI (exécutions à distance, messages actifs, interruption de réception) ;
- des outils d'aide à la programmation ;
- des outils de déboguages ;
- un support pour la multiprogrammation à base de processus légers ;
- la gestion des tâches ;
- des fonctions d'entrées/sorties.

Certaines de ces spécifications sont partiellement adoptées dans la nouvelle version MPI-2. Les environnements de programmation parallèle basés sur le modèle d'échange de message se révèlent fastidieux à utiliser quand il s'agit de paralléliser des applications complexes et irrégulières. En effet le programmeur doit gérer à la fois les communications effectuées entre processeurs, ainsi que les synchronisations ou encore l'ordonnancement des tâches de calculs et le placement des données sur les différents processeurs.

2.5.3 Mémoire virtuelle partagée : ATHAPASCAN

Dans cette section, nous présentons en détail le langage ATHAPASCAN. Nous commençons par présenter les objectifs, puis nous aborderons la syntaxe du langage. Enfin nous terminons la section par l'ordonnancement des programmes ATHAPASCAN et rappelant les résultats de complexité obtenus par des travaux précédents [43, 42, 32]. Cette présentation permet de présenter les fondements de nos travaux sur HOMA, présentés dans la seconde partie de ce manuscrit, et sur les extensions apportés à ATHAPASCAN, présentées dans la troisième partie du document.

L'environnement de programmation parallèle ATHAPASCAN [43] est développé au sein du laboratoire ID-IMAG (Informatique et Distribution) dans le cadre du projet INRIA-APACHE. Son développement a débuté en 1996. C'est un interface de programmation parallèle de haut niveau au sens où aucune référence n'est faite par rapport au support d'exécution. De plus, ATHAPASCAN est un langage explicite au sens où le parallélisme est exprimé à l'aide de deux mots-clés (*Shared, Fork*). Enfin, il gère l'ordre d'exécution des tâches et les communications entre processeurs.

ATHAPASCAN a l'avantage d'assurer une bonne portabilité grâce à la possibilité d'utiliser des plates-formes parallèles distribuées. En fait ATHAPASCAN a la propriété de permettre d'exploiter deux concepts, l'échange de message et la mémoire partagée, afin de tirer parti des performances des deux types d'architectures en même temps (à mémoire partagée et à mémoire distribuée).

De plus ATHAPASCAN garantit la sémantique d'accès aux données de la mémoire partagée grâce à des synchronisations implicites entre les tâches. En effet chaque tâche déclare les accès effectués sur une mémoire partagée fournie par la bibliothèque. Cet environnement est en fait composé de deux modules complémentaires : KAAPI [44] et ATHAPASCAN. KAAPI est le moteur exécutif qui permet l'utilisation d'un graphe de flot de données dans un contexte de multi-programmation distribuée.

KAAPI est basé sur un couplage entre différentes bibliothèques de processus légers (threads POSIX) et de communication standards. KAAPI permet de gérer les communications effectuées entre processus, sans se soucier du type de réseau utilisé.

ATHAPASCAN, anciennement appelé Athapascan-1 [84], est l'interface applicative (API) au-dessus d'KAAPI dont nous nous servons pour l'écriture des programmes qui illustrent cette thèse. Il implémente un modèle de programmation de haut niveau basé sur un mécanisme de création de tâches collaborant entre elles par l'accès à une mémoire virtuelle partagée.

2.5.3.1 Syntaxe de programmation d'ATHAPASCAN

Cette section présente la syntaxe du langage ATHAPASCAN comme présenté dans la thèse de F. Galilée [42].

ATHAPASCAN définit une mémoire partagée afin de permettre aux tâches de coopérer. Cette mémoire peut contenir des objets de tout type, et ceux-ci sont déclarés comme des objets standards mais de type `Shared<T>` où `T` représente le type spécifié par l'utilisateur. Le parallélisme est exprimé par création de tâches représentant l'exécution d'une procédure de manière asynchrone. Une tâche est créée par appel à la procédure générique `Fork<>()` de la bibliothèque instanciée avec le type de la tâche à créer (type représentant une fonction classe[14]) qui prend en paramètre la liste des paramètres de la tâche.

Droit d'accès et l'interface. La tâche représente l'unité de calcul en ATHAPASCAN et peut être considérée comme une procédure exécutée de manière asynchrone et ne faisant aucun effet de bord, c'est-à-dire dont la seule interaction avec l'environnement est effectuée à travers ses paramètres. Chaque tâche spécifie au moment de sa création les accès qui seront effectués sur la mémoire partagée au cours de son exécution ou lors de l'exécution de toute sa descendance. Les différentes données qui seront accédées sont spécifiées dans la liste des paramètres de la tâche et la description de ces droits d'accès (lecture `r`, écriture `w`, accumulation `cw`, modification `r_w`) est faite à l'aide d'un mécanisme de typage. La règle générale est qu'une tâche ne peut accéder, soit directement, soit via sa descendance, à un objet pour lequel elle n'a pas déclaré l'accès correspondant (restriction des droits d'accès) : les tâches ne font pas d'effet de bord sur la mémoire. Le typage des paramètres permet de vérifier facilement à la compilation la validité des accès effectués sur les données. Les différents types possibles et l'interface associée sont répertoriés dans les tableaux 2.1 et 2.2.

Sémantique des accès aux données. Une tâche a un fonctionnement entièrement asynchrone par rapport à la tâche qui l'a créée. Cela implique donc, entre autres, que la tâche mère ne peut accéder les résultats de la tâche fille : ces résultats seront exploités par une tâche nécessairement différente. Le système exécutif d'ATHAPASCAN garantit (afin de permettre à d'éventuels ordonnanceurs de faire des estimations fiables sur la durée d'exécution des tâches) que l'exécution d'une tâche a lieu sans aucune synchronisation. Pour cela, une tâche ne pourra débuter son exécution que si toutes les données accédées en lecture sont prêtes, c'est-à-dire que toutes les tâches qui écrivaient sur cette donnée sont terminées.

[14] Il s'agit d'un terme introduit dans la définition de la bibliothèque standard de C++ (STL) qui correspond à une classe possédant un opérateur d'appel de fonction

Droit / mode d'accès `Typage`	**Description**
lecture directe `Shared_r`	Accès restreint à la lecture seule (éventuellement concurrente) par la tâche et éventuellement sa descendance. Les tâches créées ne peuvent requérir qu'un accès en lecture seule sur cette donnée, c'est-à-dire les types `Shared_r` ou `Shared_rp`.
lecture différée `Shared_rp`	Accès en lecture seule par la descendance uniquement. La tâche courante ne peut accéder la donnée. Les tâches créées ne peuvent requérir qu'un accès en lecture seule sur cette donnée, c'est-à-dire les types `Shared_r` ou `Shared_rp`.
écriture directe `Shared_w`	Accès en écriture exclusive par la tâche courante uniquement. Les tâches créées ne peuvent requérir aucun accès sur cette donnée.
écriture différée `Shared_wp`	Accès en écriture seule par la descendance uniquement. La tâche courante ne peut accéder la donnée. Les tâches créées ne peuvent requérir qu'un accès en écriture sur cette donnée, c'est-à-dire les types `Shared_w` ou `Shared_wp`.
écriture concurrente directe `Shared_cw`	Accès en écriture concurrente (à sémantique d'accumulation) par la tâche et éventuellement sa descendance. Les tâches créées ne peuvent requérir qu'un accès en écriture (avec la même sémantique d'accumulation) sur la donnée, c'est-à-dire les types `Shared_cw` ou `Shared_cwp`.
écriture concurrente différée `Shared_cwp`	Accès en écriture seule (à sémantique d'accumulation) par la descendance uniquement. La tâche courante ne peut accéder la donnée. Les tâches créées ne peuvent requérir qu'un accès en écriture (avec la même sémantique d'accumulation) sur la donnée, c'est-à-dire les types `Shared_cw` ou `Shared_cwp`.
modification directe `Shared_r_w`	Accès en lecture et écriture de la donnée par la tâche courante uniquement. Cet accès, qui permet la mise à jour « en place » de la donnée, est exclusif. Les tâches créées ne peuvent requérir aucun accès sur cette donnée.
modification différée `Shared_rp_wp`	Accès en lecture et écriture par la descendance uniquement. Les tâches filles peuvent requérir tout type d'accès sur la donnée.

Tableau 2.1 *Types autorisés lors de la déclaration des paramètres formels d'une tâche. Les types sont composés d'un* ***droit d'accès*** *lecture, écriture, accumulation et modification (`r`, `w`, `cw`, `r_w`) qui spécifie le type d'accès qui sera effectué par le sous graphe engendré par la tâche requérant cet accès, et d'un* ***mode d'accès*** *direct ou différé (`p` pour postponed) qui spécifie si la tâche requérant un droit d'accès va utiliser ce droit ou se contenter de le passer à sa descendance. Cette information permet de déduire les contraintes de localité des tâches puisque les données réellement accédées sont connues.*

La sémantique des accès aux données de la mémoire partagée est définie par rapport à une exécution séquentielle du programme : la valeur retournée par la fonction membre `read()` lors de toute exécution (éventuellement parallèle) est identique à celle retournée lors d'une exécution séquentielle [42]. Bien que cette sémantique lexicographique semble impliquer des contraintes de synchronisa-

Access `Interface`	**Description**
lecture concurrente `const T& read() const;`	Si la lecture est autorisée (`Shared_r< T > x`) l'appel `x.read()` retourne une référence constante sur la valeur contenue dans l'objet partagé.
écriture exclusive `void write( T* );`	Si l'écriture est autorisée (`Shared_w< T > x`) l'appel `x.write( p )` stocke la valeur pointée par `p` dans la donnée partagée.
modification exclusive `T& access();`	Si la modification est autorisée (`Shared_r_w< T > x`) l'appel `x.access()` retourne une référence sur la valeur contenue dans la donnée partagée.
écriture concurrente `void cumul( const T& );`	Si l'accumulation est autorisée (`Shared_cw< F, T > x`) l'appel `x.cumul( v )` accumule `v` dans la valeur de la donnée partagée `x` (à l'aide de la fonction `F`). Dans le cas où la donnée partagée ne contiendrait aucune valeur lors de l'accumulation, une copie de `v` est effectuée à la donnée.

Tableau 2.2 *L'accès à la valeur d'une donnée partagée se fait à l'aide des fonctions d'interface (si le mode et le droit d'accès l'autorisent).*

tion, l'existence de restrictions sur le passage des droits d'accès aux tâches filles permet de garantir que l'exécution d'une tâche peut avoir lieu sans interruption. Cela implique, sur une machine à un seul processeur, que toutes les tâches filles peuvent être exécutées *à la fin* de l'exécution de la tâche mère.

2.5.3.2 Ordonnancement non préemptif

La sémantique d'ATHAPASCAN est donc lexicographique. Une telle sémantique semble requérir des synchronisations : une tâche semble devoir être interrompue pour attendre les valeurs produites par ses filles. Bien que lexicographique, la sémantique d'ATHAPASCAN autorise donc une exécution parallèle et non préemptive des tâches. Parmi ces exécutions, celle ressemblant le plus à l'ordre séquentiel des instructions définit un ordre total sur les tâches : l'*ordre de référence* qui consiste à exécuter le corps de la tâche mère dans son intégralité avant de passer, dans l'ordre où a eu lieu les créations, à l'exécution du corps des filles. Un intérêt pratique de tels ordonnancements est de ne pas requérir l'implantation d'un mécanisme de migration ou de préemption des tâches une fois qu'elles ont débuté leurs exécutions. Les *contraintes de précédence* entre les tâches sont déduites des accès effectués sur les données partagées puis les tâches peuvent être exécutées sans synchronisation interne : leur exécution peut être vue comme atomique.

Modèle de coût. ATHAPASCAN et KAAPI construisent dynamiquement un graphe représentant les accès que font les tâches sur les données de la mémoire partagée. Ces accès sont définis lors de la création de la tâche via le type de ses paramètres. Ce graphe de flot de données permet d'une part de définir (puis garantir) la sémantique des accès aux données et d'associer à l'interface de programma-

tion ATHAPASCAN un modèle de coût. D'autre part de fournir aux politiques d'ordonnancement une information complète sur l'exécution : précédences et localités des tâches et des données.

L'ordonnancement dans ATHAPASCAN/ KAAPI est assuré par un module séparé de la bibliothèque. Cette séparation permet un développement aisé de nouvelles stratégies d'ordonnancement. De plus, l'accès au graphe de flot de données permet de considérer toutes les politiques d'ordonnancement basées sur sa connaissance, comme par exemple les stratégies de type statique qui permettent de minimiser le volume des communications de données nécessaire (ces stratégies sont appelées en cours d'exécution sur une portion de graphe, graphe qui a été construit dynamiquement).

Pour certaines politiques d'ordonnancement, un modèle de coût est associé au modèle de programmation ATHAPASCAN/ KAAPI. Ce modèle de coût est un modèle algorithmique qui prend en compte les communications pour mesurer le temps d'exécution *T* et l'espace mémoire utilisé *S*. Nous rappelons ici les notations et les résultats publiés dans [43]. Les paramètres sont :

- *q*, le nombre de processeurs virtuels émulés sur les *p* processeurs réels de la machine (afin de pouvoir majorer la latence des communications soit par du calcul soit par d'autres communications).
- *h*, le délai d'accès à distance d'un bit de donnée. Ce délai peut être borné en utilisant des processeurs virtuels, comme présenté dans [56] (*h* dépend de *p* et de *q*).
- T_1 indique le temps séquentiel qui est défini à partir d'une exécution séquentielle du programme.
- T_∞ le temps parallèle est la profondeur arithmétique du graphe prenant en compte les poids (le coût de calcul) des nœuds des tâches. T_∞ est alors la limite la plus basse du temps minimal exigé pour une exécution non-préemptive sur un nombre non-borné de processeurs en négligeant le temps de communications (modèle PRAM [34]).
- C_1, le volume total d'accès distant. Cette valeur représente la somme sur tout le graphe d'exécution G des tailles des données accédées en lecture directe.
- C_∞, le volume d'accès distant effectué par un plus long chemin dans ce graphe (selon ce critère d'accès).
- σ, la taille du graphe *G*, c'est-à-dire le nombre de nœuds et d'arêtes.

Le calcul de la valeur T_∞, représentant la longueur d'un chemin critique du graphe, peut être effectué dynamiquement lors de la construction du graphe à partir d'une exécution. Il faut tenir compte dans ce calcul du coût de description du graphe, c'est-à-dire du coût de la tâche mère (la tâche mère étant exécutée intégralement avant l'une quelconque de ses filles).

Il est possible, à l'aide de ce modèle de coût, de garantir la durée et la consommation mémoire de toute exécution en fonction de la stratégie d'ordonnancement utilisée. Par exemple, une politique d'ordonnancement similaire à celle présentée dans [17] permet de garantir les majorations suivantes, et ce pour toute exécution :

$$T_p \leq O\big(\tfrac{T_1}{p} + h(\tfrac{C_1}{p}) + \tfrac{q}{p}O(T_\infty + hC_\infty) + h\tfrac{q}{p}O(\sigma)\big)$$
$$S_p \leq S_1 + qO(T_\infty + hC_\infty)\big)$$

Ce résultat est similaire à celui obtenu par le langage NESL [16], mais tient compte, en plus, des coûts de communications. Il est possible, en utilisant une autre politique d'ordonnancement, d'obtenir une majoration du même type que celle de Cilk [18] comme présenté dans [42] :

$$T_p \leq O\left(\frac{T_1}{p} + h(\frac{C_1}{p}) + h(\frac{q}{p})O(T_\infty + hC_\infty)\right)$$
$$S_p \leq qS_1$$

2.5.4 Bus logiciel d'objets répartis : CORBA

CORBA [67] est un standard d'architecture d'intergiciel à objets repartis basé sur le modèle *client-serveur*. C'est l'acronyme de *Common Object Request Broker Architecture*. CORBA est une norme définie par l'OMG *(Object Management Group)*, consortium ouvert qui réunit des industriels et des universitaires et qui a pour but de proposer un standard d'interopérabilité d'un bus à objets ou promouvant la technologie CORBA. Il existe de nombreuses implémentations de la norme CORBA, dont certaines sont des logiciels libres.

L'architecture définie par CORBA est indépendante du type de machine, du système d'exploitation, et du langage de programmation utilisé. CORBA est composé essentiellement de l'ORB (*Object Request Broker*, courtier de requêtes) qui est le coeur de l'architecture CORBA, et des objets pris en charge par l'ORB. L'ORB assure le transport des requêtes sur le réseau. Du point de vue du programmeur, tout objet CORBA est accédé de façon transparente, comme s'il s'agissait d'un objet local. L'ORB se charge d'intercepter les invocations aux méthodes, de localiser l'objet, d'acheminer par le réseau les invocations avec leurs paramètres et de transmettre les éventuelles valeurs de retour des méthodes.

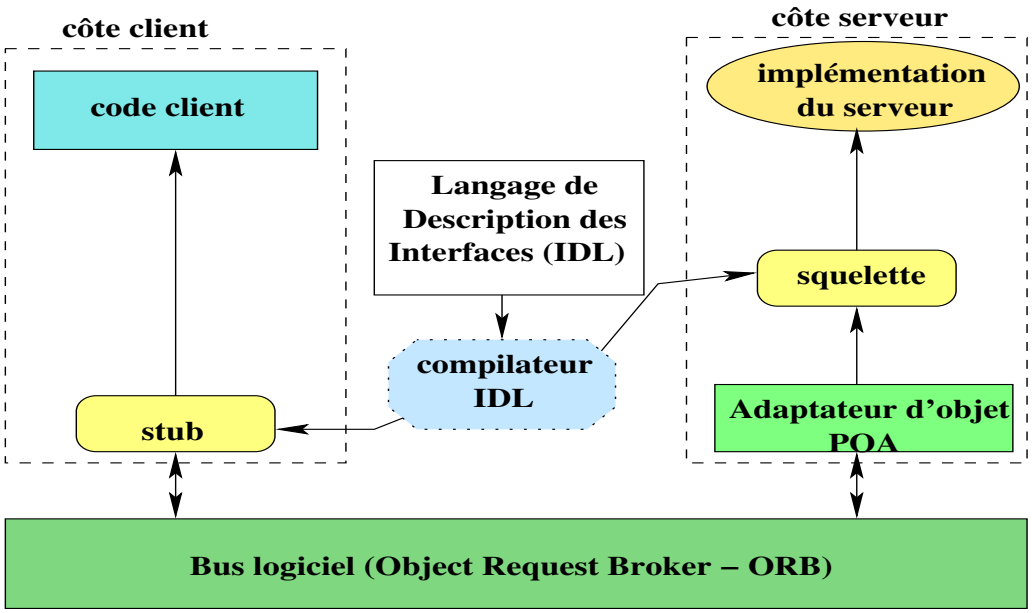


Figure 2.2 *La chaîne de compilation de l'IDL CORBA. Grâce au fichier IDL, le compilateur IDL décharge le programmeur de la définition des stubs du côté du client et du squelette coté serveur. Le concepteur d'un objet CORBA n'a plus qu'à implanter les interfaces du serveur ; l'utilisateur accède à l'objet comme si celui-ci était un objet du langage natif de programmation. L'ORB se chargeant du transfert des requêtes.*

Langage IDL. Les interfaces des objets CORBA sont spécifiées à l'aide du langage IDL *(Interface Definition Language)* qui permet l'indépendance par rapport au langage d'implémentation des objets. Dans la suite de cette section, nous présentons la sémantique d'invocation de méthodes dans IDL CORBA. Ces détails sont intéressants pour l'analyse de ce type de système afin de concevoir des applications qui couplent des codes.

Le langage IDL (Interface Definition Language) permet d'exprimer, sous la forme de contrats, la coopération entre les serveurs et les clients, en séparant l'interface de l'implantation des objets et en masquant les divers problèmes liés à l'interopérabilité, l'hétérogénéité et la localisation de ceux-ci. Un contrat IDL spécifie les types manipulés par un ensemble d'applications réparties, c'est-à-dire les types d'objets (ou interfaces IDL) et les types de données échangés entre les objets.

La figure 2.2 présente le processus de compilation IDL CORBA. Les contrats IDL sont projetés en souches (*stub*) dans l'environnement de programmation du client et en *squelettes* dans l'environnement de programmation du serveur. Le client invoque localement les *stubs* pour accéder aux objets. Les *stubs* construisent des requêtes, qui vont être transportées par le bus, puis délivrées aux squelettes qui les délégueront aux objets.

Un objet qui implante un interface IDL exporte un ensemble de méthodes publiques. La sémantique d'interaction entre le serveur et le client est spécifiée par le mode de passage des paramètres et le mode d'invocation des méthodes.

Mode de passage de paramètres. Les méthodes sont définies dans l'IDL en précisant la direction de passage des paramètres. Un argument de mode *in* d'une méthode signifie que le corps de méthode l'utilisera en lecture seulement et de mode *out* signifie que le corps de méthode produit l'argument. Le mode *inout* signifie que le corps de méthode modifie (lecture et écriture) l'argument. Ces indications permettent au compilateur IDL de CORBA de générer un code efficace pour le transfert de données à chaque invocation de méthode entre un client et un serveur : seules les données qui doivent être lues par le serveur sont transmises du client (paramètres en *in* ou *inout*) en début d'invocation, de même seules les données modifiées ou produites par le serveur sont transmises vers le client après l'exécution du corps de méthode (paramètres en *out* ou *inout*).

Mode d'invocation de méthode. Il y a deux types d'invocation de méthodes dans CORBA. Par défaut les méthodes sont synchrones : le client invoque la méthode et se bloque jusqu'à la réception du message de réponse. Avec l'attribut « *oneway* », on peut déclarer une méthode dite *asynchrone* mais sous la condition que tous les arguments soient définis en mode *in* et que la méthode ne possède pas de valeur de retour. Il existe un autre mode, l'invocation différée (*send_deferred*), mais uniquement disponible avec le mécanisme dynamique DII d'invocation de méthode. Il permet à l'application de continuer son exécution pendant l'exécution de la requête. Elle obtient ultérieurement le résultat soit par une attente bloquante via *get_response*, soit par une scrutation non bloquante via *poll_response*. Pour avoir le même comportement avec les souches IDL, il faut utiliser plusieurs flots d'exécution concurrents (*multithreading*).

2.6 Synthèse

Les modèles de programmation parallèle savent utiliser les réseaux hautes-performances et sont souvent très efficaces sur des architectures spécifiques telles que les supercalculateurs ou des grappes de calculateurs. Ces modèles ne définissent pas les protocoles que l'intergiciel doit utiliser lors d'une communication inter codes. Les modèles ne disposent pas de la notion d'interface. Les interactions entre les codes sont alors difficilement exprimables. Les interactions sont cachées et fixées dans le programme et il est difficile de remplacer un code par une nouvelle version. Le déploiement de l'application est aussi critique car certains de ces modèles sont très statiques. Ces modèles n'offrent pas de concept permettant de gérer l'architecture de l'application. Si chaque code est écrit par des groupes différents, il faut que chaque groupe ait une connaissance précise des autres codes pour réaliser l'interconnexion des programmes de l'application. Cela peut poser des problèmes dans le cas où l'application est composée de codes propriétaires. Enfin, ces intergiciels ne permettent pas à un programme extérieur à l'application de se connecter et de se déconnecter dynamiquement. Dans la pratique ces modèles servent à la programmation d'un unique code parallèle.

Les modèles de programmation distribuée ont un certain nombre d'avantages. Ils permettent de définir clairement les interfaces entre les différents codes de l'application. Ils savent gérer l'hétérogénéité et supportent les connexions dynamiques entre les codes. L'inconvénient majeur de ces modèles est la non-prise en compte de l'encapsulation de codes parallèles. Tout d'abord, il est difficile d'exprimer la notion de communication parallèle entre deux codes distribués. Il est cependant possible de réaliser des communications en passant par des nœuds maîtres dans les deux programmes. Bien que cette solution fonctionne, elle apporte un goulot d'étranglement pour les communications qui est dommageable pour la performance de l'application. De plus, cette solution peut demander une modification considérable des codes parallèles. Ces modèles de programmation ne proposent pas d'opérations collectives pour l'écriture d'algorithmes parallèles. Ces primitives (telles que les barrières par exemple) doivent être implémentées par le concepteur de l'application. Les modèles de programmation distribuée n'offrent pas de gestion de la distribution des données dans un code parallèle. Les intergiciels distribués utilisent très rarement les protocoles spécifiques des réseaux hautes-performances disponibles sur les grappes de calculateurs.

Que ce soit par l'utilisation des modèles de programmation distribuée ou de programmation parallèle, il est possible de concevoir des applications de couplage de codes. L'utilisation directe de ces modèles ne va pas sans contraintes. Dans le cas des modèles de programmation distribuée, la construction de codes parallèles est complexe et les performances de l'application risquent d'être très dégradées. Dans le cas des modèles de programmation parallèle, la non-définition des protocoles, l'absence de structuration du code et le couplage statique amènent aux concepteurs des applications à faire des choix de conception contraignants.

3

Modèles et outils pour le couplage de codes

Sommaire

L'objectif de ce chapitre est d'étudier les caractéristiques des environnements logiciels pour les applications de couplage de codes. Nous commençons par l'approche « coordination, » un modèle structuré de conception d'application par assemblage de composant. Afin d'obtenir le bon niveau de réutilisation et d'hétérogénéité des modules d'application, cette approche se base sur la notions de composants logiciels. Nous pensons que le modèle « coordination des composants logiciels » peut rapprocher les recherches actuelles et futures dans le domaine du couplage de codes en se focalisant sur les normes et les outils de développement des composants parallèles. Nous présentons certains des outils logiciels et des projets de recherche autour du couplage de codes, puis nous concluons ce chapitre par une analyse de ces outils.

3.1 Coordination d'activités

Les bibliothèques de bas niveau telles que MPI, ont été développées pour construire des applications parallèles monolithiques et elles n'offrent, en conséquence, qu'un support limité pour la définition de codes réutilisables par d'autres applications. Une autre limitation concerne le manque de structuration de l'application. Tous les processus de l'application couplée sont au même niveau, et on parle alors d'architecture plate.

Sous le terme « *coordination* » [9, 24, 52, 70, 45] se regroupe un ensemble de modèles ou formalismes et mécanismes qui proposent un moyen d'assembler ensemble des « *composants* » de manière à former une application qui puisse s'exécuter sur des architectures parallèles (ou distribuées). Les composants doivent pouvoir être écrits dans différents langages en utilisant différents paradigmes de programmation. L'objectif est ambitieux et veut offrir une manière de concevoir des applications distribuées ou non, à partir de composants élémentaires.

Dans la suite de cette section, nous étudions le concept de composant logicielle dans l'optique de couplage de codes. Puis nous présentons l'approche de coordination de composants.

3.1.1 Composants logiciels

Le développement d'applications passe par l'utilisation d'un langage de programmation offrant des abstractions structurantes (structure de données, fonctions, objets) simplifiant la mise en œuvre des détails de bas niveau (de manière extrême, l'accès à un champ d'une structure en terme d'instruction load/store des processeurs). Le paradigme de la programmation objet offre une bonne abstraction structurante de programmation par la notion d'objet. Elle permet de découpler les interfaces des objets de leurs implémentations.

Néanmoins, l'expérience montre qu'un objet, en dehors du contexte où il a été créé, est difficilement réutilisable. Le concept de composant logiciel a été introduit [83] afin d'étendre la notion d'objet par la prise en compte du cycle de vie. En particulier, un composant est une unité de déploiement.

3.1.1.1 Composants séquentiels

Parmi les différents modèles à composants issus des intergiciels CORBA (Common Object Request Broker Architecture) [67], COM (Common Object Model Microsoft) [26] ou encore Enterprise

JavaBean Sun [62], et grâce à l'importance des développements utilisant CORBA, le modèle à composant CCM (Corba Component Model) [68] est l'un des modèles qui sera certainement le plus utilisé pour la réalisation d'applications de simulation numérique à hautes performances.

Dans ces différents modèles, un composant est associé à un espace d'adressage. Une communication entre composants correspond à un mouvement de données et/ou à une invocation d'une méthode à distance entre espaces d'adressages. Les composants communiquent par une notion de port, utilisent des interfaces (*require*) et en fournissent d'autres (*provide*).

Typiquement, un composant est constitué d'interfaces et du code associé. Il est associé à un environnement d'exécution (appelé « *container* » dans CCM). Le grain d'un composant dépend des performances du modèle de programmation du composant utilisé et peut aller de la réalisation d'opérateurs élémentaires [12] à un code complet de simulation numérique [7, 12, 64].

3.1.1.2 Composants parallèles

De la même manière qu'un composant permet d'encapsuler une application séquentielle, la notion de composant parallèle vise à définir les interfaces qui permettent d'encapsuler une application parallèle. Peu de travaux de normalisation existent dans ce domaine, citons le CCA Forum [65, 10] (Common Component Architecture) dont l'objectif est de développer un ensemble de concepts et d'interfaces permettant de rendre réutilisables et interopérables les applications parallèles du Département à l'Énergie Américain (DOE). Les définitions actuellement disponibles se basent sur la description d'une interface plutôt que sur un modèle de composant parallèle. Les composants communiquent par une notion de port étendue aux communications collectives.

Les auteurs de PACO étendent leurs travaux sur les objets CORBA parallèles à la définition de composants parallèles, GridCCM [30], pour encapsuler facilement des codes MPI de type SPMD (Single Program Multiple Data). Ces codes travaillent sur des descriptions de données régulières (tableau multidimensionnel) tout en conservant la possibilité d'obtenir des communications parallèles lors des invocations de méthodes sur des objets distribués. En utilisant une description des données des arguments des méthodes, comme celle introduite dans High-Performance Fortran (HPF) [55] (bloc, bloc cyclique, *etc*), un compilateur peut générer les souches clientes et serveurs qui distribuent (ou redistribuent) les données. On peut citer également le projet PARDIS [57] qui travaille aussi sur la notion d'objet CORBA parallèle, utilisée dans l'approche CCA [10]. Ainsi que l'approche de l'OMG, avec Data Parallel Corba [69].

3.1.2 Coordination de composants

Le modèle « coordination de composants » propose de concevoir des applications distribuées à partir de composants élémentaires. Plus précisément, la programmation d'une application parallèle ou distribuée peut être vue comme la combinaison de deux développements distincts : d'une part la programmation des aspects « *calculatoires* » de l'application, c'est-à-dire l'ensemble des processus qui calculent sur des données ; et d'autre part, la programmation des aspects de « *coordination* » qui définissent les communications et les synchronisations entre les composants.

La description des interactions entre composants est importante. Il s'agit de spécifier l'application distribuée, de permettre au support d'exécution de la déployer efficacement, puis de générer les communications entre composants. Le modèle de représentation de ces interactions se base sur des

modèles formels à base d'algèbre de processus, de flots de données, de réseaux de Petri ou de flots de contrôles.

La connaissance des interactions avant ou durant l'exécution de l'application permet de connaître le schéma de communication entre composants et d'utiliser des primitives de communication efficaces (par exemple des communications collectives) ou d'adapter les algorithmes de communication à la topologie du réseau de communication. Elle permet aussi de gérer la configuration de l'application [70, 82].

Les critères de choix d'un langage ou d'un paradigme de coordination peuvent être : la performance des transferts de données entre les composants, la possibilité d'exploiter le parallélisme d'exécution de plusieurs composants, la simplicité d'écriture de la description des interactions, ou encore, l'intégration dans un outil d'administration d'application répartie.

La performance des communications est un point clé dans l'obtention des performances des applications à base de composants. En effet, les architectures actuelles de calcul sont constituées de grappes de calcul interconnectées par des réseaux à longue distance. L'interopérabilité des différentes bibliothèques de communication et la distribution différente des données entre composants sont à l'origine des principaux problèmes rencontrés lors de l'exploitation de ces architectures par des applications de couplage de codes. En particulier dans le cas du couplage d'applications parallèles, il est important de ne pas dégrader les performances par une mauvaise exploitation du caractère distribué des données : *de facto* les données peuvent être communiquées en parallèles, sans transiter par une machine collectrice.

3.2 Outils pour le couplage de codes

Cette section résume les environnements qui offrent certains supports nécessaires pour les applications de couplage de code. Nous classifions ces systèmes dans plusieurs catégories selon l'approche considérée. La première catégorie concerne les approches basées sur une extension des modèles de programmation parallèle. La seconde catégorie regroupe les environnements ou outils qui se basent sur des modèles de programmation du domaine du calcul réparti. Nous l'avons décomposé en deux catégories : celle des approches basées sur des modèles de calcul distribué et celle plus spécifiquement basé sur le modèle RPC appliqué au metacomputing. Enfin la dernière catégorie concerne les approches basées sur des flux de données que l'on retrouve dans les environnement de réalité virtuelle ou pour les applications interactives.

3.2.1 Extensions aux modèles parallèles

Dans certaines recherches, l'objectif principal est d'étendre la programmation parallèle pour prendre en compte la distribution des codes. Il s'agit principalement d'augmenter les fonctionnalités des environnements de programmation parallèle, comme par exemple, de permettre la communication entre plusieurs codes parallèles en utilisant des bibliothèques existantes d'échange de message (comme MPI). PAWS et MpCCI sont dans cette catégorie.

3.2.1.1 MpCCI : échange de message

Le but principal de MpCCI (Mesh-based parallel Code Coupling Interface) [54] est d'établir une bibliothèque qui fournisse une interface facile à utiliser pour le couplage de deux ou plusieurs codes autonomes de simulation, quel que soit le modèle de programmation utilisée. MpCCI est situé entre l'application et la bibliothèque MPI. Au lieu de transférer juste des données d'un processus à un autre, MpCCI tient compte des maillages de discrétisation des méthodes numériques sur lesquels les données sont localisées. Ainsi les outils pour la recherche de voisinage et l'interpolation de valeurs sont fournis.

MpCCI est indépendant de la stratégie de parallelisation utilisée par un code. Il intègre les programmes SPMD de même que MPMD. Basé sur MPI, MpCCI exécute la communication aussi directement que possible entre les processus individuels de codes différents, grâce à une extension de la notation de communicateur MPI.

3.2.1.2 PAWS : partage de données

PAWS (Parallel Application WorkSpace) [11] est une infrastructure logicielle pour connecter les applications parallèles dans un modèle de type composant. Son contrôleur central coordonne les interactions entre les applications parallèles à travers un réseau pour permettre de partager des structures de données parallèles telles que des tableaux multidimensionnels. Les applications utilisent PAWS API pour indiquer quelles structures de données vont être partagées et quand les données sont prêtes à être envoyées ou à être reçues.

PAWS applique un descripteur générique de données parallèles et son défi le plus difficile est de fournir un mécanisme de redistribution des structures de données par un schéma de distribution parallèle. Ce problème peut arriver dans plusieurs situations : quand les applications utilisent un nombre de processeurs différents, ou lorsque différentes stratégies de distribution sont utilisées.

PAWS utilise la bibliothèque de communication Nexus [36]. Quand PAWS établit une connexion entre deux objets de données, le contrôleur est informé et il calcule l'ordonnancement des communications pour le transfert des données. Cet ordonnancement est donné aux deux applications qui utilisent cette information pour l'envoi et la réception des messages. PAWS évite le goulot d'étranglement de mettre en série tous les messages sur un seul canal de communication. Il établit les chemins parallèles de communication Nexus des processus d'une application parallèle aux autres processus de l'application connectée.

Résumé. Les environnements présentés sont dédiés aux machines parallèles avec, cependant, quelques travaux en cours pour viser les infrastructures de type grille. Les couches de communication Ad-hoc (MPI, sockets, shared memory segments,...) sont utilisées avec un couplage statique (à la compilation). Toutes ces environnements ne décrivent pas explicitement le couplage, la normalisation et l'interopérabilité. Cependant elles fournissent des fonctions puissantes telles que l'interpolation entre maillages pour les applications en calcul scientifiques.

3.2.2 Extensions parallèles aux modèles distribuées

Les deux points de vue du parallélisme et des systèmes répartis peuvent être pris en compte dans un modèle de programmation. L'aspect « réparti » des grilles est essentiellement défini par son organisation à grande échelle. En effet, les grilles de calcul vont servir de plate-forme de déploiement pour des applications qui n'étaient pas envisageables auparavant, par leur taille, leur complexité et leur consommation de ressources.

La suite de cette section résume les extensions proposées autour de CORBA pour résoudre ces problèmes d'intégration avec les codes parallèles.

3.2.2.1 « Distributed sequence » de PARDIS

PARDIS [58] est un système contenant le support explicite pour l'interopérabilité des applications parallèles et distribuées. PARDIS est basé sur CORBA. Comme CORBA, il fournit l'interopérabilité entre les composants hétérogènes en spécifiant leurs interfaces dans le langage de description d'interface, IDL CORBA. Cependant, PARDIS étend le modèle d'objet de CORBA en introduisant les objets SPMD représentant des calculs « données-parallèles ».

Les objets SPMD permettent au courtier de réagir directement avec les ressources distribuées d'une application parallèle. Cette capacité assure la livraison de requêtes à tous les processus d'une application parallèle et permet au courtier de transférer les arguments distribués directement entre les processus du client et ceux du serveur. Pour permettre ces transferts d'arguments, PARDIS définit un type d'argument distribué (« *distributed sequence* ») qui est une généralisation de la séquence de CORBA représentant les structures de données distribuées des applications parallèles.

PARDIS est l'un des premiers essais pour introduire le concept d'objet parallèle dans CORBA. PARDIS permet l'utilisation d'appel non-bloquant de méthode basée sur la notion de « *futur* » [58], permettant d'invoquer une méthode, puis ultérieurement d'attendre si besoin ses résultats.

3.2.2.2 Objet parallèle de PACO / PACO++

Parallel CORBA Object (PaCO) [72] est une autre tentative pour introduire le concept d'objet parallèle dans CORBA. Un objet parallèle est de type SPMD. Il est constitué d'une collection d'objets identiques. PACO modifie à la fois le langage IDL pour définir un objet parallèle et l'ORB permettant des les gérer. Il est possible d'associer des types de distribution aux arguments d'une opération d'un objet parallèle. Les distributions sont celles définie dans le langage High Performance Fortran [55].

PaCO++ [73] propose une extension portable de PACO. Afin d'être portable, PaCO++ satisfait différentes contraintes. Tout d'abord PaCO++ ne modifie pas la norme CORBA afin d'être utilisé facilement avec différentes implémentations. Ensuite, les objets PaCO++ peuvent être insérés dans une application CORBA existante sans la modifier. De plus, PaCO++ s'adapte à son environnement d'exécution. Par exemple, PaCO++ utilise en interne des barrières de synchronisations. Si le code utilisateur utilise une bibliothèque de communication telle que MPI ou PVM, PaCO++ peut utiliser la leur fonctionnalité. Les travaux sur PaCO et PaCO++ se poursuivent dans l'étude et le développement de GridCCM.

3.2.2.3 « Data parallele CORBA » de OMG

Plus récemment, l'OMG a adopté une spécification qui définie une architecture pour la programmation parallèle dans CORBA [69]. Au contraire de PARDIS et de PACO, il n'y a pas de modification du langage IDL. Cette approche utilise des mécanismes de plus bas niveau qui nécessitent des modifications de l'ORB tel que l'ajout d'un nouveau type de gestionnaire d'objets : le PPA (Parallel Part Adapter). Un objet parallèle est vu comme une collection d'objets identiques. Une invocation, depuis un ORB standard, requiert l'utilisation d'un proxy qui permet de faire la jonction entre les différents ORBs. Bien que normalisé, peu d'implantation offre le support aux objets « data parallel. »

Résumé. Les travaux présentés ci-dessus montrent différentes approches pour l'intégration d'applications parallèles. La plus aboutie est celle de PaCO++, d'une part parce qu'elle ne modifie pas l'ORB et, d'autre part, parce qu'elle encapsule correctement les distributions des données des codes parallèles de type régulier. En revanche elle ne permet pas de créer des flots de contrôle parallèle à la différence de l'approche PARDIS avec la notation de *futur*. Enfin soulignons qu'avec ces approches il n'existe pas une séparation claire entre les schémas de communications et de contrôles, ce qui implique généralement que le programme de couplage des différents codes est fortement lié aux codes. En résumé, ces approches échouent sur l'assemblage à hautes performances des composants parallèles réutilisables.

3.2.3 Metacomputing basé sur le modèle RPC

RPC est un modèle simple à utiliser pour construire des applications distribuées. Cependant, l'utilisation à grande échelle de ce modèle pose les défis importants en ce qui concerne la performance. Dans le cas où il existe plusieurs serveurs pour répondre à chaque client, l'ordonnancement des invocations devient important. Ainsi la localisation de données est un défi majeur pour ce type d'ordonnancement dans le cas d'un ensemble d'invocations avec les dépendances de données. Dans cette section nous présentons les systèmes Netsolve, Ninf et DIET.

3.2.3.1 Netsolve/ Ninf : « Request Sequencing »

Netsolve [25] et Ninf [80] sont les systèmes de metacomuting basés sur RPC pour créer des bibliothèques numériques qui peuvent être appelées à distance d'une manière transparente. Le but d'un tel système n'est pas de construire un environnement complet pour les nombreux types possibles d'applications de calcul, il s'agit d'offrir un ensemble d'interfaces faciles à utiliser pour « globaliser » les applications existantes.

Ces systèmes contiennent trois entités :

- Le **Client** qui a besoin d'exécuter quelques fonctions à distance. En plus des programmes C et Fortran, le client peut être un environnement interactif, tel que *Matlab* ou *Mathematica*.
- Le **Serveur** exécute des fonctions demandées par les clients. Un serveur peut être parallèle et les fonctions exécutées peuvent être arbitrairement complexes.
- L'**Agent** est le point de convergence de ce type de système. Il maintient une liste de tous serveurs disponibles et il fait les choix des ressources pour chaque demande des clients.

Le principal problème pour l'obtention des performances est alors d'équilibrer la charge de calcul tout en minimisant les communications entre clients et serveurs. Dans Netsolve et Ninf, le concept « Request Sequencing » [33] a été introduit afin de permettre de faciliter le calcul d'un bon ordonnancement des invocations sur les différents serveurs, tout en permettant au système de minimiser les communications entre serveurs. Seules les données en entrée et en sortie de la séquence sont communiquées au client, les autre sont des données temporaires qui peuvent être détruites après utilisation.

3.2.3.2 DIET : multi-agents

NetSolve et Ninf ont un ordonnanceur centralisé qui peut devenir un goulot d'étranglement quand beaucoup de clients essaient d'accéder à plusieurs serveurs. De plus quand les réseaux sont extrêmement hiérarchiques, l'emplacement de l'ordonnanceur a un grand impact sur la performance. DIET (Distributed Interactive Engineering Toolbox) [21] proposent une architecture hiérarchique de composants pour construire ce type d'applications.

DIET offre un tel service à une très grande échelle. Un client qui a un problème à résoudre doit pouvoir obtenir une référence au serveur qui est le mieux adapté pour lui. DIET est conçu pour tenir compte de l'emplacement des données dans l'ordonnancement des travaux. Les données sont gardées aussi longtemps que possible sur (ou près des) les serveurs de calcul afin de minimiser les temps de transfert.

L'ordonnanceur est réparti à travers une hiérarchie de « Local Agents » et de « Master Agents ». Les détecteurs NWS[1] [85] sont placés sur chaque nœud de la hiérarchie pour recueillir les disponibilités des ressources, les indicateurs de charges ainsi collectés sont utilisés par un outil de prédiction d'exécution, nommé FAST [23]. Le service de base de données est fourni par SLiM[2].

Résumé. Ces systèmes permettent aux utilisateurs de résoudre des problèmes scientifiques complexes sur des serveurs distribués. Netsolve et Ninf ont développé leur propre support de communication à la différence de DIET qui est basé sur la technologie CORBA.

Ces systèmes impliquent quelques modifications pour utiliser les services proposés. L'ordonnancement d'une seule requête est leur objectif premier. Dans Netsolve et Ninf, le *Request Sequencing* permet l'optimisation des communications sur une séquence de requêtes mais, cette démarche demande une aide explicite du programmeur. Il n'existe pas de support efficace pour exploiter les objets de données parallèles.

3.2.4 Systèmes orientés flux de données

Une sémantique stricte pour la synchronisation et la consistance ne peut pas exploiter l'intérêt à autoriser des interactions plus faibles et ainsi cela affecte défavorablement la performance et le passage à l'échelle du système. Il existe des applications qui ont des contraintes de temps réel dans leurs spécifications. La simulation d'environnements virtuels interactifs, la reconnaissance interactive de la parole, et la vision interactive sont des exemples de telles applications. La nature interactive de telles applications utilise la synchronisation comme un mécanisme de « contrôle de vitesse » pour adapter l'exécution des diverses parties de l'application. Cependant, il n'existe pas de manière simple

[1] Network Weather System
[2] Scientific Libraries Metaserver

au programmeur de représenter les conditions temporelles dans de telles applications. Stampede [75, 76] et FlowVR [5] sont des environnements pour le développement d'applications distribuées à base de flux de données.

3.2.4.1 Stampede : échange de messages temporels

Stampede [75, 76] est un système de programmation parallèle pour les applications de calcul y compris la vision interactive, la parole et la collaboration multimédia. Les défis de ce système sont la communication, la synchronisation, et la gestion de tampon dans la programmation de telles applications orientées « *stream* » (flux de données) avec des contraintes de temps réel. Les threads sont connectés par des canaux transportent des flux de données, chaque donnée est identifiée par une estampille temporelle.

Dans Stampede, un programme est une collection dynamique de threads communiquant sur des canaux des données temporellement estampillées. Les threads peuvent être créés pour être exécutés n'importe où dans la grappe. Les canaux peuvent être créés n'importe où dans la grappe et ont des noms uniques. Les threads peuvent se connecter à ces canaux pour faire des entrée/sortie via les opérations « put / get ». Une estampille temporelle est utilisée comme un nom pour une donnée qu'un thread met dans ou obtient d'un canal. Le système d'exécution de Stampede s'occupe de la synchronisation et de la communication, et il gère le stockage des données dans les canaux.

Dans Stampede deux critères sont contrôlés automatiquement par le système. Le premier, appelé « *Space* » (espace), mesure la mémoire occupée par les données dans les canaux. Le second, appelé « *Time* » (temps), mesure le temps d'occupation des ressources. Le système contrôle ces critères en garantissant qu'une donnée avec une estampille ancienne est libérée automatiquement et qu'elle n'utilise pas de ressource, tout en garantissant un traitement interactif des données.

3.2.4.2 FlowVR : partage de données temporelles

FlowVR [5] est un intergiciel pour les applications distribuées de réalité virtuelle (VR) sur les environnements de grappe ou grille. FlowVR permet de coupler des codes hétérogènes et il est orienté-composant pour favoriser la réutilisation de code. Les paradigmes classiques de communication se focalisent sur une approche synchrone (le canal FIFO) ou une approche asynchrone (« *sampling* »), cependant FlowVR permet un grand nombre de politiques intermédiaires pour améliorer la performance de l'application afin de masquer les latences tout en garantissant la vitesse de rafraîchissement.

Pour améliorer la latence et la vitesse de rafraîchissement, les applications de VR peuvent profiter d'un modèle d'échange de données basé sur « *sampling* ». Le producteur met à jour des données dans un tampon partagé lu de manière asynchrone par le consommateur. Quelques unes de ces données peuvent être perdues si le consommateur est plus lent que le producteur. Cette pertes de données peut améliorer la performance, mais la cohérence d'application peut ne pas être maintenue. Ceci peut être acceptable dans certaines applications. C'est par exemple utilisé dans les systèmes couplant simulation et visualisation qui s'exécutent à différentes fréquences (à peu près 1000 Hz et 60 Hz respectivement).

Dans FlowVR, une application de VR est vue comme ensemble de modules distribués échangeant des données. Chaque module itère indéfiniment, consommant et produisant les données. Les modules ne sont pas conscients de l'existence d'autres modules, le moteur de FlowVR s'occupe du mouvement

des données entre les producteurs et des consommateurs. Ceci permet une interface de programmation (API) relativement simple.

Chaque message est associé avec une liste d'estampilles et des petites données utilisées pour le routage ou le filtrage des messages. Le réseau de FlowVR permet de construire des communications collectives complexes, une caractéristique désirable pour le couplage efficace de plusieurs modules parallèle écrits dans un environnement de programmation parallèle.

Résumé. Stampede et FlowVR offrent des interfaces explicites pour le couplage de codes. Elle sont basées sur des couches de communication Ad-hoc qui permettent le couplage statique (à la compilation) de codes parallèles ou séquentiels.

3.3 Conclusion

Les outils présentés dans ce chapitre montrent la diversité des applications de couplage de codes. Il paraît difficile de pouvoir synthétiser ces fonctionnalités dans un seul modèle pour le couplage de codes. De nombreux travaux, ainsi que notre travail, tentent de promouvoir une approche basée sur de la coordination d'activités parallèles et distribuées. Dans cette perspective, le chapitre suivant présente les différentes problématiques soulevées par cette approche en montrant les contributions de notre travail présentées dans les parties suivantes de ce mémoire.

4

Problématiques du couplage à hautes performances de codes

Sommaire

L'objectif de ce chapitre est de présenter la problématique de notre thèse, les solutions proposées et les résultats obtenus. Nous analysons les propriétés de deux modèles de conception des applications concurrentes, « l'appel de procédure à distance » et « la mémoire partagée » dans l'optique d'une utilisation pour les applications de couplage de codes. Nous nous focalisons essentiellement sur les caractéristiques pour les hautes performances.

4.1 Introduction

Dans les chapitres précédents, nous avons présenté les applications ciblées et les approches de programmation pour le couplage de codes. L'hétérogénéité et la nécessité d'accès aux différentes ressources informatique sont les propriétés naturelles d'une application de couplage de codes et c'est pour cela que les grappes ou les grilles informatiques sont considérées comme des architectures cibles capables d'apporter la puissance nécessaire. La première application, SIMBIO (section 1.1.1.1 page 2), est basée sur le couplage de codes parallèles encapsulés sous forme d'objets repartis CORBA. La deuxième, Sappe (section 1.1.1.2 page 3), est une application orientée flux de données. Parmi les différents modèles de programmation présentés, nous intéressons à « l'appel de procédure à distance » ou RPC (section 2.2.1 page 14) et « mémoire partagée » (section 2.2.2 page 14). Bien que ces choix soient liés à nos motivations, la facilité de programmation et l'utilisation de grilles informatiques sont importantes.

Ce chapitre analyse les deux modèles considérés pour le couplage de codes en focalisant sur les caractéristiques des hautes performances. Nos critères pour les performances sont essentiellement la vitesse d'exécution d'un système et le passage à l'échelle. Elles sont caractérisées par des grandeurs appelées *métriques*. Pour une application donnée, une métrique usuelle est le temps d'exécution en fonction de la taille du problème. Pour les réseaux, les métriques usuelles sont le débit et la latence. Le débit représente la quantité d'information par unité de temps et la latence est le temps nécessaire pour transmettre la quantité d'information minimale permise par le réseau.

Notons ici que dans cette thèse nous nous basons sur les modèles de programmation orientés à objets. C'est pour cela que nous utilisons dans la suite de ce manuscrit le terme « méthode » à la place de « procédure » et respectivement « invocation » pour « appel » et « invocation de méthode (RMI) » pour « appel de procédure (RPC). »

4.2 Problèmes de l'invocation de méthode à distance (RMI)

L'utilisation d'une grille de calcul basée sur l'invocation de méthode à distance (RMI) est un cadre populaire pour les applications de couplage de codes. Pourtant il existe certains défis pour construire facilement de telles applications à hautes performances. Les sémantiques associées au protocole RMI et les environnements exécutifs supportant les RMI sont considérés dans notre analyse. Dans tout le reste de cette thèse nous basons nos travaux sur CORBA en ce qui concerne les RMI. Néanmoins la plus part des résultats restant corrects dans le cas d'autres environnements RPC.

4.2.1 Problèmes sémantiques des RMI

Afin de faciliter l'utilisation des RMI, ses sémantiques de contrôle et de communication sont très simples. Le problème majeur est que les flots de contrôle et les flots de communication sont inséparables. Ceci peut imposer des attentes inutiles chez les appelants et des communications inutiles de données. La suite de cette section présente dans le détail ces problématiques.

4.2.1.1 Sémantique de contrôle et la synchronisation

L'une des difficultés principales vient de la sémantique de RMI entre un client et un serveur. Une invocation est surtout une instruction de blocage : le client attend jusqu'au retour des valeurs du serveur. Les invocations non-bloquantes sont souvent limitées aux méthodes qui n'ont pas de valeurs de retour. Donc, afin de produire des flot parallèles de contrôle (entre l'appelant et l'appelé), un programmeur doit mélanger des invocations bloquantes avec du calcul multi-threadé, ou bien il doit mélanger des invocations non-bloquantes avec une autre façon pour contrôler les valeurs de retour du serveur (par exemple en utilisant le modèle d'exécution « event-driven », le concept « futur » comme dans PARDIS, *etc.*). En tous cas, la manière naturelle d'invoquer une méthode doit être adaptée.

Afin de surmonter ces limitations, nous proposons dans le chapitre 5 (section 5.2.2) une approche automatique permettant une exécution non-préemptive efficace nommée « invocation-par-continuation », qui se base sur une nouvelle manière de compiler les stubs clients et les patrons serveurs.

4.2.1.2 Sémantique de communication des données

La sémantique de passage des paramètres couple les flots de contrôle et de communication. Tous les paramètres effectifs d'un paramètre formel d'*entrée* d'une méthode sont communiqués du client au serveur. Tous les paramètres en *sortie* sont communiqués en retour au client. Ce mode de passage des paramètres permet l'optimisation des communications pour une invocation entre un client et un serveur. Cette optimisation locale (par rapport à une invocation) n'implique pas une optimisation globale (sur un ensemble d'invocations) quand un client invoque plusieurs méthodes sur une série de serveurs pour réaliser une simulation complexe. Tous les paramètres sont communiqués entre les serveurs en passant par le client même s'il n'a pas besoin d'eux. Le client devient le goulot d'étranglement pour l'utilisation de beaucoup de composants. Le problème est aggravé dans le cas d'une simulation itérative. Ainsi, la communication devrait être optimisée en considérant un ensemble d'invocations.

La section 5.2.3 du chapitre 5 propose une approche automatique pour communiquer ces paramètres des invocation en permettant un transfert direct entre les serveurs : elle y est nommée « communication-par-nécessité. »

4.2.2 Problèmes des supports exécutifs de RMI

Afin d'obtenir des hautes performances, les environnements basés sur modèle RMI doivent pouvoir exploiter tout le parallélisme entre les invocations de méthode.

4.2.2.1 Extraction du parallélisme

L'exécution parallèle d'un ensemble d'invocations de méthodes nécessite un support des deux cotés : du côté serveur et du côté client. Des exécutions concurrentes de méthodes sur différents serveurs sont possibles avec la plupart des systèmes existants (comme le multi-threading dans CORBA) qui fournissent le support nécessaire. Cependant, les invocations concurrentes de méthodes nécessitent la gestion des dépendances de données entre toutes les invocations du côté client afin de respecter la sémantique initiale. Les méthodes communiquent à travers des paramètres effectifs (connus lors des invocations du côté du client) et à travers les états partagés au sein de même composant serveur (à connaître lors d'implantation des serveurs).

- **Dépendances liés aux paramètres effectifs.** Le mode de passage des paramètres décrit la dépendance entre les invocations. Par exemple, une donnée qui est passée comme une sortie à une méthode et puis comme une entrée à une autre, établie une dépendance entre ces invocations. Dans la section 5.2.1 du chapitre 5, nous proposons une technique de compilation exploitant le parallélisme entre les invocations par l'utilisation d'un graphe de flot de données entre les invocations.
- **Dépendances liés aux états partagés au sein d'une même composant serveur.** Les méthodes au sein d'un même composant peuvent communiquer aussi part l'accès aux états du composant. C'est le langage de description d'interface qui doit permettre d'exploiter cette information. Pourtant les systèmes existants n'offrent pas cette possibilité. Ceci nous oblige à une exécution séquentielle des invocations sur un objet serveur.

4.2.2.2 Exploitation des « données parallèles »

L'intégration de codes parallèles dans le modèle RMI nécessite l'encapsulation efficace de ce type de codes. Le parallélisme de données est une approche populaire pour la conception de codes parallèles. L'exploitation des « données parallèles » impose les défis importants. Premièrement, une donnée distribuée, qui serait paramètre d'une invocation, doit être connue par l'intergiciel au moment de l'exécution afin de lui permettre de la redistribuer ou bien de collecter les sous parties en vu de l'invocation.

Cependant, les codes de calcul étant souvent parallèles, il est souhaitable de développer des modèles d'objets ou de composants qui, en plus d'être répartis, prennent aussi en compte le parallélisme. Des exemples d'extension vers le parallélisme de modèles à composants ou à objets sont PARDIS [58], PaCO [72], PaCO++ [73], CCA [10] ou encore *Data Parallel* CORBA [69], présentés précédemment. La caractéristique principale de ces modèles est de combiner à la fois des aspects du calcul parallèle et des aspects du calcul réparti, aussi bien au niveau des ressources utilisées que du modèle de programmation. Nous pouvons les voir comme une généralisation et une rencontre des systèmes parallèles et répartis.

4.2.2.3 Prédiction de l'exécution

Afin de mieux ordonnancer les invocations selon le temps d'exécution et l'espace mémoire, il est essentiel de connaître à l'avance les futures invocations. En utilisant ces informations, l'algorithme d'ordonnancement peut optimiser le flot de contrôle (surtout dans les cas où existent plusieurs serveurs pour répondre aux clients) et le flot de communication (éviter les transferts inutiles ou doubles). La

section 5.2.1 du chapitre 5 montre comment il est possible de résumer ces informations dans un graphe de flot de données.

Dans la section 5.3 du chapitre 5, à l'aide de KAAPI, le noyau exécutif de flot de données, nous présentons les résultats théoriques de l'ensemble de solutions présentées sous la forme d'un modèle de coût prédisant l'exécution sur les machines homogènes d'une classe d'application distribuée en CORBA.

4.3 Problèmes liés aux accès aux données partagées

La mémoire partagée est un modèle de programmation facile à utiliser et efficace au moins sur les machines avec une mémoire physiquement partagée (SMP) que sur les machines distribuées si les informations des accès aux objets sont connues à l'avance. Pourtant l'utilisation de ce modèle pour le couplage de code implique certains défis que nous résumons dans cette section.

4.3.1 Partage de données temporelles

Dans le chapitre 1 (section 1.1.1.2 page 3), nous avons présenté une application qui nécessite l'échange de données temporellement estampillées. Nous appelons ce type de données **« données temporelles »** car le temps (ex. le temps d'accès) est l'identificateur des données.

4.3.1.1 Consistance temporelle

Le terme *« consistance spatiale »* [75] est utilisé pour une sémantique de synchronisation purement basée sur l'ordre des accès (à connaître lors de compilation) qui établie une interaction fixe quelles que soient les conditions d'exécution. L'accès aux données temporelles tolère un certain degré de liberté par rapport de l'état partagé lors de l'exécution. Les auteurs de [75] définissent ces types de synchronisation sous le terme « consistance temporelle. » La « collection temporelle » d'objets partagés (chapitre 7) étend le modèle « donnée partagée » afin de décrire une telle consistance temporelle.

4.3.1.2 Latence liée à la distance temporelle des accès

Dans certains types de consistance temporelle, l'accès aux données doit respecter certains conditions temporelles déterminées par la nature de l'application. Ceci peut imposer des attentes inutiles pour la satisfaction de ces conditions. Par exemple, une lecture qui attend une valeur produite par une écriture selon un attribut de temps, ou bien une écriture qui attend la satisfaction de certains critères (*ex.* la libération de l'espace d'une zone tampon). La collection temporelle (chapitre 7) permet de découpler la déclaration d'accès à un objet partagé de l'accès effectif à sa donnée, c'est-à-dire que nous permettons au moteur exécutif sous-jacent de recouvrir les latences d'accès par du calcul, si le matériel le permet effectivement.

4.3.2 Accès à un ensemble de données partagées

L'exécution de programme parallèle sur les machines distribuées nécessite que le modèle de programmation puisse exprimer la distribution des données partagées (ou exploiter les *« données parallèles »*). Une approche de haut niveau garantit l'intégralité fonctionnelle des données partagées (le modèle SPMD d'objet parallèle) mais une approche bas niveau permet au concepteur d'associer les différents calculs sur les parties d'une donnée partagée (le modèle MPMD d'objet parallèle). La « collection spatiale » d'objets partagés (chapitre 8) offre un support de bas niveau respectant la sémantique d'accès à ses éléments.

II

HOMA : composition à hautes performances de services repartis

5

Ré-ordonnancement automatique des invocations de services

Sommaire

Ce chapitre présente l'environnement HOMA *pour ordonnancer des invocations de méthodes sur une architecture à mémoire distribuée à grande échelle afin d'exploiter leur parallélisme. Nous présentons tout d'abord les défis des environnements basés sur RMI pour construire facilement des applications à hautes performances. Ensuite, nous décrivons les idées de* HOMA *permettant une exécution parallèle d'un ensemble d'invocations, sans changer leurs sémantiques initiales. Enfin, une analyse théorique du coût d'exécution et de l'espace mémoire prédisant les performances d'*HOMA *sur des grappes homogènes de calcul est fournie en se basant sur les résultats utilisés du support exécutif.*

5.1 Introduction

Les simulations multi-échelles et multi-physiques émergent comme des solutions pour atteindre de haute précision dans les simulations de système physiques complexes. La mise en application des différentes physiques, techniques de calcul, et modèles de programmation demandent que ces programmes soient développés comme une collection de composants logiciels indépendants. Par exemple, [60] rapporte une simulation de la future génération d'avions qui nécessite plusieurs centaines de composants logiciels et consommera beaucoup de temps CPU. Ces simulations doivent alors exploiter la puissance agrégée de ressources dispersées des groupes disponibles : de tels programmes basés sur un modèle à composant doivent pouvoir exploiter une grille de calcul [37].

Plusieurs environnements ont été proposés de construire des applications pour une grille de calcul en fournissant des services pour encapsuler les composants (serveur). Un *Network Enabled Server* système (NES) [79] est un environnement de « Metacomputing » où un client demande les services de serveurs en utilisant l'appel de procédure à distance (RPC). De nombreux projets de recherche adoptent ce modèle fondamental de calcul : Netsolve [25], Ninf [80], DIET [22]. L'ordonnancement pour les hautes performances (pour une seule ou un ensemble d'invocations), l'extraction automatique de parallélisme entre plusieurs invocations et l'optimisation des mouvements de données représentent certains des défis principaux de tels systèmes.

Ce chapitre présente l'environnement HOMA [46, 47, 48, 49, 50] pour ordonnancer des invocations de méthodes sur une architecture à mémoire distribuée à grande échelle. La conception de HOMA est motivée par les simulations numériques multi-physiques construites par assemblage de composants (couplage de codes). La nature itérative et dynamique des algorithmes de couplage pour ces simulations numériques oriente vers une analyse des compositions à l'exécution avec gestion des dépendances de données. HOMA a pour cible les applications utilisant le standard CORBA de OMG. Le standard a été utilisé avec succès dans plusieurs projets de simulations numériques [12, 22, 60]. De plus, quelques implémentations à hautes performances sont disponibles [81, 4], ce qui fait de CORBA un outil majeur pour construire des applications distribuées à hautes performances sur grille de calcul.

Les faiblesses des environnements basés sur RMI pour construire facilement des applications à hautes performances sont les suivantes.

– La première difficulté provient de la sémantique des invocations de méthodes entre un client et un serveur. Une invocation est surtout une instruction bloquante : le client est en attente jusqu'à la réception des valeurs de retour du serveur. CORBA limite les invocations non-bloquantes aux méthodes qui n'ont pas de valeurs de retour. Afin de produire un flot parallèle de contrôle, un

programmeur doit mélanger invocations bloquantes avec la création de flot d'exécution (calcul multi-threadé), ou alors des invocations non-bloquantes avec une autre manière de contrôler les valeurs de retour du serveur (callback, le modèle d'exécution *event-driven*, le concept de « futur » [57], ...). En tous cas la façon naturelle de programmer avec des invocations de méthodes doit s'adapter.

– La deuxième limitation est due à la sémantique de communication des paramètres effectifs dans une invocation RMI : tous les paramètres effectifs d'un paramètre d'entrée formel d'une méthode (**in** dans CORBA) sont communiqués du client au serveur ; tous les paramètres de sortie (**out** dans CORBA) sont communiqués en retour au client. La *direction* d'un paramètre défini dans le langage de description d'interface IDL COBRA permet l'optimisation des communications pour une invocation entre un client et un serveur. Cette optimisation est locale (les invocations d'un client vers un serveur) et ne considère pas l'optimisation globale quand un client invoque plusieurs méthodes sur un ensemble de serveurs pour exécuter une simulation complexe : les données sont retransmises au client même si celui-ci ne sert que de « passerelle » pour les données en entrée à un serveur. Le client devient le goulot d'étranglement lors de passage à l'échelle de l'application avec un nombre important de composants. Le problème est aggravé dans le cas des simulations itératives.

HOMA est un prototype logiciel dont le but est de fournir un environnement qui permette de ré-utiliser des composants logiciels et de les assembler afin de construire une application distribuée à hautes performances pour la simulation numérique. Il est basé sur la technologie CORBA [68]. HOMA suppose implicitement que l'application implique de nombreux composants distribués sur plusieurs ordinateurs. Dans la section suivante, nous décrivons dans premiers temps, les idées permettant une exécution à hautes performances d'un ensemble d'invocations sans changer la sémantique initiale de CORBA. Enfin, une analyse théorique du coût d'exécution prédisant les performances d'HOMA sur des grappes homogènes de calcul est fournie.

5.2 Caractéristiques d'HOMA

Cette section décrit les trois caractéristiques principales pour les hautes performances d'HOMA. Afin que HOMA puisse prédire (à l'exécution) le futur proche d'un calcul, le support exécutif d'HOMA capture toutes les invocations sans les exécuter. Lors de cette étape un graphe de flot de données est construit qui représente les dépendances entre les invocations et les données. Ensuite HOMA permet une exécution non-préemptive efficace afin de contrôler l'usage des ressources à l'exécution. De plus, pour éviter le transfert inutile des données, HOMA implante une communication paresseuse des paramètres effectifs. Toutes ces caractéristiques sont transparentes à l'utilisateur final grâce au compilateur IDL d'HOMA présenté dans le chapitre 6.

5.2.1 Interprétation abstraite

Afin d'atteindre une exécution efficace sur une architecture parallèle, la connaissance des dépendances de données associées à l'application apparaît comme le point clé pour calculer un bon ordonnancement. HOMA contrôle à l'exécution « l'interprétation abstraite » pour obtenir les informations concernant les paramètres des invocations. A la compilation du contrat IDL, HOMA produit les

nouveaux *stub* client et *squelette* serveur qui permettent de capturer le type et le mode de passage de chaque paramètre impliqué dans chaque invocation. A l'exécution, les instructions du flot de contrôle du programme sont interprétées de manière standard, alors que les invocations sont interceptées pour décrire le graphe de flot de données de leurs exécutions.

Le graphe de flot de données est biparti : les nœuds forment deux ensembles ; les « tâches » et les « données ». Un nœud de type « tâche » représente un appel à une méthode sur un objet d'un serveur. Les arcs d'entrée sont des paramètres d'entrée (`in` dans le contrat IDL), les arcs de sorties sont des paramètres sorties (`out`). Un nœud de type « données » représente une version d'une variable. La figure 5.1 représente le graphe de flot de données d'un programme simple. A gauche de la figure, le mode de passage des paramètres est présenté en *superscript*. Chaque nœud du graphe a des attributs : le site d'exécution de la tâche (le site où l'objet serveur est instancié) et la taille des données. De plus, chaque nœud a un état (prêt/non-prêt), cela représente le flot de données en respect avec les dépendances. d'accès aux paramètres.

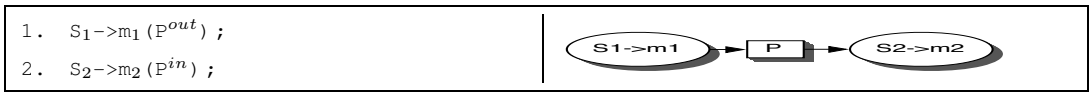


Figure 5.1 *Gauche : deux invocations. Droite : graphe de flot de données associé construit à l'exécution.*

Une donnée est prête si elle vient d'être créée ou si une tâche qui possédait un accès en modification ou écriture a fini de s'exécuter. Une tâche est prête si toutes ses données d'entrée sont prêtes. La génération des *stub*s clients et des *squelette*s serveurs encapsule chaque paramètres d'invocations dans une référence globale comme fourni par support d'exécution d'ATHAPASCAN (chapitre 6). Un tel objet sert de *futur* pour les résultats des méthodes qui ne sont pas exécutées. L'interprétation abstraite détecte les opérations de lecture à l'exécution et bloque le flot de contrôle jusqu'à la disponibilité des données. Ce concept est aussi utilisé, mais de manière explicite, dans PARDIS [57] pour permettre un parallélisme entre les invocations.

5.2.2 Exécution non-préemptive efficace

Après avoir ordonnancé le graphe de flot de données obtenu par interprétation abstraite, le support d'exécution d'HOMA évalue chaque tâche en respectant les contraintes de flot de données. La tâche contenant l'invocation à distance d'une méthode synchrone est très lente en comparaison des autres et limite ainsi l'efficacité des codes parallèles. L'invocation bloquante d'un serveur est fractionnée en deux invocations non bloquantes. Cette transformation est faite dans le *stub* client et le *squelette* serveur générés par le compilateur IDL HOMA. Nous appelons cette transformation « *invocation par continuation* » [49] définie comme la version adaptée du principe « *attente par nécessité* » [20]. La figure 5.2 présente les différences entre une invocation « normale » et une « invocation par continuation ».

L'exécution de la première tâche invoque la première méthode non-bloquante, initialise l'invocation sur le serveur et ajoute une tâche dans le graphe de flot de données pour la *continuation*. La tâche de continuation devient automatiquement prête lorsque le serveur a fini l'invocation de la méthode non bloquante : il signale, par une invocation de méthode non-bloquante, la tâche de continuation. Le fait le plus important est que cette transformation permet une exécution non-préemptive [43] des

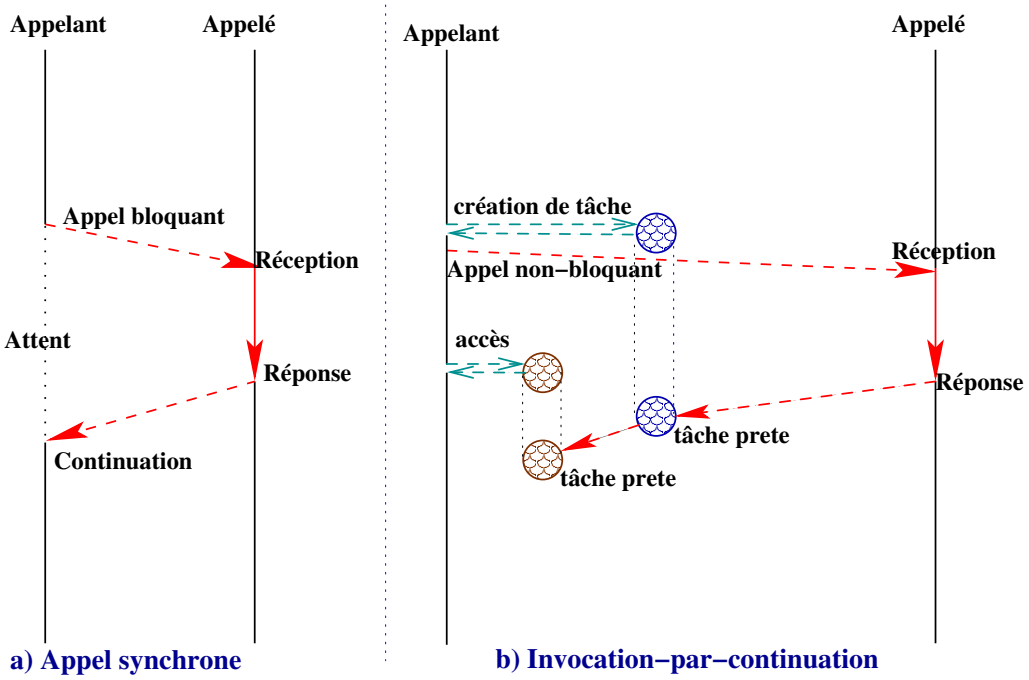


Figure 5.2 *Invocation-par-continuation en comparaison avec l'invocation bloquante.*

tâches avec une exécution efficace : étant donné que les tâches ne commencent jamais leurs exécutions avant d'avoir leurs paramètres d'entrée prêts, tous les processeurs restent actif tant qu'il existe des tâches prêtes dans le graphe. Par exemple, il est possible d'ordonnancer un graphe de flot de données en utilisant un seul fil (thread) de contrôle pour invoquer les méthodes sur plusieurs serveurs en parallèle. Ce mécanisme est présenté plus en détail dans le chapitre suivant.

5.2.3 Mouvement automatique des données

En raison de la représentation d'une exécution par un graphe de flot de données d'une séquence d'invocations, il est possible de générer uniquement les communications nécessaires entre un client et les serveurs. Le compilateur IDL HOMA permet de réaliser automatiquement les transformations nécessaires à la stratégie *communication par nécessité*. HOMA traduit tous les paramètres des invocations en références globales. Une référence est composée par son nom et son numéro de version. Si un serveur produisant des paramètres (*out* et *inout*) connaît à l'avance tous les serveurs qui les consommeront, son *squelette* peut envoyer immédiatement les données aux consommateurs après sa production. Le récepteur les stockera dans un cache jusqu'à leur consommation. Ce mode de communication est appelé méthode *put*. Si un *squelette* demande une donnée qui n'a pas encore été reçue, il fait une demande au processus qui la produit : il s'agit de la méthode de communication *get*, présentée en figure 5.3.

Dans ces deux méthodes de communication le client qui produit la séquence des invocations ne reçoit jamais les données, sauf s'il demande à lire les données produite par une invocation (figure 5.2, cas b) à droite). La méthode *put* doit être utilisée si les données sont de petites tailles. Cependant, parce que cette méthode exige le stockage des copies des données sur les serveurs, elle doit être

soigneusement utilisée avec des données de grande taille pour éviter une saturation de la mémoire.

De plus, la technique d'« *invocation par continuation* » permet aux serveurs de communiquer directement entre eux, donc les communications peuvent être réalisées en parallèle entre des serveurs s'exécutant en parallèle.

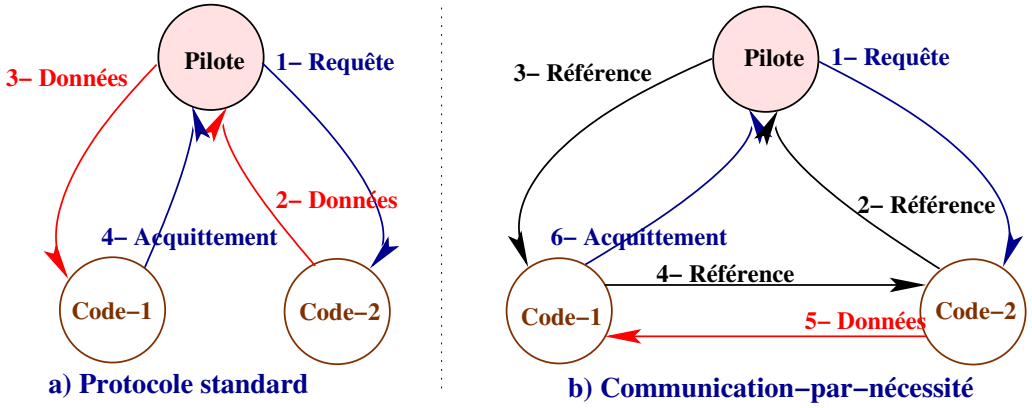


Figure 5.3 *Communication-par-nécessité (méthode « get ») versus communication RMI standard.*

5.3 Résultats théoriques d'ordonnancement

HOMA est basé sur le noyau exécutif KAAPI du langage ATHAPASCAN [43, 51] présenté dans le chapitre 2 dans la section 2.5.3, page 19. Rappelons les notations utilisées sur un graphe de flot de données : T_1 indique le **temps séquentiel** qui est défini à partir d'une exécution séquentielle du programme. T_∞ **le temps parallèle** est la profondeur arithmétique du graphe prenant en compte les poids (le coût de calcul) des nœuds des tâches. T_∞ est alors la limite la plus basse du temps minimal exigé pour une exécution non-préemptive sur un nombre non-borné de processeurs en négligeant le temps de communications. C_1 **le volume de communication** et C_∞ **le retard** sont évalués sur le graphe de la même manière que T_1 et T_∞, mais en tenant compte seulement des poids (des tailles) des nœuds de données. C_1 est la somme des poids de tous nœuds de données. En supposant que la mémoire partagée est limitée, C_1 est alors une limite supérieure sur le nombre total des accès effectués à la mémoire partagée. C_∞ est la longueur du chemin critique des communications.

5.3.1 Modèle de coût

Supposons que le coût de communication d'un message de taille L est hL. Grâce à l'algorithme de vol de travail détaillé dans [43], n'importe quel graphe de flot de données peut être exécuté sur p processeurs dans un temps $T_p = O(T_1/p + T_\infty + h(C_1/p + C_\infty)) + O(\sigma)$. Où σ est le surcoût pour calculer l'ordonnancement du programme. Le résultat est valide sur une machine parallèle homogène (une grappe), si et seulement si les tâches sont non-bloquantes, et si les communications sont directes entre les processeurs.

Proposition 1 *En utilisant* HOMA*, le temps pour exécuter un client CORBA sur une machine avec p processeurs est :* $T_p^{homa} = O\big(\frac{T_1}{p} + T_\infty + h(\frac{C_1}{p} + C_\infty)\big) + O(\sigma)$.

Cette proposition se base sur les résultats d'exécutions parallèles des programmes ATHAPASCAN publié dans [43]. Le résultat est valide sur une architecture parallèle homogène. Le modèle de coût d'exécution d'HOMA est hérité de celui du moteur exécutif KAAPI d'ATHAPASCAN [43] : HOMA traduit une invocation bloquante à distance en invocations non-bloquantes (section 5.2.2) et il rend capable la communication directe entre les serveurs (section 5.2.3). Une telle technique ajoute un surcoût dans la création d'une tâche supplémentaire pour la continuation et une communication supplémentaire dans le cas du mode *get*. Mais asymptotiquement cela n'augmente ni le travail du programme ni le volume de communication.

Ainsi, $T_p = T_p^{homa}$ reste valide pour les applications distribuées CORBA où les codes du client sont compilés et exécutés avec l'environnement HOMA. Les caractéristiques mentionnées ci-dessus du compilateur IDL et du support d'exécution d'HOMA permettent d'exécuter n'importe quel programme CORBA comme un programme ATHAPASCAN/ KAAPI.

Il est bon de noter que si l'environnement standard CORBA est utilisé, et à cause des limitations expliquées dans l'introduction, l'exécution est séquentielle et le transfert des données est fait via le client. A cause des communications non-directes, le volume de données transférées entre les serveurs est doublé mais reste en $O(C_1)$. Donc, la complexité d'une exécution quelconque est : $T_p^{corba} = O\big(T_1 + hC_1\big) + O(\sigma)$. Si le programme est massivement parallèle ($T_\infty << T_1$) et qu'il implique des données de tailles importantes ($C_\infty << C_1$), alors utilisation d'HOMA par rapport CORBA entraine un gain linéaire par rapport au nombre de processeurs $\big(\frac{T_p^{corba}}{T_p^{homa}} = O(p)\big)$.

5.3.2 Utilisation de l'espace mémoire

Généralement, il n'est pas possible d'obtenir une accélération linéaire sur le temps d'exécution tout en limitant l'espace mémoire nécessaire à cette exécution. Cependant, ATHAPASCAN [43] est basé sur un algorithme d'ordonnancement qui limite l'espace mémoire pour exécuter un programme sur une machine p-processeurs. HOMA hérite de ces résultats présentés dans la section 2.5.3 du chapitre 2. Rappelons les notations décrites la section 2.5.3 : S_1 est l'espace mémoire nécessaire pour une exécution séquentielle sur 1 processeur ; S_p est l'espace mémoire nécessaire pour une exécution sur p processeurs. Nous avons donc la proposition suivante :

Proposition 2 *Pour chaque programme HOMA qui a besoin d'espace mémoire S_1 pour l'exécution séquentielle, il existe un ordonnancement avec le temps parallèle $O(T_1/p + T_\infty \log p)$ et l'espace mémoire $S_1 + O(pT_\infty + \log(pT_\infty))$.*

5.3.3 L'exploitation efficace des données parallèles

Il existe plusieurs tentatives pour intégrer les données des applications parallèles dans le modèle d'objet CORBA [69, 72, 57]. Le concept d'objet parallèle CORBA (PACO) [77, 72] fourni une technique efficace pour encapsuler les codes parallèles basés sur les bibliothèques d'échange de messages (*i.e. MPI*). Il définit les objets de données parallèles comme une collection d'objets CORBA identiques. La spécification de « donnée parallèle » CORBA [69] a été approuvée par OMG. Elle définit une approche supplémentaire pour l'implantation et l'utilisation d'objets CORBA capable de profiter de ressources parallèles du calcul. Les ORBs parallèles [69] exploitent ces avantages seulement dans le cas d'une invocation client-à-serveur.

Il est intéressant de noter que même si des composants utilisant des données parallèles CORBA [69, 72] sont exécutés dans une application, la sémantique de communication implique que le transfert de paramètres s'effectue par client. Ainsi, même si le travail est parallélisé $(T_1/p + T_\infty)$, en raison du goulot d'étranglement engendré, les coûts des communications reste de l'ordre de $h(C_1 + C_\infty)$. Les serveurs ne communiquent pas **directement** leurs données en présence d'un client qui décrit le couplage. Grâce à l'approche « communication par nécessité », HOMA permet aux composants parallèles de communiquer en parallèle (section 5.2.3). Le *squelette* étendu contient des méthodes supplémentaires qui sont les extensions des méthodes des objets définies par l'utilisateur (décrits dans le chapitre suivant). HOMA considère le même partitionement de données et distribution de requêtes des objets parallèles pour son *proxy*. En utilisant des ORBs Parallèles, les *proxies* créés par HOMA pourraient directement communiquer entre eux, ainsi le coût des communications reste de l'ordre de $h(C_1/p + C_\infty)$.

5.4 Conclusion

Ce chapitre a présenté notre approche pour une composition efficace et qui passe à l'échelle pour les applications distribuées du point vu de la composition des invocations de services en utilisant la technologie CORBA. Pour atteindre cet objectif, nous avons présenté comment, en compilant différemment les stubs et squelettes CORBA, nous pouvons construire le graphe de flot de données à l'exécution d'une séquence d'invocations de méthode CORBA. Ensuite, en utilisant un support exécutif adapté (ici le support exécutif KAAPI du langage ATHAPASCAN), nous montrons comment l'exploiter ses résultats de complexité en ce qui concerne l'exploitation du parallélisme. Ceci a été rendu possible grâce aux transformations « invocation par continuation » et « communication par nécessité ». Le résultat de cette compilation permet de créer des flots parallèles d'exécution, ce qui permet à plusieurs serveurs concurrent de communiquer en parallèle. De plus, nous montrons que le modèle de coût de ATHAPASCAN/ KAAPI est hérité dans HOMA et nous présentons le gain théorique vis-à-vis de l'utilisation d'une implémentation standard de CORBA.

A notre connaissance, le travail de HOMA est la première tentative pour considérer l'efficacité et la scalabilité de la composition des invocations de méthodes CORBA pour les applications à hautes performances dans la simulation numérique. Bien que basé sur CORBA et sa sémantique d'invocation de méthodes, les résultats de HOMA peuvent s'étendre au cas des autres systèmes utilisant RPC avec des sémantiques proches.

6

Implantation et évaluation d'HOMA

Sommaire

*Le chapitre précédent a présenté les principes et les résultats théoriques d'*HOMA *sur le temps d'exécution et la consommation mémoire d'applications décrites par assemblage de composants. Dans ce chapitre, nous décrivons l'implantation d'*HOMA *au-dessus de CORBA et* KAAPI. **HOMA *contient un compilateur IDL qui réalise automatiquement les transformations nécessaires et fournit un support d'exécution basé sur* KAAPI. *Nous évaluons également* HOMA *en fonction du surcoût ajouté à une invocation normale de méthode CORBA et aux résultats théoriques du chapitre précédent. Dans la dernière partie de ce mémoire, l'évaluation d'*HOMA *sur l'application SIMBIO de simulation en chimie est présentée.*

6.1 Implantation d'HOMA sur CORBA et ATHAPASCAN

HOMA est implanté sur la technologie à objet CORBA en utilisant le support exécutif KAAPI du langage ATHAPASCAN pour la gestion des graphes de flot de données. L'architecture d'HOMA (section 6.1.1) montre les interactions des différents modules et la manière dont il réutilise les codes existants. A l'aide de son compilateur IDL, HOMA automatise toutes les modifications requises sur les codes couplés (serveurs) et sur le pilote du couplage (client). L'exploitation du parallélisme sur un ensemble d'invocations de méthode et l'optimisation des communications des données sont réalisées en générant automatiquement les versions étendues des *stubs* client et des *squelettes* serveurs. Les sections 6.1.3 et 6.1.4 décrivent en détail ces transformations.

6.1.1 Architecture

Le compilateur IDL HOMA produit le *stub* client et les *squelettes* serveurs afin de permettre l'interprétation abstraite, l'exécution efficace non-préemptive et le mouvement automatique des données décrits dans la section 5.2 du chapitre précédent. Les codes sources des clients et des serveurs ne changent pas mais ont besoin d'être recompilés. Néanmoins, cette limitation peut être facilement levée pour les serveurs en utilisant un proxy du serveur qui serait généré par HOMA. En revanche, du coté des clients cela est autrement plus difficile dans le cadre d'une implantation de CORBA et des applications en C++. La figure 6.1 montre les interactions entre les *stubs* et les *squelettes* étendus HOMA et les *stubs* et les *squelettes* natifs CORBA. Le *stub* HOMA intercepte les invocations de méthodes pour produire les informations nécessaires pour construire le graphe de flot de données et gérer le mouvement automatique des données. Ces informations permettent de transformer les paramètres en références globales et de gérer les données dans un *cache*. Ce *cache* est distribué sur tous les processus de l'application et permet de stocker les données comme valeur de type `any` CORBA. Le graphe est géré et déplié à l'exécution en utilisant le noyau exécutif KAAPI du langage [43]. Le processus entier de compilation est détaillé dans sur la figure 6.1.

Nous avons étudié deux approches pour réutiliser les codes encapsulés dans des objets CORBA : soit par re-compilation des codes des serveurs, soit par la production d'un objet *proxy*. Dans la première approche, HOMA insère un *squelette* étendu qui intercepte les requêtes reçues (figure 6.1, la partie serveur à gauche). HOMA peut aussi produire un objet *proxy* qui intercepte les invocations avant les rediriger vers l'objet serveur (figure 6.1, le serveur droite).

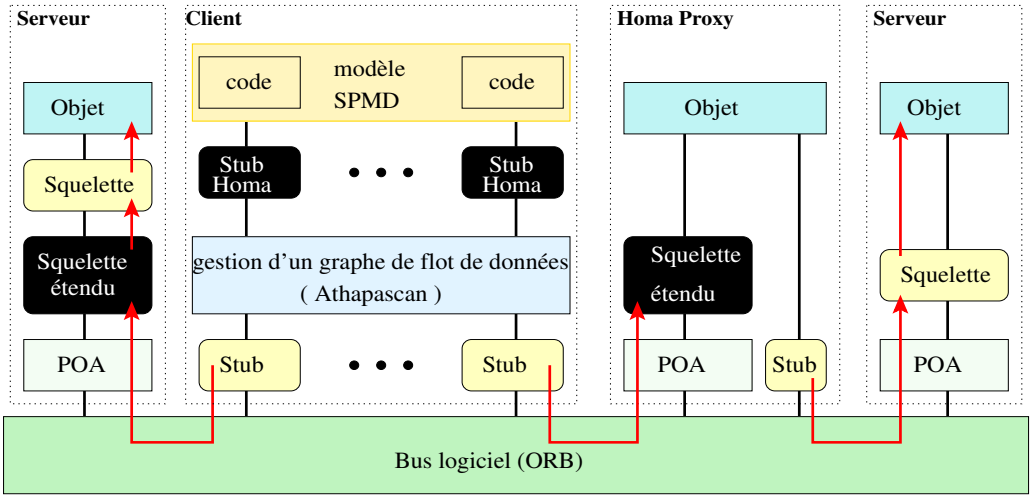


Figure 6.1 *L'architecture de HOMA.*

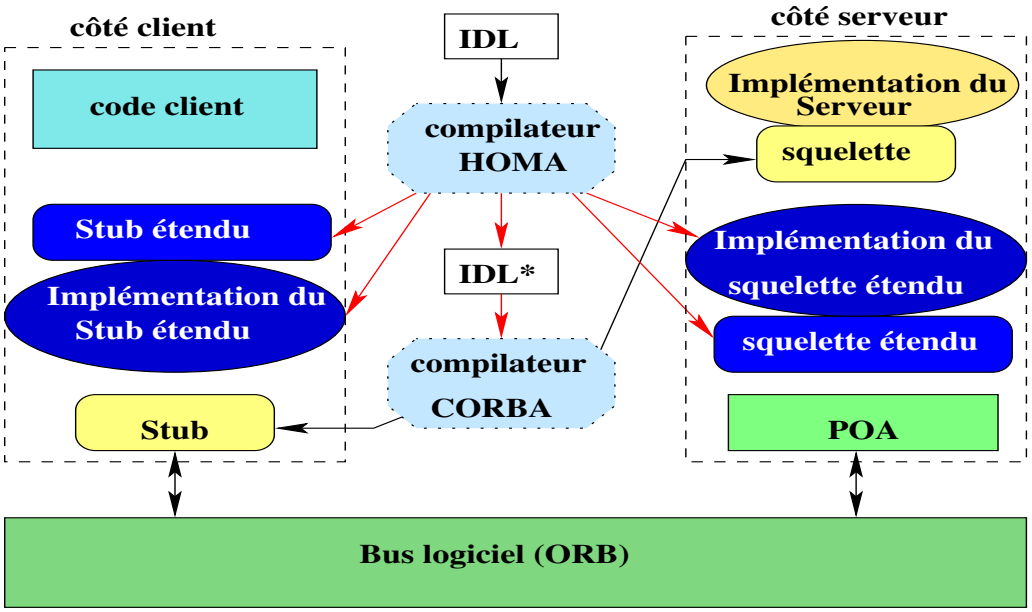


Figure 6.2 *La chaîne de compilation de l'IDL aux stubs et squelettes étendus.*

6.1.2 Processus de compilation

Afin d'obtenir une exécution parallèle des invocations de méthodes et d'effectuer la communication par nécessité des paramètres, nous proposons la chaîne de compilation (présentée par figure 6.2). Cette chaîne nous permet d'obtenir une application performante en recompilant le code pilote (le client) et les applications couplées (les serveurs).

Dans la première étape, le compilateur HOMA produit la version étendue des interfaces, notée IDL*, en générant la version asynchrone (*oneway*) de chaque méthode synchrone. Les paramètres retournés, (*out* et *inout*) sont convertis en un seul paramètre *in* qui représente les données pour la

continuation de l'invocation, comme présenté sur figure 6.4.

```
1. struct type {
2.   long a;
3.   sequence<double> b;
4. };
5. interface I {
6.   long foo(in long a,
7.      out long b, inout type c);
8. }
```

Figure 6.3 *Un exemple d'une définition IDL.*

De plus, le type des paramètres formels est traduit en références anonymes (`Key` dans la figure 6.4) pour permettre l'utilisation de la « communication par nécessité » vue dans chapitre précédent. Ainsi IDL^* est compilé en utilisant le compilateur standard IDL-vers-C++ fourni dans la plupart des implémentations de CORBA. Il produit un *stub* client et un *squelette* serveur pour l'interface étendue. Le compilateur HOMA produit aussi l'implémentation, du coté serveur, des méthodes asynchrones supplémentaires de IDL^* (`a_foo` sur la figure 6.4). Le compilateur HOMA produit également un *stub* parallèle ATHAPASCAN pour client prenant en compte les méthodes asynchrones du *stub* étendu produit à partir de l'IDL^*. L'implémentation du serveur (qui est à la charge du programmeur) reste inchangé : la méthode originale `foo` doit toujours être implantée et le client pense toujours qu'il l'invoque directement.

6.1.3 Projection des objets CORBA en objets ATHAPASCAN

Dans cette section, nous détaillons la manière par laquelle la partie cliente d'HOMA est réalisée. Nous commençons par l'implantation du *stub* de l'interface étendu qui transforme une invocation de la méthode originale CORBA en création de tâche ATHAPASCAN. L'idée de base est de transformer les paramètres d'une invocation en données partagées. Ensuite, nous décrivons le schéma du corps d'une tâche ATHAPASCAN qui prend en charge l'invocation à distance des méthodes de l'interface étendu et le transfert des paramètres effectifs.

Stub parallèle HOMA. Le but du *stub* client est de construire la requête correspondant à l'invocation d'une méthode via l'ORB. Avec l'utilisation ATHAPASCAN, le client qui invoque une mé-

```
1. #include <homa_rt.idl> // HOMA definition of Key and Continuation
2. interface I {
3.   long foo(in long a, out long b, inout type c);
4.   oneway void a_foo(in Key a, in Key c, in Continuation cc);
5. }
```

Figure 6.4 *La définition IDL* générée de la figure 6.3.*

thode doit créer une nouvelle tâche. L'exécution d'une tâche correspond à l'exécution d'une méthode native CORBA sur un objet CORBA. Etant donné qu'une invocation CORBA est, par défaut, une invocation bloquante, le compilateur HOMA produit, pour chaque méthode bloquante, une méthode non-bloquante définie dans IDL^* via l'attribut *oneway*. De plus, tous les paramètres $inout$ et out sont convertis en paramètres in et en ajoutant un paramètre qui permet de préciser l'information où serveur doit stocker la valeur produite.

De l'IDL de la figure 6.3, le compilateur HOMA produit le *stub* client parallèle de la figure 6.5. Nous supposons que toutes les définitions standards IDL-vers-C++ de la structure de données et des interfaces sont dans l'espace de nom `Orig::` (nativement ces définitions sont dans l'espace de nom CORBA ou les espaces de nom des modules IDL).

```
1. namespace homa{
2.   typedef Shared<Key> Long;
3.   typedef Shared<Key> type;
4.   ...// same typedef's for the data types
5.   class I { // Definition interface I
6.     Shared<Orig : :I_ptr> _self; //Athapascan's reference to CORBA object
7.   public :
8.      //Definition of foo using typedef's in namespace homa : :
9.     Long foo(Long& a, Long& b, type& c){
10.      Long retval;
11.      Continuation cc( b, c, retval );
12.      Fork<INVOKE_foo>()(_self, a, c, cc );
13.      Fork<CONT_foo>()( cc, b, c, retval );
14.      return retval;
15.   } /*foo*/ }; /*class I*/ } /*namespace homa*/
```

Figure 6.5 *Le stub client pour l'IDL de la figure 6.3. Les définitions `I_ptr` et `I_var` ne sont pas présentées, elles sont équivalentes à des pointeurs sur un objet de la classe `homa : :I`. La tâche `INVOKE_foo` est définie dans la figure 6.6.*

Un client qui invoque la méthode `foo` sur une variable de type `homa : :I_var` crée implicitement une tâche. Les opérateurs de conversion existent entre le type `Shared<Key>` ATHAPASCAN, les définitions anonymes (tel que `homa::Long`) et les définitions standards C++ (`Orig::Long`). Les opérateurs de sérialisation nécessaire à l'utilisation des « variables partagées » d'ATHAPASCAN sont omis. Sur la figure 6.5, la ligne 11 déclare une variable de continuation définie comme une variable partagée ATHAPASCAN. Cette variable partagée est déclarée comme non-prête. En utilisant ce mécanisme, la tâche `CONT_foo` ne deviendra prête que lorsque l'exécution à distance sera finie et qu'un signal sera reçu.

Via l'utilisation de ce *stub*, le client exécute ainsi un programme ATHAPASCAN. La section suivante présente comment créer les tâches à partir de la définition IDL.

Génération des tâches ATHAPASCAN. Afin d'utiliser ATHAPASCAN comme support d'exécution détectant automatiquement les dépendances entre les invocations, les définitions IDL sont tra-

duites en définitions ATHAPASCAN [1]. Chaque invocation de méthode sur un programme client crée une tâche. Chaque argument de méthode et chaque objet d'interface est traduit en un objet partagé. Le paramètre formel IDL est traduit en un `Shared` ATHAPASCAN : `in`, `inout` en `Shared_r` et les paramètres de sortie (*out*, *inout* et la valeur de retour) sont indirectement passés via l'objet de continuation déclaré en ligne 11 sur la figure 6.5. La référence de l'objet CORBA (`_self`) est explicitement passée à la fonction (ligne 12 sur la figure 6.5, la line 4 sur la figure 6.6). Dans ATHAPASCAN une tâche est une fonction. Ainsi une invocation de méthode est traduit en appel de fonction avec un paramètre supplémentaire qui représente l'objet.

Deux méthodes ne peuvent pas être exécutées simultanément sur un même objet sans pour autant faire certaines suppositions sur la définition de la méthode (par exemple la méthode ne change pas l'état interne de l'objet). Ainsi, leur exécution sont mises en série et la tâche `INVOKE_foo` requiert que la référence à l'objet CORBA utilise le mode d'accès exclusif (`Shared_r_w`) : deux tâches sur un même objet CORBA sont exécutées suivant l'ordre de leurs créations, *i.e.* dans le même ordre que si le client utilise le *stub* standard CORBA.

```
1. namespace homa{
2. class I {
3. struct INVOKE_foo {
4.   void operator()(Shared_r_w<Orig : :I_ptr> obj, // CORBA object
5.               Shared_r<Orig : :Long> a, Shared_r<Orig : :type> c,
6.               Continuation cc )
7.   {
8.      obj.access()->a_foo( a.read(), c.read(), cc.reference() );
9.   }/*operator()*/   };/*struct*/   };/*class*/   }/*end of namespace homa : :*/
```

Figure 6.6 *L'implantation de tâche `INVOKE_foo` associée à la méthode `foo`.*

A la ligne 8 dans la figure 6.6, la méthode `a_foo` est non bloquante (*oneway*) et est invoquée avec la référence à l'objet continuation. Sur le serveur, la méthode `a_foo` exécute la méthode `foo` (ligne 12 de la figure 6.7). A la fin de l'invocation, il invoque alors sur le client une méthode asynchrone (`Put_Data`) pour retourner les valeurs (`Key` anonyme) qui signale à la tâche de continuation que `a_foo` a terminé son exécution.

L'objet de retour. Au retour des résultats, il est nécessaire de mettre à jour les données et d'activer la tâche de continuation. Ces opérations sont réalisées à travers l'invocation d'une méthode non-bloquante, appelée `Put_Data`, sur un objet CORBA, appelé `Call_Back`, située sur le processus client. Le *squelette* étendu HOMA encapsule l'ensemble des résultats sous la forme d'un paramètre et invoque la méthode `Put_Data` à la fin de son exécution (figure 6.7).

6.1.4 Localisation du schéma de communication des données

Le but de cette section est de montrer comment HOMA gère la communication des données entre les serveurs couplés. Le corps des méthodes de l'interface étendu décrit ce schéma de communication :

[1] En pratique il s'agit des définitions du moteur exécutif KAAPI d'ATHAPASCAN. Mais le code KAAPI, étant de plus bas niveau et donc plus long à écrire, est omis et nous utilisons à la place une syntaxe équivalente en ATHAPASCAN.

tout d'abord la méthode cherche les paramètres en entrée, puis appelle la méthode originale et enfin stocke les paramètres en sortie dans le cache local du processus. L'idée est assez générale et peut supporter des stratégies quelconques de remplacement de données dans les caches. Cependant, HOMA a choisi la stratégie OCR (Owner Compute Rule) : les données sont stockées sur le processus de calcul qui les a produites.

```
1 . // a, b et c les paramètres de la méthode originale (ici "foo")
2 . cache = Get_Cache() ;// récupère la référence de l'objet cache en entrée
3 . // récupère les paramètres en mode "in" et "inout" (ici a, c)
4 . if (/* si la valeur de paramètre en entrée (ici a) est passé en mode indirect */ )
5 .    // si cette valeur est déjà arrivée, on récupère la référence locale
6 .    // sinon on la récupère sur le cache en sortie du producteur via un appel à distance
7 .    cache->Find_CacheCopy(keyref_TMP) »= a_DATA;
8 . else
9 .    a.data()»=a_DATA;
10. // même procédé pour les autres entrées (ici c)
11. // appel local à la méthode originale "foo"
12. foo(a_DATA, b_DATA, c_DATA);
13. // prépare les valeurs de retour (ici b, c)
14. If (// si la taille de donnée produit (ici b) est supérieure à un seuil)
15.    // on la stocke dans le cache en sortie
16.    // et met à jour le mode de transfert en indirect dans sa clé
17. else
18.    // sinon on met à jour son mode de transfert en direct
19. // insertion de la donnée (ici b) dans la liste des valeurs de retour
20. // même procédé pour les autres valeurs de retour (ici c)
21. // invocation de la méthode "Put_Data" sur l'objet de retour "Call_Back" vers le client
```

Figure 6.7 *Implantation de la méthode* `a_foo` *qui est la version étendue de* `foo` *de la figure 6.4. Les trois paramètres de la méthode originale sont a (en « in »), b (en « out ») et c (en « inout »).*

Squelette étendu HOMA. Les corps de méthodes étendues sont générés automatiquement par le compilateur IDL HOMA. Sa fonction est de transférer les invocations vers les méthodes originales implantées par le concepteur des objets serveurs. Avant d'appeler la méthode originale associée, il faut récupérer les données des paramètres en entrée ; si les données ne sont pas encore arrivées, une invocation à distance les récupère (dans la stratégie « get », c'est le premier accès ; dans la stratégie « put », le producteur qui ne l'a pas encore envoyé). Après l'appel de la méthode originale, les données de retour sont transférées à l'initiateur soit sous la forme de données immédiates (si elles sont petites) soit sous la forme d'une clé d'accès à distance. La figure 6.7 montre cet algorithme sur l'exemple précédent.

6.1.4.1 Architecture des caches

Les données dans le cache sont stockées en utilisant le type `Any` CORBA. Un cache est un objet CORBA. Le compilateur HOMA produit le code nécessaire pour créer un objet cache sur un processus (Unix). Une référence à une donnée est la référence à l'objet CORBA du cache plus la référence à son entrée dans le cache. De plus, un seuil permet de communiquer les petites données dans une référence. Nous avons appelé `Key` l'union d'une référence et de sa valeur (cf. figure 6.4).

Deux caches sont gérés par HOMA sur chacun des processus : le premier correspond aux données produites par chacun des serveurs possédant une méthode avec une direction *inout* ou *out* et il est appelé *cache des valeurs en sortie* ; le second correspond aux données qui ont besoin d'être lues pour réaliser une invocation de méthode sur un serveur et il est appelé *cache des valeurs en entrée*. Un cache des valeurs en entrée contient uniquement des copies ; une politique de type LRU gère le cache pour limiter l'utilisation mémoire. Chaque cache des valeurs en sortie garde une liste des références sur chacune des valeurs copiées. Les valeurs de production sont stockées dans un cache des valeurs en sortie jusqu'à ce qu'elles soient effacées. La gestion des données dans ce type de cache est gérée par le programme ATHAPASCAN client : un accès à une variable partagée dans ATHAPASCAN représente une *version* à la donnée partagée associée. La destruction des versions par ATHAPASCAN [43, 51] est automatique et appelle la destruction du type associé. Le destructeur nous sert à supprimer la donnée du cache et ses copies.

Les objets caches. Dans la stratégie de type « get », l'objet cache gérant les paramètres de sortie est un objet reparti car les autres objets serveurs ont besoin d'y accéder. Il en va de même pour le cache gérant les paramètres d'entrée pour la stratégie de type « put ». L'interface associé à ces objets exploite trois fonctionnalités : insertion, récupération et suppression d'un élément. La récupération d'un élément se fait par une clé. Cette clé est produite lors de l'accès au cache de sortie et est utilisée pour accéder au cache d'entrée.

Génération des références. Une référence dans le cache est codée par une référence à l'objet CORBA « cache », *i.e.* l'IOR de l'objet, et par sa clef d'accès dans le cache qui est codée avec un entier. Toutes les données plus petites qu'un certain seuil (déclaré dans l'implantation) sont directement communiquées et ne sont pas stockées dans les caches. Les données stockées dans les caches ou les données immédiates sont du type `Any` CORBA. L'*union* d'une référence et d'une donnée immédiate est définie dans le fichier d'entête IDL de HOMA (`homa_rt.idl`).

6.2 Evaluation d'HOMA par des expériences élémentaires

Tous ces résultats ont été obtenus en utilisant l'implantation CORBA OmniORB3. Les programmes ont été exécutés sur le iCluster1 de l'INRIA à Grenoble (PC 733Mhz, 256Mo, réseau 100Mbit/s). Les résultats sont donnés en utilisant la version actuelle de l'environnement HOMA, qui implante seulement le mode *get* présenté ci-dessus. L'algorithme d'ordonnancement est un algorithme centralisé, basé sur une liste qui exécute la première tâche prête de la liste.

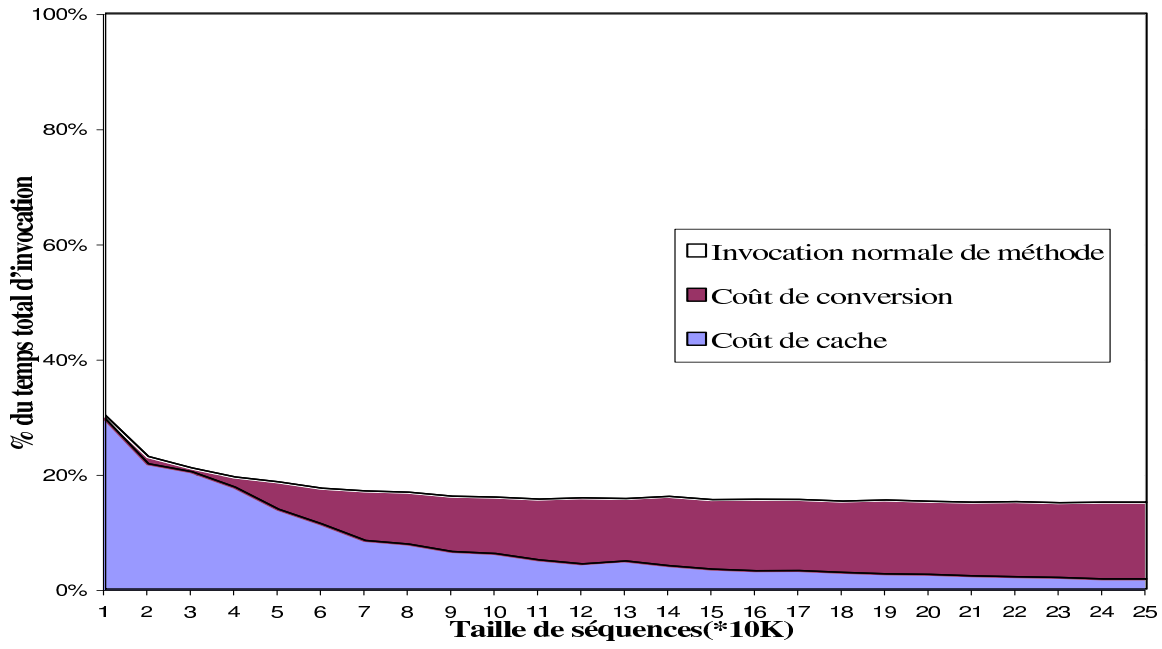


Figure 6.8 *La décomposition du surcoût d'invocation d'une méthode par* HOMA.

6.2.1 Surcoût d'invocation d'une méthode par HOMA

La première expérience, figure 6.8, montre la décomposition des coûts d'une invocation de méthode avec l'utilisation du cache des données en fonction de la taille des résultats d'invocation (une séquence de double). Outre la gestion du cache (zone en gris clair) et le coût de l'invocation (zone en blanc), on s'aperçoit que les coûts de conversion d'une séquence de double vers le type *Any* (qui inclue une copie mémoire) est non négligeable (zone en noir). Les autres coûts (conversion *Any* vers la séquence de double, accès aux fonctions d'insertion / extraction du cache) sont négligeables et ne sont pas représentés ici.

6.2.2 Bande passante cumulée

Dans cette expérience, nous exécutons le programme de la figure 6.9. N est le nombre de paires (un couple de deux serveurs ; consommateur et producteur). Le paramètre des invocations est une séquence de taille K et chacun des $2N$ objets est placé sur un processeur particulier (p=2N).

L'invocation `produce` sur `Ps[i]` écrit une séquence de taille K alors que l'invocation `consume` sur `Cs[i]` liste tous les éléments de la séquence d'entrée. Ce programme est caractérisé par $T_1 = O(N)$ et $T_\infty = O(1)$; $C_1 = O(NK)$ et $C_\infty = O(1)$. La figure 6.10 rapporte le temps d'exécution en utilisant CORBA et HOMA avec deux tailles de séquence ($K = 3.2MBytes$ et $K = 9.6MBytes$). Le temps pour CORBA croit de manière linéaire avec le nombre de processeurs et avec K comme prédit par $T_p^{corba} = O(NK)$. Avec HOMA, nous observons que le temps est presque constant : les résultats ajustent notre prédiction de complexité $T_p^{homa} = O(K)$. Néanmoins, une analyse plus fine montre que ce temps est linéaire en N : ceci est dû au terme $O(\sigma)$ dans notre modèle de coût (pro-

```
1. // Call the producters
2. for(int i=0; i < N; i++)
3.    // x[i] is output parameter
4.    Ps[i] -> produce ( x[i] );
5. // Call the consumers
6. for(int i=0; i < N; i++)
7.    // x[i] is input parameter
8.    Pc[i] -> consume ( x[i] );
```

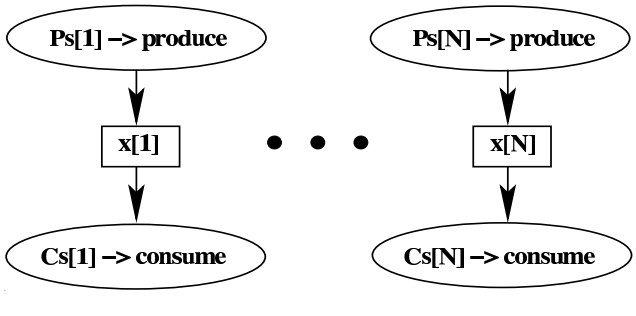


Figure 6.9 *A gauche : le code client. A droite : le graphe de flot associé.*

position 1 page 50). Ce terme tient compte du temps de construction du graphe de flot de données qui correspond majoritairement au surcoût pour dérouler les boucles afin de créer $O(N)$ tâches avec les dépendances. Mais le temps d'une invocation étant de plusieurs ordres de grandeur supérieur au temps de construction d'un noeud dans le graphe de flot de données, ce surcoût est négligeable.

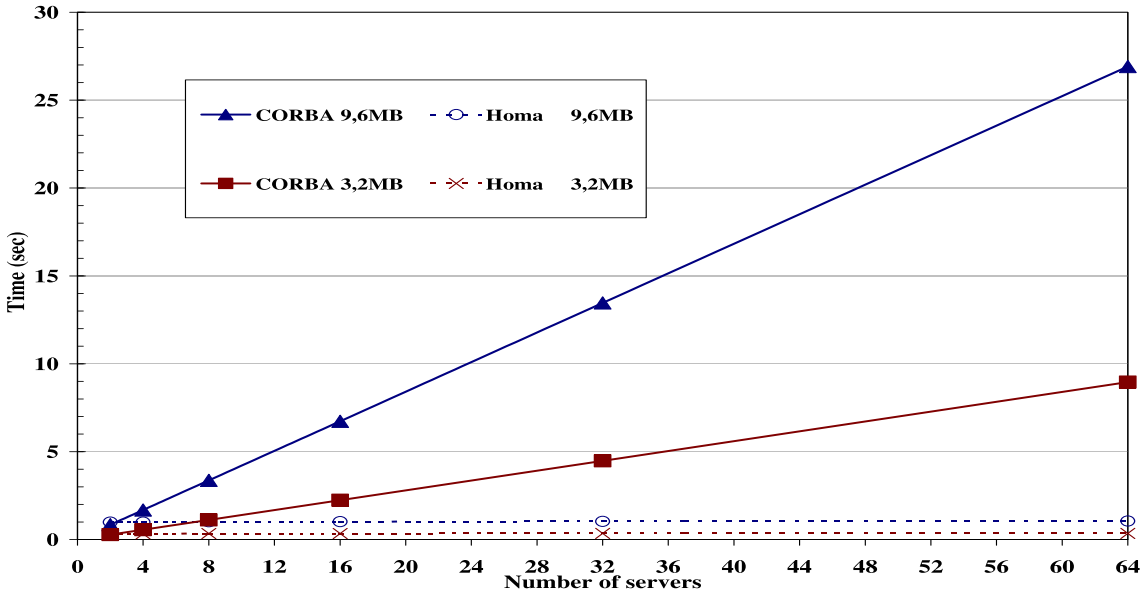


Figure 6.10 *Le temps d'exécution du code de la figure 6.9.*

En utilisant HOMA, la bande passante cumulée grandit linéairement avec le nombre de paires, la bande passante cumulée maximum est de $304MBytes/s \simeq 3Gbits/s$ pour 32 serveurs communiquant avec 32 serveurs (64 serveurs). La bande passante pair-à-pair moyenne entre deux serveurs est environ $9.5MBytes/s$[2]. Dans une autre expérience, le passage à l'échelle d'HOMA pour 120 serveurs a été observé avec une petite dégradation sur la bande passante moyenne pair-à-pair mais celle-ci reste

[2]La bande passante pair-à-pair maximum par rapport à MPI sur TCP est d'environ $11MBytes/s$ sur le iCluster1.

linéaire en fonction du nombre de processus. Donc HOMA, permet exploiter le parallélisme sur une séquence d'invocations de méthode et permet de gérer les communications parallèles.

Sur des expériences similaires où des objets parallèles peuvent être utilisés sur un même réseau rapide ethernet, les résultats publiés pour PACO [77, 72] relatent que la bande passante pair-à-pair varie de $9.1MBytes/s$ à $11.3MBytes/s$. Cette bande passante dépend de la version de l'implantation, de l'architecture et du nombre de serveurs. Néanmoins, avec PaCO/PaCO++, un programme itératif, comme exprimé par les boucles de la figure 6.9, doit être écrit différemment au risque que le client soit impliqué dans chacune des communications comme expliqué dans le chapitre 5 section 5.3.3.

La différence entre la bande passante pair-à-pair de MPI et de TCP provient de l'architecture du cache générique utilisant le type Any CORBA implique quelques copies supplémentaires en mémoire ; la stratégie « get » implique une communication supplémentaire pour obtenir les données par rapport à la stratégie « put ». Tous ces facteurs limitent l'obtention de la bande passante maximale disponible. Néanmoins les résultats sont satisfaisants.

6.3 Conclusion

Dans ce chapitre, nous avons décrit l'implantation d'HOMA au-dessus de CORBA en utilisant le support exécutif KAAPI du langage ATHAPASCAN pour la gestion des graphes de flot de données. L'architecture d'HOMA montre comment HOMA se positionne dans la chaîne de compilation d'une application CORBA et comment il réutilise les codes existants. A l'aide de son compilateur IDL, HOMA automatise toutes les modifications requises sur les codes couplés (des serveurs) et sur le code qui pilote le couplage (un client). L'exploitation du parallélisme sur un ensemble d'invocations de méthode et l'optimisation des communications de données sont réalisées en générant automatiquement les versions étendues des *stubs* client et des *squelettes* serveurs.

Nous avons évalué également HOMA en fonction du surcoût ajouté à une invocation normale de méthode CORBA et aux résultats théoriques du chapitre précédent. Nous avons observé que HOMA peut agréger la bande passante pair-à-pair et nous avons vérifié notre prédiction de complexité : le temps d'exécution avec HOMA de l'invocations indépendantes sur p processeurs est quasiment constant comme prédit.

HOMA est dédié aux applications construite par assemblage de serveurs et où la description de la coordination entre serveurs est décrite par un code (le client). La partie suivante vise à permettre de décrire cette coordination de manière concurrente et distribuée.

III

Modèle pour le couplage de codes

7

Collection temporelle d'objets partagés

Sommaire

Une collection temporelle est un ensemble d'accès à des objets partagés associé à une notion d'ordre lié à un temps d'exécution. L'exemple typique est de considérer l'ensemble des données qui transitent sur un canal de communication reliant deux applications. L'objectif de ce chapitre et de définir cette notion de collection temporelle. Ensuite nous proposons un interface de niveau applicatif qui étend l'interface ATHAPASCAN. *Outre l'intérêt d'une telle sémantique pour le développement d'applications distribuées, les définitions que nous donnons permettent de découpler la déclaration d'un accès à un objet partagé de l'accès effectif à ses données en mémoire, c'est-à-dire que nous permettons au moteur exécutif sous-jacent de recouvrir les latences d'accès par du calcul, si le matériel le permet effectivement.*

7.1 Introduction

Dans la partie précédente de ce mémoire, nous avons étudié la problématique du couplage de codes parallèles comme une composition de services sur différents objets répartis. La description du couplage est donnée par un algorithme qui pilote les invocations aux services. Bien que cet algorithme puisse être exécuté comme un programme parallèle (il s'agit, rappelons-le, d'un programme ATHAPASCAN), il suit toujours une sémantique de type séquentielle[1] pour, l'invocation des services : c'est-à-dire, qu'à tout point dans l'exécution de l'algorithme du pilote du couplage, la valeur d'une variable est donnée par la dernière invocation, suivant un ordre séquentiel, que cette variable soit produite ou modifiée.

Avec HOMA nous avons montré comment on peut réaliser un ordonnancement non-préemptif efficace dans ce cadre. Ce type de couplage de codes couvre beaucoup d'applications, cependant il existe d'autres applications de couplage qui ne peuvent pas être définies ainsi. Il s'agit en particulier des applications couplées par un flux de données, telles que celles développées avec des environnements interactifs comme FlowVR [5] ou Stampede [75, 76], dans lesquels le couplage est distribué sur plusieurs processus. Une sémantique de type séquentielle, comme celle d'ATHAPASCAN, n'est pas suffisante pour définir une telle exécution.

Dans ce chapitre, nous visons à définir des objets permettant de programmer des applications dont le couplage est décrit par un ensemble de processus communicants par envoi de messages sur un canal. La description d'une telle sémantique ne peut plus être de type séquentiel : la valeur retournée par une lecture d'un objet partagé dépend à la fois d'un écrivain « précédent » ainsi que de la sémantique du canal de communication (par exemple FIFO ou autre). Dans ce chapitre, nous visons à définir précisément quel est l'écrivain « précédent », indépendamment de l'entrelacement (ordonnancement) des instructions des différents processus concurrents à chaque bout du canal de communication.

D'autre part, les communications entre processus ne sont pas synchrones et le délai de transit d'une donnée dépend de paramètres difficilement contrôlables comme la charge de la liaison physique reliant les processus ou bien les protocoles de communication utilisés. Le second objectif de ce chapitre est d'offrir des opérateurs qui permettent de masquer ces latences de communication par recouvrement du calcul. De la même manière que pour les langages de type flot de données, et en

[1] En pratique il s'agit de la sémantique donnée par l'ordre de référence d'ATHAPASCAN qui est présentée dans le chapitre 2 section 2.5.3.

particulier d'ATHAPASCAN, la solution proposée est de différer la déclaration d'une lecture/écriture de l'opération effective à la donnée.

Le plan de ce chapitre est le suivant : Dans un premier temps nous présentons notre définition générique d'un objet représentant une donnée partagée, appelé par la suite *objet partagé*. Nous illustrons cette définition en comparaison avec la sémantique d'ATHAPASCAN. Ensuite nous étendons la sémantique d'ATHAPASCAN pour considérer une nouvelle abstraction appelée *collection temporelle* qui permet de définir la sémantique entre producteurs et consommateurs couplés par des canaux de communication. Ce chapitre se termine par une présentation de l'interface de programmation qui étend celle d'ATHAPASCAN et par un exemple de programme. Le chapitre suivant présentera une nouvelle abstraction qui permet de capturer les aspects distribués des données d'une application parallèle, celle-ci sera appelée *collection spatiale*.

7.2 Définition générique d'objet partagé

Dans le cadre de cette partie, nous considérons une application distribuée dont les processus communiquent à travers une mémoire partagée. Deux tâches de calcul qui veulent communiquer doivent partager une même donnée dans cette mémoire partagée. Nous appelons une telle donnée partagée un « objet partagé ». Une première définition suit :

Définition 3 ***Un objet partagé*** *est une structure de donnée qui possède un interface et qui peut être accéder par plusieurs tâches s'exécutant sur plusieurs processus.*

Dans le cadre du langage ATHAPASCAN, les objets partagés sont déclarés par le mot clé `Shared` et peuvent être passés en paramètre des tâches créées par le mot clé `Fork`. Ces tâches peuvent s'exécuter sur plusieurs processus en fonction des choix de l'algorithme d'ordonnancement utilisé.

Définition 4 ***Un accès à objet partagé*** *est une référence qui est associée à toute création de tâche qui accède à un objet partagé. Un accès est typé (lecture, écriture ou modification) et possède un mode qui spécifie la manière dont est accédé l'objet partagé (direct ou différé).*

Dans ATHAPASCAN, un accès correspond à la liaison entre un paramètre effectif et un paramètre formel d'une tâche au moment de sa création. Le type et le mode de l'accès est donné seulement par le type C++ de définition du paramètre formel : il s'agit des types définis dans la section 2.5.3, `Shared_r`, `Shared_w`, et `Shared_rw`, éventuellement avec les modes d'accès différés nommés *postponed*.

A l'exécution d'un programme ATHAPASCAN, la sémantique du langage spécifie quelle est la valeur retournée pour chaque accès en lecture à un objet partagé. Cette valeur est appelée *version* de l'objet partagé. Dans [42], chaque objet partagé est considéré comme une succession de versions, comme illustré dans la figure 7.1. Une version est produite par un accès en écriture : une tâche qui possède un accès en écriture à un objet partagé peut écrire une nouvelle valeur de la structure de donnée associée. Un accès en lecture retourne la version qui correspond à la valeur écrite par le dernier accès en écriture suivant l'ordre de référence d'ATHAPASCAN présenté dans la section 7.2.2.

Pour définir la sémantique de notre extension au langage ATHAPASCAN, nous allons revoir ces définitions en ne considérant que les opérations d'accès à un objet partagé. De ce point de vue, et à la différence de [42], chaque objet partagé est considéré comme une succession d'accès.

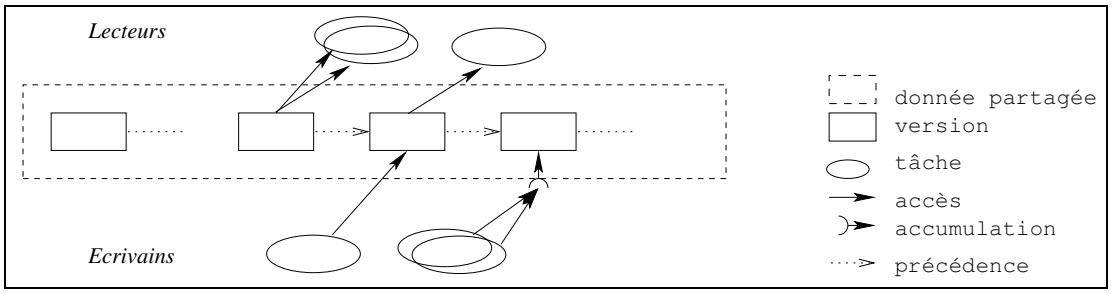


Figure 7.1 ***Succession des versions associées à un objet partagé.*** *Lorsqu'une tâche prend pour paramètre une référence sur une donnée en mémoire partagée, une référence sur une des versions associées à l'objet partagé lui sera retournée lors de l'exécution. En présence d'une fonction d'accumulation, l'écriture concurrente est tout aussi envisageable que la lecture concurrente.*

Définition 5 ***Un objet partagé*** *est un ensemble d'accès à une donnée dans la mémoire partagée. Chaque accès possède un type et un mode comme définis précédemment. Le premier accès est toujours une écriture, correspondant à l'écriture de la valeur initiale de la donnée.*

7.2.1 Consistance associée à un objet partagé

La consistance d'un modèle de mémoire partagée permet de définir quelle est la valeur retournée par une lecture en présence d'un ensemble d'écritures effectuées par des processus concurrents. Dans notre cas, la consistance associée à la mémoire partagée est définie par rapport au graphe de flot de données de l'exécution du programme. Cette approche est généralement appelée *programmer-centric* ou *computation-centric* dans [40, 39] et se différencie de l'approche classique, dite *processor-centric*, où la consistance de la mémoire est définie par rapport aux dates d'exécution des instructions par les processeurs. La consistance associée à notre mémoire partagée est plus précisément nommée *location-consistency* dans [40, 39] et est une généralisation de la consistance séquentielle définie par Lamport [59] au sens où la consistance de la mémoire est définie pour chaque donnée de la mémoire partagée et non pour la mémoire dans son ensemble.

La suite de cette section, nous donne une vision commune et bien précise sur les vocabulaires qui seront utilisés pour nos définitions étendues. La définition suivante rappelle ce qu'est un graphe de flot de données présenté dans la définition 2 page 16.

Définition 6 *Soit $G = (V, E)$, un graphe de flot de données correspondant à une exécution complète d'un programme* ATHAPASCAN*, et tel que les tâches (V_t) et les versions (V_v) forment l'ensemble $V = (V_t \cup V_v)$ des noeuds, et les accès des tâches sur les versions forment l'ensemble E des arêtes. Soit x un objet partagé de la mémoire partagée et $V_x \subset V_v$ l'ensemble des versions associées à cet objet x.*
*R_x est l'ensemble des accès en **lecture** de cet objet x.*
*W_x est l'ensemble des accès en **écriture** de cet objet x.*

A partir de cette définition, nous pouvons alors proposer une définition générale d'un modèle de consistance associé à un objet partagé :

Définition 7 *Un modèle de consistance associé à un objet partagé x est donné par la définition d'un ordre total sur les accès de l'objet x.*

Par la suite, nous allons définir cet ordre à partir de deux sous ordres total sur chacun des ensembles d'accès en lecture et écriture (R_x et W_x). Nous noterons par :

Définition 8 *Les ordres sur les accès à un objet partagé :*
Soit $\prec_r$ un ordre total sur R_x, et soit $ord_r : R_x \mapsto IN$ la fonction ordinale qui retourne un entier pour chaque accès de R_x compatible avec l'ordre $\prec_r$: c'est-à-dire que si $a \prec_r a'$ alors $ord_r(a) \prec_{IN} ord_r(a')$.
Soit $\prec_w$ un ordre total sur W_x, et soit $ord_w : W_x \mapsto IN$ la fonction ordinale sur W_x.

Nous donnons ci-dessous une définition générale de la sémantique associée à un objet partagé, qui sera définie précisément pour le langage ATHAPASCAN dans la section suivante. Puis, nous étendons cette définition pour préciser la sémantique associée au nouvel objet partagé que nous appelons collection temporelle.

Définition 9 *La sémantique associée à un objet partagé est donnée par une fonction appelée « **version apparentée produite** » $\Psi_x : R_x \mapsto W_x$ telle que pour tout accès $a_r \in R_x$, $\Psi_x(a_r) = a_w$ ($a_w \in W_x$).*

La sémantique associé à un objet partagé permet de définir quelle est l'accès en écriture qui produit la valeur retournée par un accès en lecture.

7.2.2 Définition de la sémantique d'ATHAPASCAN

L'objet partagé d'ATHAPASCAN est déclaré par « `Shared` ». Pour ce type d'objet partagé, la sémantique d'ATHAPASCAN est telle que pour toute exécution, la valeur retournée lors d'un accès en lecture est identique à la valeur retournée lors de l'exécution « séquentielle » de l'application. La sémantique d'ATHAPASCAN est aussi dite lexicographique puisqu'il est possible, en lisant le source du programme, de déterminer quelle est la tâche qui écrit une version qui correspond à une lecture.

Plus précisément, l'ordre d'exécution qui définit la sémantique d'un programme ATHAPASCAN est l'« ordre de référence » qui est très proche d'un ordre séquentiel d'exécution (comme celui défini par C++).

Définition 10 *[42] L'ordre ordre de référence d'*ATHAPASCAN*, $\prec_{Ath}$ est un ordre total sur les tâches et leurs instructions, défini de la manière suivante (t, t', t" et t_i représentent des tâches) :*
- *Si l'exécution de t crée $t_1, t_2..., t_n$ dans cet ordre, alors $t_1 \prec_{Ath} t_2 \prec_{Ath} ..., \prec_{Ath} t_n$.*
- *Si t crée t' , alors $t \prec_{Ath} t'$.*
- *Soient t_i et t_{i+1} deux tâches soeurs ($t_i \prec_{Ath} t_{i+1}$), alors pour toute tâche t' créée lors de l'exécution de la descendance de $t_i, t' \prec_{Ath} t_{i+1}$.*

La figure 7.2 permet de comparer l'*ordre de référence*, l'ordre séquentiel C++ et un ordre d'exécution qui respecte les dépendances de données et les créations des tâches. L'exemple implique quatre tâches indépendantes récursives.

Théorème 1 *Pour chaque objet partagé x d'un programme* ATHAPASCAN*, l'ordre de référence $\prec_{Ath}$ définit de manière unique un ordre total $\prec^x_{Ath}$ sur tous les accès de x.*

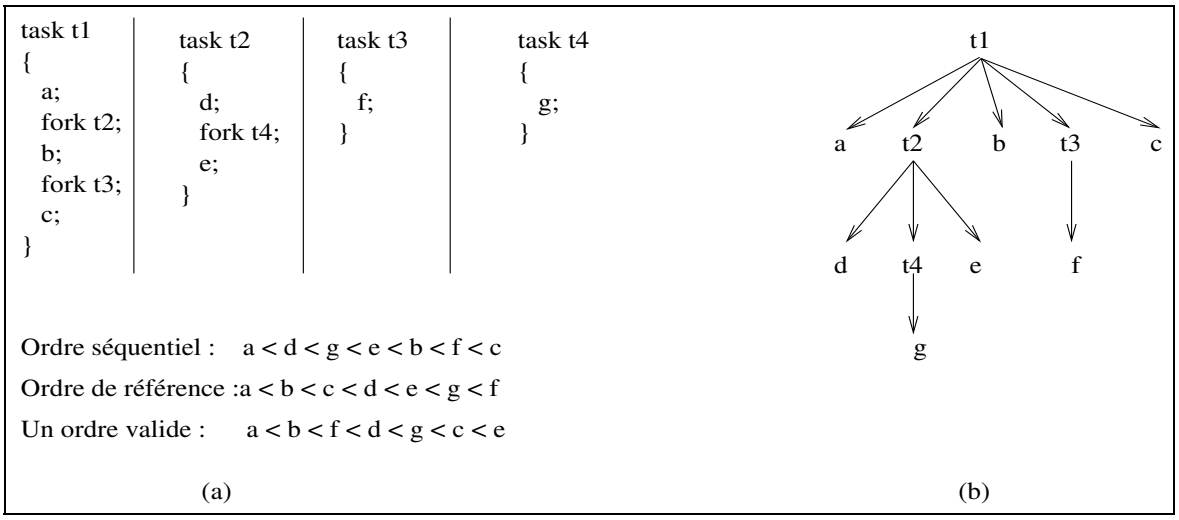


Figure 7.2 ***Ordre de référence [42].*** *En (a) est représenté le code de 4 tâches. La tâche t_1 crée les tâches t_2 et t_3, et la tâche t_2 crée la tâche t_4 . Les tâches t_2, t_3 sont indépendantes. De même que les tâches t_3 et t_4. Les instructions, autres que les créations de tâches, sont représentées par les symboles a à g. En (b) est représenté le graphe représentant le contexte d'exécution des différentes instructions. Enfin, en (a) de nouveau, sont représentés trois ordres d'exécution valides des instructions. Le premier est l'ordre séquentiel, le second l'ordre de « référence » et le troisième est un ordre quelconque. Notons que les deux premiers sont définis de manière unique.*

Preuve. La démonstration est directe en fonction de la définition de l'ordre de référence et en notant qu'une tâche ne peut avoir deux accès sur le même objet partagé.

Définition 11 *La fonction Ψ_x « version apparentée produite » d'*ATHAPASCAN *est définie comme étant la fonction « **dernière version produite** » qui est l'unique fonction $\mathcal{W}_x : R_x \mapsto W_x$ telle que pour tout accès $a_r \in R_x$, $a_w = \mathcal{W}_x(a_r)$ est le plus grand accès dans W_x plus petit (au sens de $\prec_w$) que a_r, au sens de l'ordre total $\prec^x_{Ath}$.*

7.3 Objet partagé de type « collection temporelle »

Dans cette section, nous visons à définir des objets permettant de programmer des applications dont le couplage est décrit par un ensemble de processus communicants par échange de messages dans un canal. Un canal se comporte comme une séquence temporelle de données car elles se distinguent par un ordre (le temps) d'entrée et de sortie du canal. Les données peuvent être le résultat d'un calcul séquentiel ou parallèle. Pour modéliser les données transitant par le canal de communication, il est suffisant de donner un entrelacement *a priori* des opérations d'entrée et de sortie du canal. Nous appelons un tel objet « collection temporelle ».

En se basant sur la définition 5 d'objet partagé, nous pouvons définir une **collection temporelle** comme étant un objet partagé x dont l'ensemble des accès est totalement ordonné. D'un point de vu de la programmation, il n'est pas souhaitable d'autoriser n'importe quel ordre total sur les accès à un

objet partagé. Aussi nous devons restreindre les ordres totaux en fonction de l'ordre total de référence $\prec_{Ath}$ du point de vu des écritures ou des lectures :

Définition 12 *Une **collection temporelle** est un objet partagé x dont l'ensemble des accès est totalement ordonné par un ordre total $\prec_{CT}$ tel que :*
- *$\prec_{CT}$ restreint sur W_x soit compatible avec $\prec^x_{Ath}$, c'est-à-dire que si a et a' dans W_x sont tels que $a \prec^x_{Ath} a'$ alors $a \prec_{CT} a'$.*
- *$\prec_{CT}$ restreint sur R_x soit compatible avec $\prec^x_{Ath}$.*

Notons que la restriction de l'ordre sur les accès en lecture ou en écriture pourrait être définie par rapport à un ordre autre lié au langage de programmation utilisé du coté des écrivains ou du coté des lecteurs, comme par exemple un ordre séquentiel C++.

De cette manière, en considérant que $\prec_{CT}$ est l'ordre $\prec^x_{Ath}$, nous pouvons remarquer que tout objet partagé d'ATHAPASCAN est une collection temporelle. La réciproque n'est pas vraie. En effet, la définition 12 impose seulement que l'ordre entre les écritures et l'ordre entre les lectures soit compatible avec l'ordre d'ATHAPASCAN. En revanche, l'ordre entre une écriture et une lecture peut être différent. Ceci autorise qu'un programme ATHAPASCAN décrive l'ensemble des écritures et qu'un autre programme ATHAPASCAN, distinct du premier, décrive l'ensemble des lectures. La liaison entre les écritures et les lectures dépendra alors d'une fonction, que nous appelons « fonction d'entrelacement », qui est spécifiée par un type de collection temporelle précis. La section suivante présente des exemples de fonctions d'entrelacement classiquement utilisées pour des applications interactives.

7.3.1 Illustration sur quelques fonctions d'entrelacement

Un accès au canal de communication est défini par une sémantique qui précise l'ordre d'arrivée des messages en fonction d'un ordre d'émission. Quelques critères des mécanismes de communication existants sont référencés dans [8] et permettent de préciser une telle sémantique en fonction d'un type de zone tampon associé à un canal :
- **FIFO** : si la zone tampon emmagasine les messages selon un ordre *FIFO* (ou premier arrivé premier servi). Le canal est appelé *FIFO*. Dans ce cas, l'ordre de consommation des messages est identique à l'ordre de leur envoi.
- **« multi-set buffer »** : si par contre, les messages peuvent être consommés d'une manière arbitraire la zone tampon est appelée *multi-set buffer*.
- **« set buffer »** : si une zone tampon ne peut emmagasiner qu'un nombre fixe de messages, *i.e.* les autres messages sont ignorés par une *stratégie de remplacement*, elle est appelée *set buffer*. Cette catégorie correspond au canaux de communication avec perte de message.

Dans cette thèse nous nous concentrons sur les types de la zone tampon de canal motivés par les applications présentes dans la première partie de ce mémoire.

FIFO (premier arrivé premier servi) : la i^{eme} lecture prend la i^{eme} écriture. Dans ce cas nous n'ignorons aucune donnée transitant par le canal. La fonction d'entrelacement peut être définie comme la fonction « *première produite* :»

Définition 13 *La fonction « **première produite** » sur la collection temporelle x est la fonction $\mathcal{F}_x$: $R_x \mapsto W_x$ telle que pour tout accès $a_r \in R_x$, $a_w = ord_w^{-1}(ord_r(a_r))$.*

FIFO borné : si le tampon **FIFO** est de type *set buffer* avec une taille fixe de n, la i^{eme} lecture retrouve la j^{eme} écriture où $j >= i$. La fonction d'entrelacement dépend du comportement à l'exécution de l'application de couplage de codes. Avec nos définitions, il est difficile d'expliciter, et sans grand intérêt pour notre travail, l'accès en écriture qui sera vu par une lecture.

Last-Received (la dernière valeur arrivée est retournée) : c'est un exemple de tampon de type « *set buffer* » où la zone tampon ne peut emmagasiner qu'un seul message : le dernier arrivé. Une collection temporelle ayant cette fonction d'entrelacement est très utile en visualisation d'images produites par des dispositifs d'acquisition ou de calcul plus rapide que l'affichage ou le traitement de ces images : pour ces applications, il n'est généralement pas nécessaire d'afficher ou de traiter toute les images pour un rendu cohérent, cela pourrait nécessiter une utilisation inutile des ressources de calcul.

7.3.2 Proposition pour ATHAPASCAN

Dans la section 7.2.2 nous avons présenté la sémantique d'ATHAPASCAN permettant à tout instant de connaître quelle est la valeur retournée par un accès en lecture. Avec notre proposition de collection temporelle, cette valeur dépend fortement du comportement dynamique des programmes. Néanmoins, nous conservons un ordre compatible avec ATHAPASCAN sur chacun des ensembles d'accès en lecture ou en écriture d'un objet partagé. Nous commençons par présenter l'interface de programmation proposé, en précisant une sémantique de type opérationnel à chacun des opérateurs introduits ; ensuite nous discuterons la définition d'une sémantique associée.

7.3.2.1 Interface de programmation

Dans cette section nous proposons une syntaxe de programmation des « *collection temporelle* », nommé « **TimeCollection** ». L'idée générale est de respecter les mêmes principes que lors de l'utilisation d'un objet partagé *Shared*. La proposition inclue les deux types de collection temporelle avec les fonctions d'entrelacement présentées dans la section précédente, **FIFO** et **LR** (Last-Received).

Cycle de vie d'une collection temporelle. Le cycle de vie d'une « collection temporelle », comme les autres types d'objet partagé, dépend des accès de tâches sur elle. Elle naît à l'exécution de l'instruction de déclaration et elle meurt quand il n'y a plus des tâches y accédant. Un objet de type *collection temporelle* en mémoire partagée est déclaré avec un nom qui doit être unique dans l'application. De plus, on ne peut déclarer un objet collection temporelle qu'avec un droit en écriture ou bien un droit en lecture.

- `TimeCollection_w[p]< TP, SEMANTIQUE sem> x([string name] [, ATTRIBUT attr]);`

  La variable `x` peut être utilisée comme paramètre effectif des tâches productrices d'accès en écriture. Elle représente une collection temporelle d'objets de type `TP`, qui est limité aux *Shared_wp< T >*. « `sem` » définit la sémantique de synchronisation (*FIFO, LR*). Pour le cas de *FIFO*, le nombre maximal des éléments qui peuvent exister ensemble dans la collection est défini par un attribut `size` passé dans le paramètre `attr`. Le `name` doit être unique pour permettre une une identification entre différents processus.

Droit / mode d'accès `Typage`	**Description**
lecture directe `TimeCollection_r`	Accès restreint en lecture seule par la tâche. Les tâches créées ne peuvent requérir qu'un accès en lecture seule sur collection (c'est-à-dire `TimeCollection_r`) et un accès en lecture différée seule sur ses éléments (fonction `get_next` du tableau 7.2).
lecture différée `TimeCollection_rp`	Accès en lecture par la descendance uniquement. La tâche courante ne peut accéder aux éléments de la collection. Les tâches créées ne peuvent requérir qu'un accès en lecture ou lecture différée sur la collection uniquement, c'est-à-dire les types `TimeCollection_r[p]`.
écriture directe `TimeCollection_w`	Accès restreint en écriture seule par la tâche. Les tâches créées ne peuvent requérir aucun accès en écriture, ou bien un accès en écriture différée sur ses éléments, c'est-à-dire de type `Shared_w[p]`.
écriture différée `TimeCollection_wp`	Accès en écriture par la descendance uniquement. La tâche courante ne peut accéder aux éléments de la collection. Les tâches créées ne peuvent requérir qu'un accès en écriture ou écriture différée sur la collection, c'est-à-dire les types `TimeCollection_w[p]`.

Tableau 7.1 *Types autorisés lors de la déclaration des collections d'une tâche. Les types sont composés d'un **droit d'accès** écriture et lecture (`w, r`) qui spécifie le type d'accès qui sera effectué par le sous graphe engendré par la tâche requérant cet accès, et d'un **mode d'accès** direct ou différé (`p` pour postponed) qui spécifie si la tâche requérant un droit d'accès va utiliser ce droit ou se contenter de le passer à sa descendance. Cette information permet de déduire les contraintes de localité des tâches puisque les données réellement accédées sont connues. Il faut noter que c'est impossible d'avoir l'accès à la fois en écriture et en lecture sur un élément de collection. Cette restriction est imposée d'après la séparation de l'ordre d'écriture et l'ordre de lecture.*

– `TimeCollection_r[p]< TP, SEMANTIQUE sem> x([string name] [, ATTRIBUT attr]);`

  La variable `x` peut être utilisée comme paramètre effectif des tâches consommatrices d'accès en lecture. Elle représente une collection temporelle d'objet de type `TP`, que ceci est limité aux *Shared_rp< T >*. « `sem` » définit la sémantique de synchronisation (*FIFO, LR*). Pour le cas de *FIFO*, le nombre maximal des éléments qui peuvent exister ensemble dans la collection est défini par un attribut `size` passé dans le paramètre `attr`. Le `name` doit être unique pour permettre une une identification entre différents processus.

Enfin ajoutons qu'il n'est possible de déclarer qu'un couple « écriture-lecture » de collections temporelles ayant le même nom et la même sémantique *sem*.

Droit d'accès et l'interface. Comme on l'a écrit précédemment pour les données partagées de type *Shared* dans ATHAPASCAN, chaque tâche spécifie au moment de sa création les accès qui seront effectués sur la mémoire partagée au cours de son exécution ou lors de l'exécution de toute sa descendance. Les différentes données, qui seront accédées, sont spécifiées dans la liste des paramètres de la tâche et la description de ces droits d'accès (écriture `w`, lecture `r`, éventuellement avec les modes

Accès `Interface`	**Description**
accès `aux éléments` **en écriture** `Shared_wp<T> get_next();`	Si l'accès demandé est autorisé (`TimeCollection_w< TP > y`) l'appel `y.get_next()` retourne la référence d'un élément de collection selon **l'ordre d'écriture** avec le droit d'accès en écriture différée `Shared_wp`.
accès `aux éléments` **en lecture** `Shared_rp<T> get_next();`	Si l'accès demandé est autorisé (`TimeCollection_r< TP > y`) l'appel `y.get_next()` retourne la référence d'un élément de collection selon **l'ordre de lecture** avec le droit d'accès en lecture différée `Shared_rp`.

Tableau 7.2 *L'accès à la valeur d'une donnée partagée de type **collection temporelle** et ses éléments se fait à l'aide des fonctions d'interface de collection et l'interface associé à ses éléments. Il faut noter que l'appel `y.get_next()` retourne la référence avec un accès différé, car uniquement les tâches de la descendances ont le droit d'accès effectif.*

Type du paramètre **formel**	Type (requis) du paramètre **effectif**
`TimeCollection_w[p]`	`TimeCollection_wp`
`TimeCollection_r[p]`	`TimeCollection_rp`

Tableau 7.3 *Règles de conversion des types de paramètres lors de la création d'une tâche. Si une tâche requiert un paramètre d'un certain type formel, alors la tâche mère, lors de la création de la tâche fille, doit nécessairement posséder la référence sur la donnée partagée avec un type compatible lui permettant la création. Les types acceptée sont situés dans la colonne de droite du tableau.*

différés `wp` ou `rp`) est faite à l'aide d'un mécanisme de typage.

Rappelons qu'il n'est pas possible d'avoir à la fois un accès en écriture et en lecture sur une même collection. Cette restriction est imposée d'après la séparation de l'ordre d'écriture et l'ordre de lecture. La règle générale est qu'une tâche ne peut accéder, soit directement, soit via sa descendance, à un objet pour lequel elle n'a pas déclaré l'accès correspondant (restriction des droits d'accès) : les tâches ne font pas d'effet de bord sur la mémoire. Les différents types possibles et l'interface associé sont répertoriés dans les tableaux 7.1 et 7.2.

Règles de conversion lors du passage de paramètres. Le tableau 7.3 énumère la compatibilité, lors de la création d'une tâche, entre les types formels et effectifs des références sur les collections temporelles. Ces règles de conversion ne sont valables que lors de la création des tâches : lors d'appels classiques de fonction ce sont les règles standards C++ qui sont appliquées.

```
struct put{
  void operator()( int n, Shared_w<int> x )
  {
    x.write( n );
  }
};
struct product{
  void operator()( int n,
    TimeCollection_w< Shared_w<int> > x)
  {
    for (int i=0; i<n; i++)
      Fork< put >()( i+1, x.get_next() );
  }
};
struct print{
  void operator()( int n,
         Shared_r<int> r )
  {
    std : :cout « "fact(%d)=%d" « n «
r.read();
  }
};
cumulative mult( int& x, const int& y )
{
  x *= y;
}
```

```
struct get{
  void operator()( Shared_r<int> x,
        Shared_cw<int, mult> b)
  {
    b.cumul( x.read());
  }
};
struct consume{
void operator()(int n,
    TimeCollection_r< Shared_r<int> > x
    ,Shared_cwp<int, mult> b )
  {
    for (int i=0; i<n; i++)
      Fork< get >()( x.get_next(), b );
  }
};
int main( int argc, char** argv )
{
  int n= atoi( argv[1] );
  TimeCollection_wp< Shared_wp<int> , FIFO>
       x_writer("thecollection");
  TimeCollection_rp< Shared_rp<int> , FIFO>
       x_reader("thecollection");
  Shared<int> r;
  Fork< product >()( n, x_writer );
  Fork< consume >()( n, x_reader, r );
  Fork< print >()( n, r );
  return 0;
}
```

Figure 7.3 ***Calcul du n-i^{eme} nombre Factoriel en* ATHAPASCAN.** *A chaque itération dans la tâche* `product` *un nouvel élément de la collection est créé et ajouté. Et à chaque itération dans la tâche* `consume` *un élément est sorti en respectant l'ordre FIFO.*

7.3.2.2 Contraintes de précédences et ordonnancement

Les accès et le passage de paramètres définis ci-dessus imposent des restrictions sur les opérations possibles. Ces restrictions servent à garantir les propriétés et définitions des collections temporelles. Les accès définis sur les collections temporelles imposent des contraintes de précédences qui sont :

- deux tâches t_1 et t_2 qui accèdent avec le **même** type d'accès (`w` ou `r`) à une collection doivent être exécutées suivant leur ordre de création.
- deux tâches t_1 et t_2 qui accèdent avec des types d'accès **différents** (par exemple, l'une en `w` et l'autre en `r`) à une collection sont concurrentes.

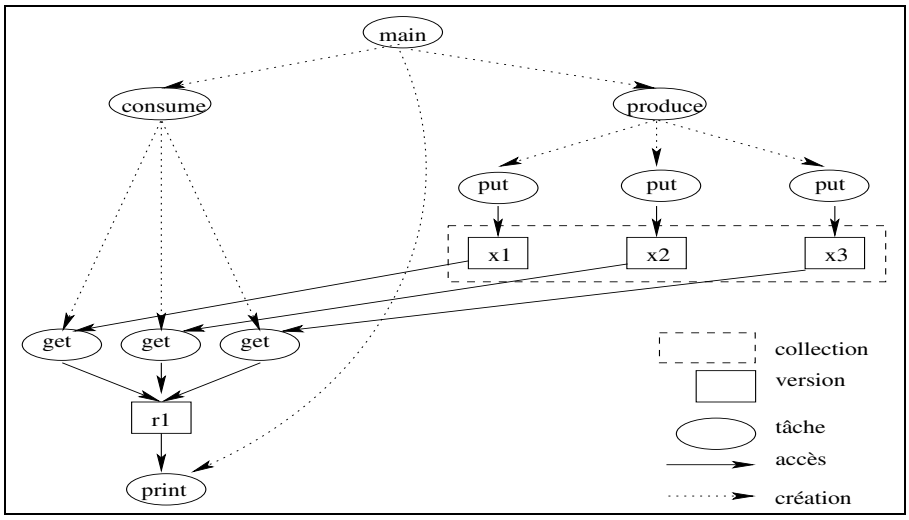


Figure 7.4 ***Graphe de flot de données avec schéma d'exécution du calcul du n-i^{eme} nombre Factoriel où*** $n = 3$***. Les six tâches « put » et « get » accèdent en concurrence les trois éléments*** x_1, x_2 *et* x_3 *d'une collection temporelle typé* `TimeCollection`*.*

7.3.2.3 Exemple de programmation

L'objectif de cet exemple est de montrer l'utilisation des données partagées de type *TimeCollection* pour un calcul parallèle plus faible que le type *Shared*. La figure 7.3 présente un programme qui calcule le n-i^{eme} nombre Factoriel en utilisant des données partagées de type *TimeCollection*. Ce programme est basé sur un algorithme ou le producteur génère la séquence des nombres et où le consommateur les lits (suivant un ordre FIFO) et les accumule dans une variable partagée. La tâche *product* génère les nombres dans la collection de manière séquentielle (en créant les tâches *put*). La tâche *consume* lit ces nombres en créant les tâches *get*, et en respectant la sémantique *FIFO*. Le résultat final est créé de manière concurrente en multipliant des valeurs retirées sur une donnée partagée de type *Shared*.

La figure 7.4 montre le graphe de flot de données à l'exécution de ce programme pour $n = 3$. Pour cet exemple on peut éviter toutes les attentes inutiles en choisissant le nombre maximal des éléments de collection supérieur à n.

Il est bon de noter ici que si le nombre de données produites est infini, nous ne pouvons pas réaliser ce type d'application sans collection temporelle. Car avec l'objet partagé `Shared`, il peut ne pas exister d'ordonnancement compatible avec la sémantique ATHAPASCAN qui nécessite un espace mémoire fini. La collection temporelle permet un meilleur ordonnancement selon les contraintes sur les performances ou bien sur l'espace mémoire.

7.4 Conclusion

Afin de pouvoir facilement réaliser l'interaction non forcément synchrone entre différents processus d'une application distribuée, nous avons défini une nouvelle abstraction, appelée *collection*

temporelle, dans le contexte du modèle de programmation parallèle ATHAPASCAN. La collection temporelle est définie comme un objet partagé. La sémantique d'accès dépend de la stratégie d'entrelacement des écritures et des lectures. La proposition présentée pour le langage ATHAPASCAN donne un interface de programmation et des contraintes de précédences pour l'ordonnancement.

Ce chapitre a présenté comment coupler deux processus dans ATHAPASCAN. Le chapitre suivant présente un second aspect nécessaire pour encapsuler la distribution d'objets inhérente aux codes parallèles.

8

Collection spatiale d'objets partagés

Sommaire

Une « collection spatiale » est un objet partagé dont l'ensemble des accès aux éléments est associé à une fonction d'indexation. A la différence d'une collection temporelle il n'existe pas d'ordre entre ces accès. L'exemple typique est la distribution de données partagées sur différentes machines. L'objectif de ce chapitre est de définir cette notion de collection spatiale. Puis nous proposons une définition dans le langage ATHAPASCAN. *De la même manière que pour les collections temporelles, la déclaration d'un accès à un objet partagé d'une collection spatiale est découplé de l'accès effectif à sa donnée. Le chapitre se termine par un exemple illustrant ce nouveau concept dans* ATHAPASCAN.

8.1 Introduction

Les applications de couplage de codes sont des applications qui sont à la foi parallèles et distribuées. Dans le cadre de notre travail d'extension du langage ATHAPASCAN pour le couplage de codes, il est essentiel de définir une entité qui puisse capturer l'aspect distribué des données d'une application parallèle. Nous appelons cette entité *collection spatiale*. Le concept de collection d'objets est une solution proposée par plusieurs projets de recherche [72, 73, 69] pour répondre à ce besoin de *distribution des données partagées*. Il existe deux regards différents sur ce concept.

- **Collection implicite :** une collection peut être considérée comme un objet qui est divisé en plusieurs sous-objets de même fonctionnalité. L'objectif est de garder la même fonctionnalité au niveau de la collection qu'au niveau des sous-objets de type collection tout en masquant leur aspect distribué. La problématique liée à cette approche est la transformation automatique des invocations sur la collection vers des invocations sur ces sous-objets [72, 73], en gérant la distribution et redistribution des données nécessaires.
- **Collection explicite :** ici l'objectif est de permettre un accès individuel à plusieurs objets d'une collection. Une collection spatiale se représente comme une structure de données partagées avec un accès aléatoire à ses éléments. L'interface sur une collection spatiale n'est pas plus identique à celle de ses éléments. Avec cette approche, la problématique est de donner une sémantique aux accès concurrents aux éléments qui permette de masquer les latences des accès aux sous-objets distribués. Nous nous plaçons dans cette approche pour définir la notion de collection spatiale.

L'objectif de ce chapitre est de définir la notion de collection spatiale dans le cadre du langage ATHAPASCAN qui puisse permettre de manipuler une description distribuée de données nécessaire au couplage de codes parallèles.

La définition des collections spatiales et leur sémantique d'accès est l'objet de la section suivante. Puis nous proposons une syntaxe de programmation pour collection spatiale dans le langage ATHAPASCAN. Enfin, un programme d'exemple est présenté.

8.2 Objet partagé de type « collection spatiale »

Un ensemble d'objets accessibles individuellement par un identificateur, représente une « collection spatiale ». Les opérations sur la collection permettent l'accès vers ses éléments. En utilisant la définition 5 d'objet partagé du chapitre précédent (page 70). Nous définissons une collection spatiale de la

manière suivante.

Définition 14 *Une **collection spatiale** x est un objet partagé dont l'ensemble $\mathcal{A}$ des accès possède une fonction d'indexation $\mathcal{I}_x : \mathcal{A} \mapsto IN$ qui pour tout accès retourne un entier. On appelle $\mathcal{I}_x(a)$ l'adresse de l'accès a dans la collection spatiale x.*

Notons que la fonction d'indexation pourrait être aussi définie sur un autre ensemble que les entiers. Une collection spatiale peut être proche d'un vecteur d'accès à des objets partagés. La différence réside dans l'accès aux éléments (les sous-objets partagés) de la collection qui retourne toujours des accès en mode différé et non un accès direct vers la donnée.

Une collection étant un objet partagé, il existe donc deux catégories d'accès : les accès à la collection vue comme une structure de donnée, et les accès aux éléments. Chacune de ses catégories possède un type et un mode d'accès comme présenté pour ATHAPASCAN dans le chapitre 2. La section suivante présente l'interface de programmation des collections spatiales.

8.3 Interface de programmation

Dans cette section nous proposons une syntaxe de programmation des *collection spatiale*, nommé « `SpaceCollection` ». L'idée générale est de respecter les mêmes principes que lors de l'utilisation des objets partagés *Shared* ou des objets `TimeCollection`. Nous considérons ici pour simplifier que les adresses des éléments sont des nombres entiers positifs.

Cycle de vie d'une collection et de ses éléments. Le cycle de vie d'une « collection spatiale », comme les autres types d'objet partagé, dépend des accès des tâches sur elle. Elle naît lors de l'exécution de l'instruction de sa déclaration et elle est détruite quand il n'y a plus des tâches y accédant.

Déclaration Une collection spatiale peut-être déclarée de manière suivante :

- `SpaceCollection< TP > x;`
  La variable `x` peut être utilisée comme paramètre effectif des tâches. Elle représente une collection spatiale d'objet de type `TP`, qui est ici limité aux types *Shared, TimeCollection* ou *SpaceCollection*. La collection est initialement vide : elle ne possède aucun élément.

Droit d'accès et l'interface. Comme l'on a écrit pour les autres données partagées, dans ATHAPASCAN chaque tâche spécifie au moment de sa création les accès qui seront effectués sur la mémoire partagée au cours de son exécution ou lors de l'exécution de toute sa descendance. Le type d'accès à la collection contraint les types d'accès possible à ses éléments. Les différents types possibles et l'interface associée sont répertoriés dans les tableaux 8.1 et 8.2.

Règles de conversion lors du passage de paramètres. Le tableau 8.3 énumère la compatibilité, lors de la création d'une tâche, entre les types formels et effectifs des références sur les données partagées. Ces règles de conversion ne sont valables que lors de la création des tâches : lors des appels classiques de fonction ce sont les règles standards C++ qui sont appliquées. Il faut noter

Droit / mode d'accès `Typage`	**Description**
accès différé aux éléments `SpaceCollection_`tp	Accès restreint aux éléments par la descendance uniquement. La tâche courante ne peut accéder aux éléments de la collection. Le type d'accès t donne le type d'accès des éléments de la collection ($t = r$ ou w ou rw). Des restrictions existent et sont données dans le texte.
modification directe `SpaceCollection_m`	Accès en modification de la structure de la collection par la tâche courante uniquement. Cet accès est exclusif à tous les autres accès.
modification différée `SpaceCollection_mp`	Accès en modification de la structure de collection par la descendance uniquement. Les tâches filles peuvent requérir le type modification d'accès sur la collection.

Tableau 8.1 *Types autorisés lors de la déclaration des paramètres « collection spatiale » d'une tâche. Les types sont composés d'un **droit d'accès** (accès à la collection qui donne le type d'accès aux éléments) qui spécifie le type d'accès qui sera effectué par le sous graphe engendré par la tâche requérant cet accès. Et pour un **mode d'accès** direct ou différé (*`p`* pour postponed) qui spécifie si la tâche requérant un droit d'accès va utiliser ce droit ou se contenter de le passer à sa descendance. Cette information permet de déduire les contraintes de localité des tâches puisque les données réellement accédées sont connues.*

que le paramètre effectif avec l'accès en modification peut être passé aux paramètres formels quelque soit les droits d'accès aux éléments, cet accès est un accès exclusif.

8.3.1 Contraintes de précédence et régles de composition

Les contraintes de précédence entre les tâches sont déduites des accès effectués sur les données partagées. L'accès à la collection spatiale doit respecter les contraintes de précédence suivantes.

Définition 15 *Les **contraintes de précédence** entre les tâches accédant à une collection spatiale sont déduites des accès effectués sur cette collection spatiale.*

- *Une tâche accédant une collection spatiale pour tout autre droit que celui de modification, (paramètre typé* `SpaceCollection_`tp*) ne peut pas être exécutée avant que toutes les tâches précédences accédants en modification (paramètre typé* `SpaceCollection_m[p]`*) ne soient terminées.*
- *Deux tâches accédant à une même collection avec un accès t en lecture sont concurrentes.*
- *Sinon les tâches doivent s'exécuter en respectant leur ordre de création.*

Ces contraintes de précédence définissent un ordre partiel, qui doit être respecté par toute exécution valide. De plus, ces contraintes de précédence ne sont valides que pour les règles de composition décrites ci-dessous.

Règles de composition Toutes les compositions d'un type d'accès à un objet partagé TP est une collection spatiale ne sont pas autorisées. Notons par T_c le type d'accès à la collection spatiale et T_e le type d'accès aux éléments de la collection (*i.e.* TP), alors :

Access `Interface`	**Description**
accès différée aux éléments TR `get_item(int i);` `int size();`	Si l'accès demandé est autorisé (`SpaceCollection_`t`p< TP > y`) l'appel `y.get_item(i)` retourne la référence de l'élément `i` de la collection avec le type et mode d'accès TR qui est l'intersection entre le type d'accès TP et t : l'intersection de $rpwp$ et $r[p]$, réciproquement $w[p]$, vaut rp, réciproquement wp ; l'intersection de $rpwp$ et $rpwp$ vaut $rpwp$. Des restrictions existes et sont données dans le texte. L'appel `y.size()` retourne la taille actuelle de collection.

modification exclusive
`boolean resize(int N);`
`int size();`
	Si la modification est autorisée (`SpaceCollection_m[p]< TP > x`) l'appel `x.resize(N)` change la taille de collection ; cette fonction est en réalité une instruction de déclaration des éléments partagés de collection. Cet appel déclare automatiquement et injecte les nouveaux accès dans la collection.

Tableau 8.2 *L'accès à la valeur d'une donnée partagée de type **collection spatiale** et ses éléments se fait à l'aide des fonctions d'interface de collection et l'interface associée à ses éléments.*

Type du paramètre **formel**	Type (requis) du paramètre **effectif**

`SpaceCollection_`t`[p]< TP >`	`SpaceCollection_`tp`< TP >`
	`SpaceCollection_m[p]< TP >`
	`SpaceCollection< TP >`

`SpaceCollection_m[p]< TP >`	`SpaceCollection_m[p]< TP >` `SpaceCollection< TP >`

Tableau 8.3 *Règles de passage des types de paramètres lors de la création d'une tâche. Si une tâche requiert un paramètre d'un certain type formel, alors la tâche mère, lors de la création de la tâche fille, doit nécessairement posséder la référence sur la donnée partagée avec un type compatible lui permettant la création. Les types acceptables sont situés dans la colonne de droite du tableau.*

- Si $T_c = rw$, T_e doit être de type rw ou $rpwp$ dans le cas des objets partagés ATHAPASCAN ou T_e est le type d'une collection temporelle.
- Si $T_c = r$, T_e doit être de type $rpwp$, r ou rp dans le cas des objets partagés ATHAPASCAN.
- Si $T_c = w$, T_e doit être de type $rpwp$, w ou wp dans le cas des objets partagés ATHAPASCAN.

Puisque les modes d'accès sont des types C++, ces règles de composition sont vérifiées à la compilation des programmes.

8.4 Exemple de programmation

Dans cette section nous décrivons un exemple d'utilisation des « collections spatiales » pour un calcul parallèle. Considérons un fichier dans le système de fichier distribué qui contient plusieurs blocs distribués sur des machines différentes. Cet exemple concerne tout calcul qui peut être divisé en calculs indépendants. Dans le système de fichier considéré, un fichier distribué est une série de blocs.

Les modes d'ouverture d'un fichier sont passés aux accès de ses blocs. A partir d'un descripteur de fichier, nous pouvons extraire le nombre de blocs *Nblocks* et le site de stockage de ces blocs (get_loc) ou (set_loc). Les figures 8.1, 8.2, 8.3 présentent le code du programme et son graphe de flot de données pour une exécution sur un fichier distribué par blocs. Les tâches de calcul (ici Gzip) sont placées sur les sites possédant les données.

8.5 Conclusion

Afin de pouvoir facilement programmer l'accès à un ensemble d'objets partagés répartis sur différents processus, nous avons défini une nouvelle abstraction, appelée *collection spatiale*. La collection spatiale est définie comme un ensemble d'objets partagés qui se distinguent par une fonction d'indexation explicite. La sémantique d'accès à la collection est séquentielle.

La proposition présentée pour le langage ATHAPASCAN montre comment réaliser cette nouvelle abstraction avec la définition étendue d'objet partagé d'ATHAPASCAN. Ensuite un exemple de programme montre l'utilisation des collections spatiales pour construire un graphe de flot de données qui décrit le parallélisme dû aux entrées distribuées. De plus la proposition permet de recouvrir les latences d'accès par le calcul en différant les accès effectifs aux données de leurs déclarations.

```
1 . const char* file1="DFS1 :/tmp/text.txt";
2 . const char* file2="DFS1 :/tmp/text.zip";
3 . int main(){
4 .   SpaceCollection< Shared_rpwp< Block > > dfile1, dfile2;
5 .   File f1, f2;
6 .   f1=open(file1,"R");
7 .   f2=open(file2,"W"); // New created file has nothing blocks
8 .   Fork<init>() ( dfile1, f1.Nblocks() );
9 .   Fork<init>() ( dfile2, f1.Nblocks() );
10.   Fork<Readfile>() ( dfile1, f1 );
11.   Fork<DGzip>() ( dfile1, dfile2 );
12.   Fork<Writefile>() ( dfile2, f2 );
13.   Fork<Closefile>() ( f1, dfile1 );
14.   Fork<Closefile>() ( f2, dfile2 );
15. } //-----------------------------------------------
16. struct init {
17.   void operator()(SpaceCollection_m< Shared_rpwp<Block> > dfile, int n)
18.   {
19.     dfile.resize(n);
20.   }
21. }; //-----------------------------------------------
22. struct Readfile {
23.   void operator()(SpaceCollection_wp< Shared_rpwp<Block> > dfile, File f)
24.   {
25.     for(i=0; i < dfile.size(); i++){
26.       dfile.set_loc(i, f.block(i)->get_loc());
27.       Fork<Readblock>(dfile.get_loc(i)) ( dfile.get_item(i), f.block(i) );
28.     }
29.   }
30. }; //-----------------------------------------------
31. struct Readblock {
32.   void operator()(Shared_w<Block> refblock, Block* b) {
33.     refblock.write(b.read());
34.   }
35. };
          //------- To be continued ---------
```

Figure 8.1 ***Compression d'un fichier distribué en utilisant la notation de « collection spatiale »*** *(première partie).*

```
36. struct DGzip {
37.   void operator()(SpaceCollection_rp< Shared_rpwp<Block> > df1,
38.                       SpaceCollection_wp< Shared_rpwp<Block> > df2)
39.   {
40.     for(i=0; i < df1.size(); i++){
41.       df2.set_loc(i, df1.get_loc(i));
42.       Fork<Gzip>(df1.get_loc(i)) ( df1.get_item(i), df2.get_item(i) );
43.     }
44.   }
45. }; //-----------------------------------------------
46. struct Gzip {
47.   void operator()(Shared_r<Block> b1, Shared_w<Block> b2){
48.     Block* tmp;
49.     gzip(b1.read(), tmp);
50.     b2.write(tmp);
51.   }
52. }; //-----------------------------------------------
53. struct Writefile {
54.   void operator()(SpaceCollection_rp< Shared_rpwp<Block> > dfile, File f)
55.   {
56.     f.Nblocks(dfile.size());
57.     for(i=0; i < dfile.size(); i++){
58.       f.set_loc(i, dfile.get_loc(i));
59.       Fork<Writeblock>(dfile.get_loc(i)) ( dfile.get_item(i), f.block(i) );
60.     }
61.   }
62. }; //-----------------------------------------------
63. struct Writeblock {
64.   void operator()(Shared_r<Block> refblock, Block* b) {
65.     b.write(refblock.read());
66.   }
67. };
68. struct Closefile {
69.   void operator()(File f, SpaceCollection_m< Shared_rpwp<Block> > dfile) {
70.     f.close();
71.     dfile.resize(0);
72.   }
73. };
```

Figure 8.2 ***Compression d'un fichier distribué en utilisant la notation de "collection spatiale"*** *(seconde partie).*

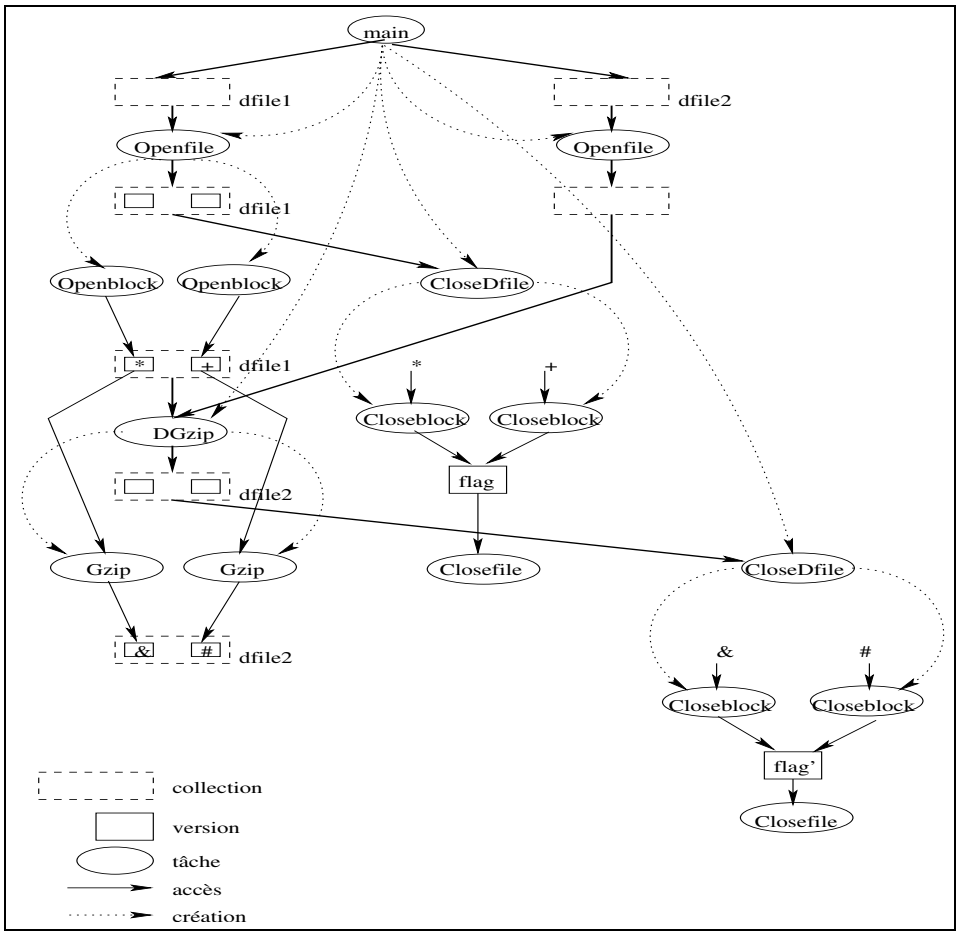


Figure 8.3 ***Graphe de flot de données d'une exécution du programme de compression d'un fichier distribuée où il y a deux blocs dans le fichier initial.*** *L'intérêt de ce graphe est de montrer comment, en utilisant des collections, on peut construire le graphe de flot de données en parallèle, avec les tâches de calcul sur ses éléments distribués. Par exemple, la tâche CloseDfile peut être exécutée en parallèle avec les tâches filles de Openfile pour dfile1 et la tâche CloseDfile en parallèle avec les filles de DGzip pour dfile2.*

IV

Applications

9

SIMBIO : couplage de codes parallèles pour la simulation numérique

Sommaire

Ce chapitre présente une application réelle de couplage de codes dans le domaine de simulation numérique. La description du couplage est donnée par un algorithme qui pilote les invocations aux méthodes des objets CORBA. Bien que cet algorithme puisse être exécuté comme un programme parallèle, il suit toujours une sémantique de type séquentielle de l'invocations des méthodes. Nous décrivons ce type du couplage par la ***collection spatiale*** *d'objets partagés. Enfin sur cet exemple réel, nous évaluons* HOMA *qui corroborent les résultats théoriques du chapitre 5.*

9.1 Application SIMBIO

Cet exemple vient du projet SIMBIO[1] [13] dans la chimie calculatoire. Le but de SIMBIO est de construire une application distribuée pour simuler une structure moléculaire complexe (une protéine) entourée par un solvant.

La figure 9.1 montre l'algorithme de couplage de l'application SIMBIO comme un schéma d'intégration multi-échelles. La dynamique moléculaire (l'objet *md*) et la méthode de continuum (l'objet *cm*) sont connectées par un objet spécial (l'objet *coupling*) qui calcule la contribution des termes de couplage sur chacun des modèles physiques. Tous les objets sont parallèles : les données *P* et *S* qui stockent les positions et les vitesses des atomes ; les données *Fmd, Fcm* et *Fg* qui stockent des champs des forces ; et la surface *S* qui représente la surface de séparation des deux modèles. Ce sont des objets ou des données CORBA représentés sous forme de vecteurs distribués. Ces données et les objets sont les « objets parallèles » de CORBA [57, 77, 69] qui sont distribués parmi les processeurs : l'état de ces objets est distribué parmi les processeurs.

La description du flot de données pour une itération dépend de l'étape actuelle et a été donné par un programme client basé sur CORBA comme présenté dans la partie gauche de la figure 9.1. La partie droite représente le graphe de flot de données entre les invocations des méthodes. Un nœud boîte représente une donnée et un nœud ellipse représente une tâche. Une flèche d'une tâche à une donnée représente le fait que la données est écrite par la tâche. Une flèche d'une donnée à une tâche représente le fait que la donnée est lue par la tâche. Chaque tâche dans le graphe de flot de données correspond à l'invocation d'une méthode sur l'objet CORBA associé (*DM*, *CM* ou *coupling*, aux lignes 4, 5, 6, 9, 10, 12).

9.1.1 Complexite de communication

La section 5.3 de chapitre 5 présente une analyse de complexité générale des applications mais ce paragraphe focalise sur le goulot d'étranglement dans le programme de la figure 9.1 utilisant un modèle de coût simple.

Supposons que chaque communication entre les applications MD et CM a la même taille de L octets. Typiquement, L vaut de $KOctets$ à $MOctets$. Ainsi une itération du programme de la figure 9.1, avec l'évaluation complète des tâches CM, a besoin de communiquer $5L$ octets. L'utilisation de P processus pour chacune des applications MD et CM (excepté le code du couplage), induit un volume de communication de $5L/P$ octets envoyés ou reçus entre les deux applications.

[1] SIMBIO a été un projet ARC subventionné par INRIA, entre 1997 et 1999

```
1. for(int k=0; k <MaxTimeStep; k++) {
2.   md->computeFF( P^in, Fmd^out );
3.   if (cmstep( k )) { //true iff CM step
4.     md->computeRhs( P^in, b^out );
5.     cm->computeA( S^in, P^in );
6.     cm->computePolarization(b^in, Fcm^out);
7.   }
8.   // Coupling sited on "md".
9.   coupling->mix(Fcm^in, Fg^out, Fmd^inout);
10.   md->integrate(V^inout, Fmd^in, P^inout);
11.   if (cmstep( k )) {
12.     cm->integrate( S^inout, Fg^in );
13.   }
14.} // End of main loop.
```

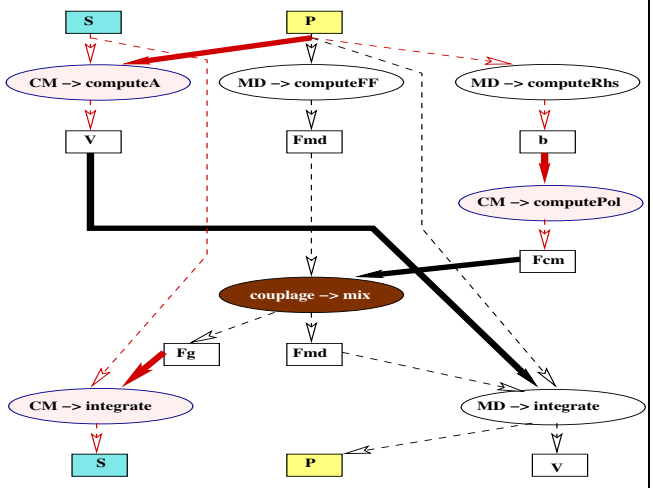


Figure 9.1 *A gauche : le code du pilote (client) SIMBIO ; les exposants représentent le mode de passage des paramètres des fichiers IDL des interfaces non présentés ici. A droite : le graphe de flot de données pour une itération où la condition cmstep est vraie. Les 5 flèches pleines représentent les communications des données entre les deux codes.*

Avec une implémentation standard de CORBA, le pilote, qui décrit ce flot de données, est impliqué dans chaque communication : le processus associé reçoit et envoie $5L$ octets de données, quelque soit le nombre de processus utilisés pour paralléliser les applications. Ainsi du point de vue de la communication, le pilote est le goulot d'étranglement de l'application de couplage de codes. Ceci est dû à la sémantique d'invocation de méthode dans CORBA qui exige la communication explicite de ses paramètres effectifs.

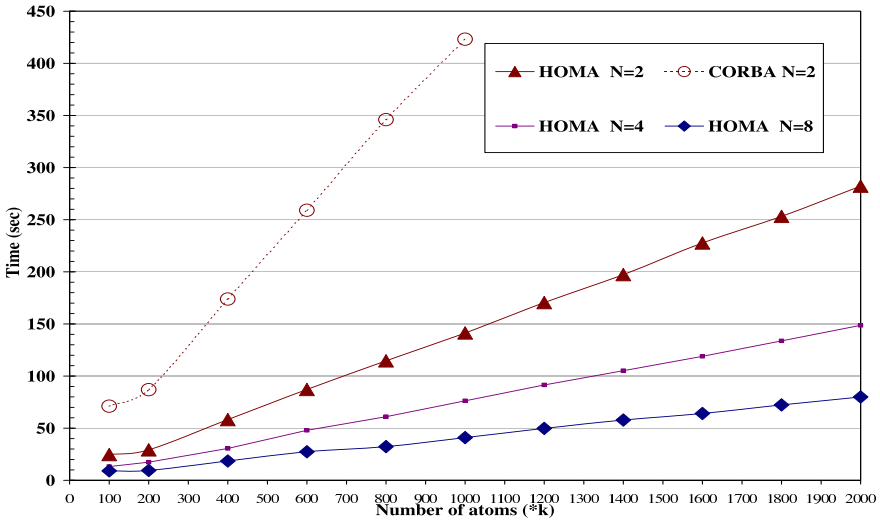
Selon le chapitre 5, en utilisant HOMA le pilote ne participe pas du tout dans la transmission des données[2] entre les applications : chaque processus d'applications *md* et *cm* envoient ou reçoivent en parallèle de données $5L$ octets. Néanmoins, le goulot d'étranglement dépend alors uniquement de l'implémentation de la tâche de couplage (distribuée ou centralisée).

9.1.2 Description du couplage par « collection spatiale »

Le code pilote décrit le couplage entre deux codes MD et CM. Chacun de ces codes couplés exploite un ensemble d'objets CORBA répartis sur différents processus. La sémantique d'accès à l'objet CORBA est *séquentielle* : à tout point dans l'exécution de l'algorithme du client, la valeur d'une variable est donnée par la dernière invocation, dans une exécution séquentielle, qui produit (**out**) ou qui modifie (**inout**) la variable. Donc nous décrivons cet algorithme de couplage comme une composition d'un ensemble d'invocations sur deux *collections spatiales* d'objets partagés de type *séquentiel* (ici MD et CM).

Du point vu de l'accès à un objet partagé, chaque invocation de méthode CORBA est une opération de « modification ». CORBA ne permet pas de décrire le mode d'accès aux états intérieurs d'un objet

[2]Sauf pour faire les invocations à distance, qui n'est pas considéré dans cette analyse.



Atomes ($\times 10^3$)	p		
	3	6	12
	Accélération		
100k	2.72	5.09	7.41
200k	2.98	5.00	9.15
400k	2.99	5.69	9.39
600k	2.98	5.40	9.46
800k	2.98	5.60	10.57

Figure 9.2 *A gauche : le temps d'exécution SIMBIO avec plusieurs objets/application. A droite : le gaine d'*HOMA *par rapport à CORBA.*

lors de la définition de ses méthodes, donc nous sérialisons les accès aux objets CORBA par le droit d'accès en modification (ici « _rw » pour les « Shared » d'ATHAPASCAN).

9.2 Exécution sur une grappe de PC

L'expérience a été faite en utilisant l'implémentation OmniORB3 [41] de CORBA. Les programmes ont été exécutés sur le iCluster1 de INRIA à Grenoble (PC 733Mhz, 256MByte, réseau 100Mbit/s). Les temps sont donnés en utilisant la version actuelle de l'environnement de HOMA, qui utilise seulement le mode *get* présenté dans le chapitre 5. L'algorithme d'ordonnancement est un algorithme simple « *greedy* » qui exécute la première tâche prête de la liste des tâches créées.

Cette expérience mesure la capacité d'HOMA à exécuter en parallèle des invocations avec des dépendances complexes entre invocations. Le figure 9.1 présente l'algorithme et le graphe de flot de données associées à cette application SIMBIO. Tous les paramètres des invocations sont des séquences de taille M (le nombre d'atomes).

Dans les expériences, les objets MD et CM sont respectivement distribués sur N_1 et N_2 sous-objets de même interface. Chaque invocation sur ces objets est traduite en plusieurs invocations sur chacun des sous-objets. Noter que le flot de contrôle dépend de l'évaluation de `cmstep` qui est seulement connu à l'exécution. Dans cette expérience $N_1 = 2N_2 = N$ et nous choisissons `cmstep` être vrai toute les deux itérations. Tous les objets sont placés sur des processeurs différents.

Notons par T_1 le travail (temps d'exécution séquentielle) de cette application SIMBIO, T_∞ le chemin critique, $C_1 = O(M)$ le volume de communication et par C_∞ le retard de communication. En première approximation nous pouvons considérer que les algorithmes utilisés dans SIMBIO sont linéaires par rapport au nombre d'atomes, ainsi $T_1 = O(M)$ et $T_\infty = O(M(\frac{1}{N_1} + \frac{1}{N_2}))$. Cette supposition mène à $T_p^{homa} = O(\frac{M}{N}(1+h))$ en utilisant notre modèle de prédiction.

Le côté gauche de la figure 9.2 montre le temps d'exécution de 10 itérations. Comme l'analyse du

modèle de coût le prévoit, il y a une relation linéaire entre le parallélisme des objets en utilisant HOMA et le temps d'exécution. Les diminutions de temps sont quasiment linéaires en fonction du nombre croissant des processeurs utilisés. Le côté droit de la figure 9.2 présente l'accélération de HOMA par rapport à une utilisation de CORBA standard et pour différents nombres d'atomes et de processeurs $p = N_1 + N_2$.

Il y a deux raisons pour expliquer la diminution de l'efficacité lorsque le nombre p de processeurs croît. La première est due au nombre différent des sous-objets alloués aux objets DM et CM ($N_1 et N_2$) ce qui implique une diffusion / réduction des données échangés (dans cette expérience il y a toujours deux sous-objets de MD pour un sous-objet de CM), ce qui met en série une partie des communications. La deuxième raison est due à l'augmentation du nombre de processeurs ce qui a pour effet de diminuer le grain de calcul et la taille de messages : le surcoût d'HOMA devient plus important.

9.3 Bilan

Dans ce chapitre nous avons décrit une application réelle de couplage de codes en utilisant des collections spatiales d'objets partagés de type séquentiel. Ce modèle de couplage respecte une sémantique de type séquentielle comme celle d'ATHAPASCAN et est donné par un unique programme qui pilote la simulation.

Cet exemple montre de bons résultats en utilisant HOMA vis-à-vis d'une implantation standard de CORBA qui corroborent les résultats théoriques présentés précédemment. Elle nous montre aussi un manque dans la possibilité de décrire les accès sur des objets lors de la déclaration de ses méthodes dans l'IDL CORBA, qui nous oblige à exécuter séquentiellement les invocations sur un même objet.

10

Sappe : couplage de codes parallèles pour la réalité virtuelle

Sommaire

Ce chapitre présente une application réelle de couplage de codes dans le domaine de la réalité virtuelle. La description du couplage est donnée par un ensemble de canaux de communication. Les données transférées sont estampillées par le temps d'entrée dans les canaux et elles sont partagées par plusieurs tâches de calcul. Nous décrivons ce type du couplage par une ***collection spatiale*** *d'objets partagé de type* ***collection temporelle****. Enfin nous mesurons les performances des collections temporelles sur un exemple élémentaire.*

10.1 Introduction

En informatique graphique, suite à un engouement général pour les jeux vidéos, les films d'animation ou encore la réalité virtuelle, l'animation d'objets 3D a énormément évoluée vers des algorithmes simulant des comportements de plus en plus réalistes mais également de plus en plus complexes. Ces animations nécessitent désormais une puissance de calcul importante aussi bien pour leur simulation que pour leur visualisation.

Dans le domaine de la réalité virtuelle, des prototypes à base de grappes de PCs permettent de créer des systèmes immersifs multi-écrans sans utiliser des machines graphiques telles que les SGI Onyx, diminuant ainsi les coûts et augmentant la flexibilité (dynamicité de l'infrastructure logicielle et matérielle) et le passage à l'échelle (prototypage et tests préalables sur petites grappes (10 PCs) et utilisation opérationnelle sur cluster de plus grande taille (200 PCs). Les grappes de PCs sont donc de plus en plus utilisées, mais l'enjeu scientifique important est de réussir à coupler toutes les compétences afin d'arriver à exploiter efficacement cette puissance de calcul dans des scénarios de type réalité virtuelle combinant la simulation d'objets 3D (fluide, végétation, etc ..), la capture de mouvements avec habillage du mannequin virtuel en temps réel (le mouvement du tissu suivant celui du personnage), l'immersion dans un centre de réalité virtuelle ou dans un système multi-écrans, avec des interactions possibles de l'utilisateur grâce aux interfaces haptiques. L'objectif scientifique réside en effet dans l'obtention d'une méthode d'intégration de programmes parallèles de simulation 3D et de rendu, en optimisant les performances de manière à rendre possible une animation en temps réel.

Au sein de ce chapitre, nous présentons l'application Sappe [86] qui nécessite l'intégration d'un code parallèle de simulation de textile dans un environnement immersif de réalité virtuelle appelé FlowVR [5] qui permet d'effectuer la visualisation de cette simulation sur plusieurs écrans.

10.2 Application Sappe

L'application Sappe [86] est en fait décomposée en deux modules dont la description du couplage est l'objet de cette section.

- Le premier module concerne la partie simulation numérique de textile. Ses calculs ont été parallélisés [86] par le langage ATHAPASCAN pour permettre son exécution sur une grappe de PCs.
- Le second module concerne la partie visualisation en utilisant la bibliotheque réalité virtuelle VR Juggler [27, 15] au sein d'environnement FlowVR [5].

10.2.1 La simulation parallèle de textile

La modélisation de textiles peut être abordée de plusieurs manières selon le but recherché. Lors de la création d'images de synthèses pour les jeux vidéos, l'objectif est de produire des images réalistes et convaincantes. Dans ce cas, les contraintes physiques sont ignorées ou du moins considérablement simplifiées. En effet, les infographistes préfèrent souvent simplifier le calcul d'une image en simulant « à la main » un mouvement plausible plutôt que de calculer le mouvement réel. En informatique graphique, les recherches se sont plutôt portées sur des simulations basées sur des modèles physiques permettant de restituer un comportement approché du tissu.

La technique usuelle pour effectuer le calcul de l'évolution d'un objet 3D, consiste à le discrétiser en particules connectées entre elles par des ressorts, formant ainsi un maillage géométrique soumis à des contraintes internes de voisinage. Cet objet peut ensuite être soumis à des forces extérieures (pour un vêtement il peut s'agir de la gravité ou du vent), et interagir avec d'autres objets (tissu en contact avec une table, collision) entraînant des modifications au sein même du maillage.

La simulation parallèle de cet objet peut consister en la décomposition de son espace de simulation ou en la décomposition de lui même en objets de plus petite taille. A ces objets sont ensuite associées des tâches de calculs. Celles-ci seront ordonnancées sur les processeurs en tenant compte de leurs contraintes de précédence générées par la décomposition de l'objet. Cet ordonnancement suit des critères de performances et de régulation de charge afin d'exploiter au mieux les ressources disponibles en terme de puissance de calcul et d'accès aux données ou aux I/O.

10.2.2 L'interaction en temps réel avec l'utilisateur

L'environnement VR Juggler [27, 15] est une bibliothèque de réalité virtuelle permettant une interaction en temps réel avec l'utilisateur. Il permet d'une part de capturer facilement des évènements extérieurs tels qu'un clic de souris, la position de la souris, des évènements claviers et d'autre part de visualiser les images 3D en temps réel.

Les dispositifs d'affichage employés en réalité virtuelle incluent aussi bien le traditionnel écran d'ordinateur, que les lunettes de réalité virtuelle (Head Mounted Displays, HMDs) ou encore les écrans de projection immersifs. Il est également possible d'employer non pas un seul écran plat de projection mais plusieurs afin d'obtenir un champ de vision plus important et permettant une immersion complète pour l'utilisateur dans le monde virtuel.

FlowVR [5] est une couche logicielle permettant d'utiliser une grappe de machines, dont chacun des noeuds supporte VR Juggler, comme une unique machine VR Juggler. En d'autres termes d'un point de vue utilisateur, il n'y a aucune différence entre exécuter une application VR Juggler sur une grappe, un unique PC ou une SGI Onyx. La suite présente la phase d'intégration de la simulation parallèle dans un environnement de réalité virtuelle immersif.

10.2.3 Le schéma de communication de données

Dans le cas où le souhait est de permettre à l'utilisateur de modifier la position d'une particule donnée du tissu en bougeant la souris, il suffit de récupérer la position de celle-ci et de la renvoyer au programme de simulation qui en prendra compte lors des calculs des pas de temps suivants. A la fin

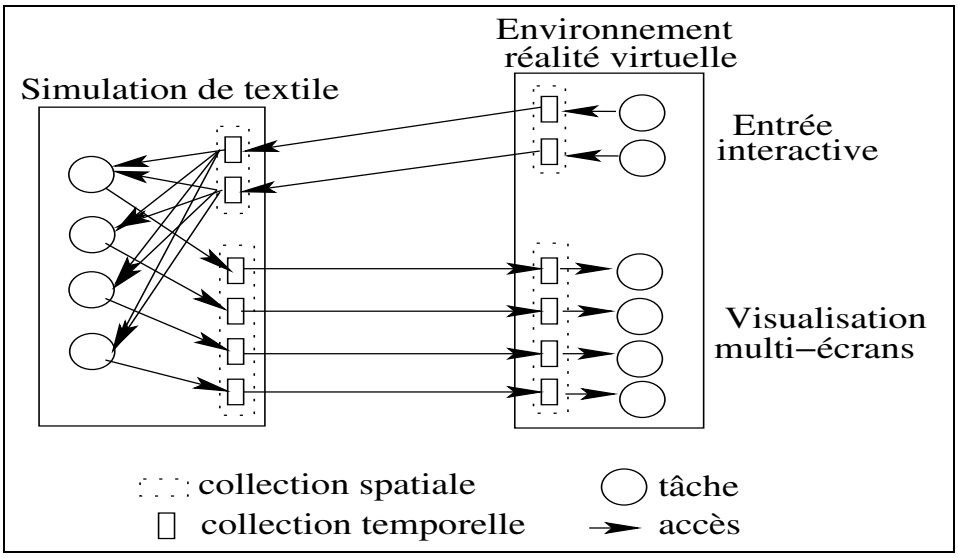


Figure 10.1 *Une représentation schématique du couplage pour l'application Sappe.*

de chaque itération, les nouvelles positions sont communiquées entre le programme de simulation et celui de la visualisation.

[86] a observé que pour 100 000 particules codées en flottants, les données transférées pour chaque itération représentent un volume de 1.2Mo. En utilisant un réseau ayant un débit de un Gigabit par seconde soit 125Mo par seconde, le temps réel en ce qui concerne le transfert est assuré car il est alors possible de transférer 5Mo toutes les 0.04 secondes. Mais si la simulation implique un million de particules, le transfert pour une itération représentant alors un volume de données de 12Mo, une seule connexion assurant le transfert de 5Mo est alors insuffisant pour du temps réel. Elle représente même un goulot d'étranglement pour l'application : il devient alors nécessaire d'effectuer plusieurs connexions entre les deux parties de l'application.

L'enjeu est de contrôler les interactions entre la simulation et la visualisation. Il y a notamment des compromis à effectuer entre la fréquence d'affichage et le pas de temps de calcul avec par exemple l'emploi d'un pas de temps adaptatif pour la simulation numérique ou encore la gestion d'une cohérence seulement locale et non globale au niveau de la visualisation. Il est également possible d'imaginer que dans le cadre de grosse scène, la visualisation soit plus lente que la simulation. Il serait alors intéressant d'augmenter la précision des calculs diminuant ainsi leur vitesse d'exécution.

10.2.3.1 Description distribuée du couplage de codes parallèles

Le parallélisme du code de simulation numérique est fait en découpant l'espace d'objet en plusieurs sous-objets ; chacun représente un ensemble de particules. A ces objets sont ensuite associés des données partagées et des tâches de calcul. Ces données partagées sont ensuite transférées sur les canaux de communication associées par des estampille temporelles. Les canaux sont de type dernier arrivé (« Last-Recieved »). Afin de garantir la cohérence des images, il est strictement nécessaire que les dernières valeurs lues sur l'ensemble des canaux soient de même estampille, cela est réalisé au niveau de FlowVR utilisé dans cette application.

Nous décrivons cette application de couplage de codes avec plusieurs **collections spatiales** d'objets partagés de type **collection temporelle**. La figure 10.1 présente le schéma du couplage entre le code parallèle de simulation textile et l'environnement réalité virtuelle (FlowVR). Il y a deux groupes de collections temporelles. Le premier correspond aux canaux de communication entre la partie «entrée» de l'environnement RV et la simulation. Les communications entre la simulation et la partie «visualisation» sont décrites par le deuxième groupe. Chaque groupe est une distribution spatiale de canaux que nous décrivons par des collections spatiales.

10.3 Expérience élémentaire

L'expérience est à pour but de mesurer la performance d'une utilisation des collections temporelles sur les deux opérateurs élémentaires de production et de consommation des valeurs. Les programmes ont été exécutés sur le I-Cluster2 de INRIA à Grenoble (104 noeuds, Itanium bi-processeurs 64 bits 900MHz, 3GByte, réseau fast-ethernet 100MBit/s).

10.3.1 Evaluation sur un canal de communication

L'expérience est basée sur un couplage de deux codes parallèles par un canal de communication. Nous avons considéré le type FIFO de canal. Un code (producteur) met les données dans le canal et l'autre code (consommateur) les retire de canal. Les codes sont écrits dans le langage ATHAPASCAN.

Dans un schéma itératif, le producteur crée une tâche de calcul pour écrire une nouvelle valeur d'une donnée partagée puis une autre tâche pour envoyer la valeur produite sur le canal. De même pour le consommateur, il prend les données de canal et puis les consume.

Le couplage est réalisé une fois par utilisation directe du canal dans les tâches de communication et l'autre fois par accès à une collection temporelle. Les figures 10.2 et 10.3 montrent les graphes de flot de données associés à ces codes. Nous voyons deux différences entre les deux graphes (sans et avec collection temporelle). La première est la synchronisation entre les tâches. Dans le cas sans collection temporelle, pour éviter l'accès concurrent au canal et respecter l'ordre séquentiel d'entrée dans canal, il faut ajouter une dépendance inutile entre les tâches de communication. En plus ces tâches sont bloquantes. Ceci peut augmenter le nombre de changement de contexte système et induire un surcoût plus important.

L'évaluation des performances est faite sur un point : le temps d'exécution pour 100 itérations. L'expérience est faite pour deux tailles de message 10 et 100K octets. La figure 10.4 présente le temps d'exécution par rapport au temps de calcul. Les gains (entre 10 à 20 pourcent) observés sont dus, d'une part, au recouvrement des communication par du calcul, et, d'autre part, aux algorithmes d'aggrégation des données mieux exploité avec les collections temporelles et leur implantations à base de messages actifs comme utilisé dans le support exécutif KAAPI.

10.4 Bilan

Sappe est une application réelle qui nécessite le couplage de deux codes parallèles par un ensemble de canaux de communication. Ce type de couplage est décrit par un ensemble de processus

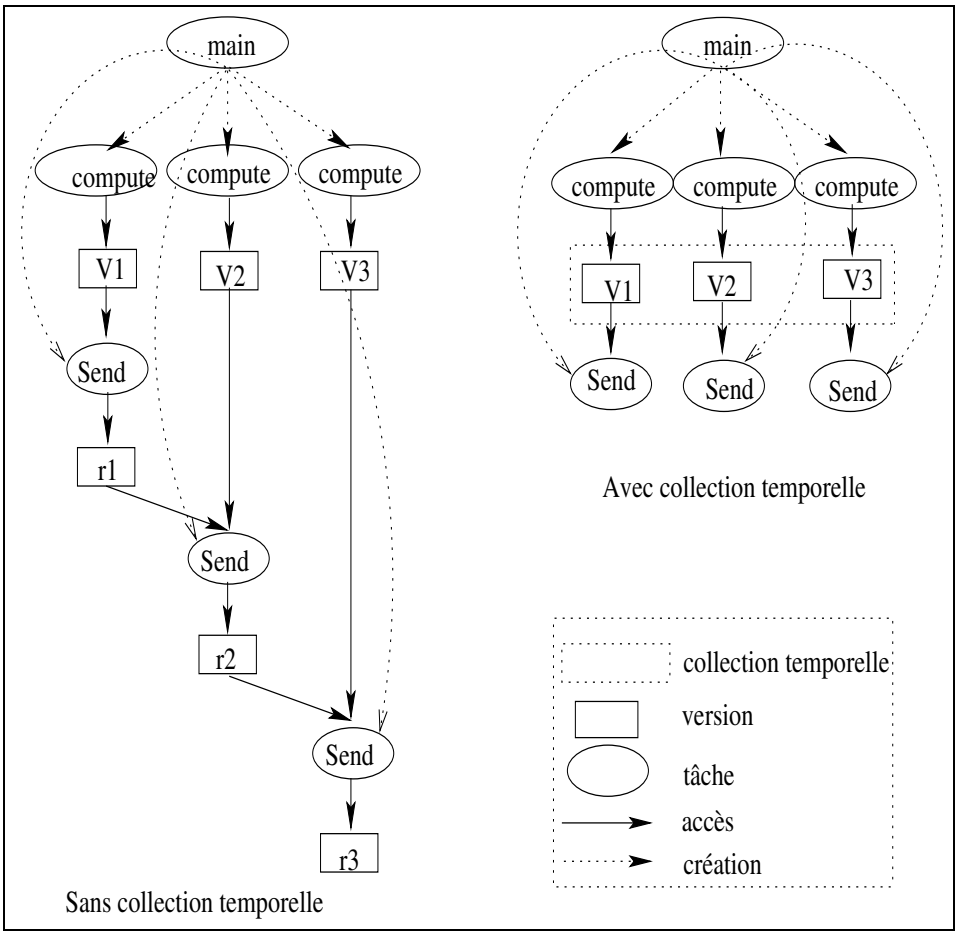


Figure 10.2 ***Les graphes de flot de données du côté des producteurs.*** *L'intérêt de ces graphes est de montrer comment utiliser les collections temporelles pour éviter des synchronisations inutiles. Par exemple, à gauche les tâches « Send » sont explicitement synchronisées par accès à une donnée partagée. Ceci est nécessaire pour respecter, à la lecture, leurs ordres d'écriture dans le canal. Avec la collection temporelle l'ordre d'envoi effectif sur le canal n'est pas nécessaire d'être compatible avec l'ordre d'écriture effectif.*

concurrents, nous l'appelons « description distribuée » du couplage de codes. Nous avons décrit ce type de couplage par une **collection spatiale** d'objets partagés de type **collection temporelle**.

Bien que l'expérience effectuée soit simple, elle montre l'avantage à utiliser une collection temporelle par rapport une utilisation directe d'opérateurs de communication sur un canal (envoi/réception).

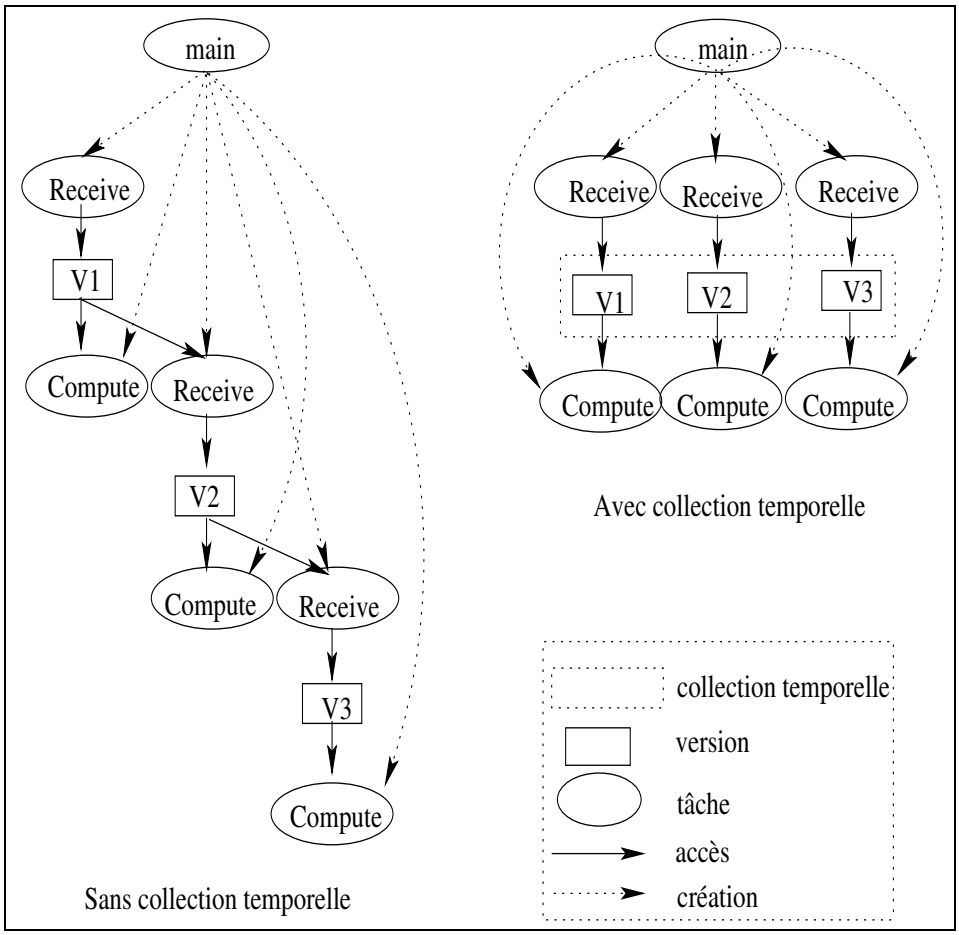


Figure 10.3 ***Les graphes de flot de données du côté des consommateurs pour les deux expériences.*** *L'intérêt de ces graphes est de montrer comment, en utilisant les collections temporelles, on peut éviter des synchronisations inutiles. Par exemple, à gauche les tâches « Receive » sont explicitement synchronisées par accès à une donnée partagée. Ceci est nécessaire pour recevoir les données tout en respectant leurs ordres d'écriture. Une collection temporelle permet de recevoir les données de manière arbitraire tout en respectant une stratégie d'entrelacement de type FIFO.*

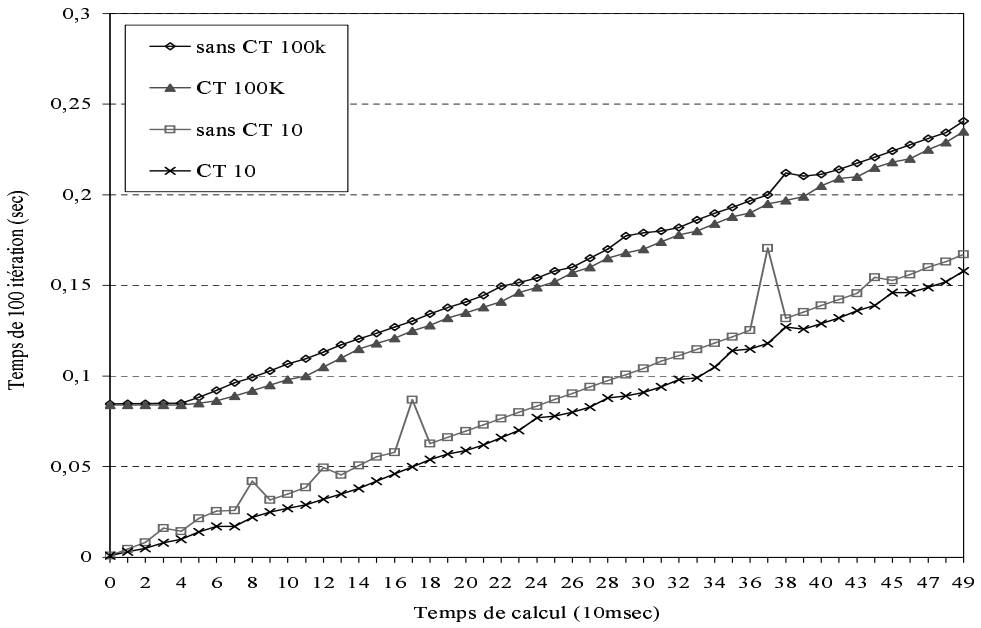


Figure 10.4 ***Evaluation de l'utilisation d'une collection temporelle sur un canal de communication.***

V

Conclusion

11

Conclusions et perspectives

Dans cette thèse, nous avons présenté certains des principaux problèmes dû au couplage de codes dans l'optique de la conception d'applications distribuées à hautes performances. Les problèmes liés au couplage des modèles physiques et à la conception des algorithmes numériques de couplage, bien que fondamentaux, ne sont pas étudiés dans ce manuscrit. Nous nous sommes intéressés à présenter les défis de deux modèles de programmation - « RPC » et « mémoire partagée » - pour un couplage à hautes performances de codes. Puis nous avons proposé certaines extensions et leurs mises en œuvre informatique.

Extensions au modèle RPC. Dans le cadre du projet HOMA, les extensions au modèle RPC ont porté, d'une part, sur la sémantique de contrôle et de communication et, d'autre part, sur les supports exécutifs pour mieux exploiter le parallélisme potentiel entre plusieurs invocations de méthodes. Les résultats théoriques de ces extensions pour une implantation sur le bus logiciel CORBA à l'aide du moteur exécutif KAAPI d'ATHAPASCAN et pour des architectures homogènes, comme des grappes de PC, sont présentés sous la forme d'un modèle de coût d'exécution. Les expériences (élémentaires et sur une application réelle) ont validé ce modèle de coût.

Extensions d'un modèle de type mémoire partagée. Afin d'étendre la consistance d'accès aux données partagées du langage ATHAPASCAN, nous avons proposé la notion de *collection temporelle*. Ce concept permet d'exploiter le parallélisme lié aux accès aux données temporellement estampillées. La *collection spatiale* permet de mieux exploiter le parallélisme d'accès à un ensemble de données partagées. Pour préciser la sémantique associée à ces nouvelles notions, nous avons donné une nouvelle définition pour la donnée partagée. Puis dans le cadre de cette notion, nous avons défini trois types de données partagées : « séquentielle », « collection temporelle » et « collection spatiale ».

Perspectives. Nous avons aussi présenté la notion de composant séquentiel et parallèle ainsi que leur composition en utilisant un modèle ou un langage de coordination. Les outils de base existent (déploiement, configuration), ou sont en cours de développement (composant parallèle). Mais peu

d'outils ou d'environnements intègrent les aspects liés à la coordination et à l'ordonnancement. Les premières solutions viendront peut être des environnements de metacomputing, des PSEs ou des ASPs qui abordent verticalement ces problèmes allant de la description de la composition des composants à leur mise en œuvre effective sur grille, et dont le besoin est important dans de nombreux domaines liés aux métiers de l'ingénieur.

Pour finir, nous nous essayerons au difficile jeu de déterminer quelques défis à résoudre pour l'avenir. Le premier concerne l'intégration d'un langage pour la coordination d'activités et la définition des composants parallèles. Il s'agit de permettre de décrire à la fois les aspects de *coordination* et les aspects liés à la distribution de données dans un modèle possédant une sémantique bien définie. Le second est la réalisation des communications à hautes performances entre plusieurs codes parallèles qui travaillent sur des structures de données distribuées irrégulièrement sur une grille.

Bibliographie

[1] Folding@home, distributed computing. http ://folding.stanford.edu/.

[2] Seti@home : The search for extraterrestrial intellignece. http ://setiathome.berkeley.edu/.

[3] IEEE Std 1516-2000, editor. *IEEE standard for modeling and simulation high level architecture (HLA) - framework and rules*. IEEE, 2000.

[4] C. Pèrez A. Denis and T. Priol. PadicoTM : An open integration framework for communication middleware and runtimes. In *Future Generation Computer Systems, 19(4) :575–585*, May 2003.

[5] J. Allard, V. Gouranton, L. Lecointre, S. Limet, E. Melin, B. Raffin, and S. Robert. Flowvr : a middleware for large scale virtual reality applications. In *Proceedings of Euro-par 2004*, Pisa, Italia, August 2004.

[6] J. Allard, B. Raffin, and F. Zara. Coupling parallel simulation and multi-display visualization on a pc cluster. In *Euro-par 2003*, Klagenfurt, Austria, August 2003.

[7] G. Alleon. The JACO3 project. In *Soft-IT Workshop Next Generation of interoperable simulation environnements based on CORBA*. INRIA-ONERA-Simulog, June 1999.

[8] G.R. Andrews and F.B. Schneider. Concepts and notations for concurrent programming. *ACM Comput. Surv.*, 15(1) :3–43, 1983.

[9] F. Arbab, I. Herman, and P. Spilling. An overview of Manifold and its implementation. *Concurrency : Practice and Experience*, 5(1) :23–70, 1993.

[10] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. Mc Innes, S. Parker, and B. Smolinski. Toward a common component architecture for high-performance scientific computing. In *Proc. of the 8th IEEE Inter. Symp. on High Performance Distributed Computation*, 1999.

[11] P.H. Beckman, P.K. Fasel, W.F. Humphrey, and S.M. Mniszewski. Efficient coupling of parallel applications using paws. Technical report, Los Alamos National Laboratory, USA, 1998.

[12] J.P. Belaud, B. Braunschweig, and M. Pons. Open software architecture for process simulation : The current status of cape-open standard. In *, European Symposium on Computer Aided Process Engineering*, 2002.

[13] P.-E. Bernard and O. Coulaud. Parallel constrained molecular dynamics. *INRIA, Research report RR-3868*, Janvier 2000.

[14] P.-E. Bernard, T. Gautier, and D. Trystram. Large scale simulation of parallel molecular dynamics. In *Proceedings of Second Merged Symposium IPPS/SPDP 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, San Juan, Puerto Rico, April 1999.

[15] A. Bierbaum, C. Just, P. Hartling, K. Meinert, A. Baker, and C. Cruz-Neira. Vr juggler : A virtual platform for virtual reality application development. *IEEE VR 2001*, March 2001.

[16] G.E. Blelloch. NESL : A Nested Data-Parallel Language. Technical Report CMU-CS-93-129, April 1993.

[17] G.E. Blelloch, P.B. Gibbons, and Matias Y. Provably efficient scheduling for languages with fine-grained parallelism. In *Proc. 7th Annual ACM Symposium on Parallel Algorithms and Architectures SPAA'95*, pages 1–12, Santa Barbara, California, 1995.

[18] R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, and Y.Zhou. Cilk : An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1) :55–69, 1996.

[19] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, and W.K. Su. Myrinet : A gigabit-per-second local area network. *IEEE Micro*, 15(1) :29–36, 1995.

[20] D. Caromel. Towards a method of object-oriented concurrent programming. *Communication of the ACM*, 36 :90–102, 1993.

[21] E. Caron, F. Desprez, E. Fleury, F. Lombard, J.-M. Nicod, M. Quinson, and F. Suter. *Calcul réparti à grande échelle*, chapter Une approche hiérarchique des serveurs de calculs. Hermès Science Paris, 2002. ISBN 2-7462-0472-X.

[22] 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.

[23] E. Caron and F. Suter. Parallel Extension of a Dynamic Performance Forecasting Tool. In *Proceedings of the International Symposium on Parallel and Distributed Computing*, pages 80–93, Iasi, Romania, Jul 2002.

[24] N. Carriero and D Gelernter. Coordination languages and their significance. *Communication of the ACM*, 35(2) :97–107, 1992.

[25] H. Casanova and J. Dongarra. NetSolve : A network server for solving computational science problems. In *Workshop of Vector and Parallel computing*, Manno, Switzerland, March 1997. SPEEDUP Society.

[26] Microsoft Corporation. The component object model specification, version 0.9. Technical report, <URL : http ://www.microsoft.com/Com/resources/comdocs.asp>, October 1995.

[27] C. Cruz-Neira, A. Bierbaum, P. Hartling, C. Just, and K. Meinert. Vr juggler - an open source platform for virtual reality applications. *40th AIAA Aerospace Sciences Meeting and Exhibit 2002*, January 2002.

[28] F. Darema, D.A. George, V.A. Norton, and G.F. Pfister. A single-program-multiple-data computational model for epex/fortran. *Parallel Computing*, 7(1) :11–24, 1988.

[29] F. Darema, D.A. George, V.A. Norton, and G.F. Pfister. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4) :321–359, 1989.

[30] A. Denis, C. Perez, and T. Priol. Portable parallel corba objects : An approach to combine parallel and distributed programming for grid computing. In *Euro-Par '01 : Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*, pages 835–844, London, UK, 2001. Springer-Verlag.

[31] J. Dongarra and al. *MPI : A Message-Passing Interface Standard.* University of Tennessee, Knoxville, Tennessee., May 1994. Available via http ://www.netlib.org/mpi/mpi-report.ps.

[32] M. Doreille. *Athapascan 1 : vers un modèle de programmation parallèle adapté au calcul scientifique*. Thèse de doctorat en mathématiques appliquées, Institut National Polytechnique de Grenoble, France, December 1999.

[33] C.A. Dorian, B. Dieter, and J. Dongarra. Request sequencing : Optimizing communication for the grid. In *EuroPar – Parallel Processing*, 2000.

[34] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 114–118. ACM Press, 1978.

[35] I. Foster and C. Kesselman. *The Grid : Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Inc., 1998.

[36] I. Foster, C. Kesselman, R. Olson, and S. Tuecke. Nexus : An Interoperability Layer for Parallel and Distributed Computer Systems. Technical Report ANL/MCS-TM-189, Argonne National Laboratory, USA, 1994.

[37] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid. *Intl J. Supercomputer Applications*, 2001.

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

[39] M. Frigo. The weakest reasonable memory. Master's thesis, 1998.

[40] M. Frigo and V. Luchangco. Computation-centric memory models. In *SPAA '98 : Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, pages 240–249, New York, NY, USA, 1998. ACM Press.

[41] Previously from AT & T Laboratories Cambridge. *OmniORB3 Programming Guide*, June 2002.

[42] F. Galilée. *Athapascan-1 : interprétation distribuée du flot de données d'un programme parallèle*. PhD thesis, Institut National Polytechnique de Grenoble, France, sep 1999.

[43] F. Galilée, J.L. Roch, G. Cavalheiro, and M. Doreille. Athapascan-1 : On-line building data flow graph in a parallel language. In IEEE, editor, *Pact'98*, pages 88–95, Paris, France, October 1998.

[44] T. Gautier. Kaapi : Kernel for adaptative, asynchronous parallel and interactive programming. http ://moais.imag.fr/kaapi.

[45] T. Gautier, O. Coulaud, and H.R. Hamidi. *Informatique répartie*, chapter Couplage de codes parallèles. Hermès Science Paris, 2005. ISBN 2-7462-08571.

[46] T. Gautier and H.R. Hamidi. Homa : automatic re-scheduling of multiple invocations in corba. *INRIA, Research report*, 2004.

[47] T. Gautier and H.R. Hamidi. Automatic re-scheduling of dependencies in a rpc-based grid. In *Proceedings of 18th annual ACM International Conference on Supercomputing (ICS'04), pages 89–94*, Saint-Malo France, June 2004. ACM Press.

[48] T. Gautier and H.R. Hamidi. High performance composition of services with data dependencies on a computational grid. In *In Proceedings of the International Conference on Parellel and Distributed Processing Techniques and Applications (PDPTA'04), pages 809–814*. CSREA Press, June 2004.

[49] T. Gautier and H.R. Hamidi. Homa : un compilateur idl optimisant les communications des données pour la composition d'invocations de méthodes corba. In *RenPar'2003*, France, Octobre 2003.

[50] T. Gautier and H.R. Hamidi. Re-scheduling invocations of services on rpc-based grid. In *In International Journal Computer Languages, Systems and Structures (CLSS), Special Issue on SEMANTICS AND COST MODELS FOR HIGH-LEVEL PARALLEL PROGRAMMING*. Elsevier press, to appear mid 2005.

[51] T. Gautier, R. Revire, and J.L. Roch. Athapascan : Api for asynchronous parallel programming. Technical Report RR-0276, INRIA Rhône-Alpes, projet APACHE, February 2003.

[52] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systemes*, 7(1) :80–112, 1985.

[53] C. Germain, V. Neri, G. Fedak, and F. Cappello. Xtremweb : Building an experimental platform for global computing. In *GRID*, pages 91–101, 2000.

[54] M. G. Hackenberg, P. Post, R. Redler, and B. Steckel. Mpcci, multidisciplinary applications and multigrid. In *ECCOMAS 2000*, CIMNE Barcelona, 2000.

[55] High Performance Fortran Forum. *High Performance Fortran Language Specification*. May 1993. version 1.0 edition.

[56] R.M. Karp, M. Luby, and F. Meyer auf der Heide. Efficient PRAM simulation on a distributed memory machine. pages 318–326, 1992.

[57] K. Keahey and D Gannon. PARDIS : a parallel approach to CORBA. In *Proceedings of the ACM/IEE International Conference of Supercomputing*, 1997.

[58] K. Keahey and D. Gannon. Pardis : Corba-based architecture for application-level parallel distributed computation. In *SC97 (Supercomputing '97)*, November 1997.

[59] Lamport L. How to make a multiprocessor that correctly executes multiprocess programs. In *IEEE Transactions on Computers*, volume C-28, pages 690–691, September 1979.

[60] I. Lopez, G.J. Follen, R. Gutierrez, I. Foster, B. Ginsburg, O. Larsson, and S. Tuecke. Using corba and globus to coordinate multidisciplinary aeroscience applications. In *Proceedings of the NASA HPCC/CAS Workshop*, pages 15–17, Feb. 2000.

[61] S.S. Lumetta, A.M. Mainwaring, and D.E. Culler. Multi-protocol active messages on a cluster of smp's. In *Supercomputing '97 : Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–22, New York, NY, USA, 1997. ACM Press.

[62] R. Monson-Haefel. *Enterprise JavaBeans*. O'Reilly, 1999.

[63] F. Mueller. A library implementation of POSIX threads under UNIX. In *Proc. of the Winter USENIX Conference*, pages 29–41, San Diego, CA, January 1993.

[64] T. Nguyen and C. Plumejeaud. *Calcul réparti à grande échelle*, chapter Intégration d'applications multidisciplines dans les environnements de metacomputing. Hermès Science Paris, 2002. ISBN 2-7462-0472-X.

[65] S. Norris, B.and Balay, S. Benson, L. Freitag, P. Hovland, L. McInnes, and B. Smith. Parallel components for pdes and optimization : some issues and experiences. In *Parallel Computing, volume 28(12), 1811-1831, Elsevier Science Publishers*, 2002.

[66] CORPORATE Institute of Electrical and Inc. Staff Electronics Engineers. *IEEE Standard for Scalable Coherent Interface, Science : IEEE Std. 1596-1992*. IEEE Standards Office, New York, NY, USA, 1993.

[67] OMG. Common object request broker architecture (corba/iiop). Technical report, OMG formal/2002-12-06, 2002.

[68] OMG. Corba component model. Technical report, OMG, formal/2002-06-65, 2002.

[69] OMG. Data parallel object. Technical report, OMG formal/2002-06-65, 2002.

[70] G. Papadopoulos and F. Arbab. Coordination models and languages. In Marvin V. Zelkowitz, editor, *Advances in Computers 46*, pages 329–400. Academic Press, 1998.

[71] C. Pelts. *Web services orchestration : a review of emerging technologies, tools, and standards*. Hewlett-Packard, Co., January 2003.

[72] C. Pérez, T. Priol, and A. Ribes. A parallel corba component model for numerical code coupling. In Craig A. Lee, editor, *Proc. of the 3nd International Workshop on Grid Computing*, LNCS, Baltimore, Maryland, USA, November 2002. Springer-Verlag.

[73] C. Pérez, T. Priol, and A. Ribes. Paco++ : A parallel object model for high performance distributed systems. In *HICSS '04 : Proceedings of the Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04) - Track 9*, page 90274.1, Washington, DC, USA, 2004. IEEE Computer Society.

[74] J. Protic, M. Tomasevic, and V. Milutinovi, editors. *Distributed Shared Memory : Concepts and Systems*. IEEE, 1997.

[75] U. Ramachandran, R.S. Nikhil, N. Harel, J.M. Rehg, and K. Knobe. Space-time memory : A parallel programming abstraction for interactive multimedia applications. In *Principles Practice of Parallel Programming*, pages 183–192, 1999.

[76] U. Ramachandran, R.S. Nikhil, N. Harel, J.M. Rehg, and K. Knobe. Stampede : A cluster programming middleware for interactive stream-oriented applications. In *IEEE Transactions on Parallel and Distributed Systems, vol. 14, no. 11*, pages 1140–1154, 2003.

[77] C. René and T. Priol. MPI code encapsulating using parallel CORBA object. *Cluster Computing*, 3(4) :255–263, 2000.

[78] A. Ribes. *Contribution à la conception d'un modèle de programmation parallèle et distribué et sa mise en oeuvre au sein de plates-formes orientées objet et composant.* Thèse de doctorat en informatique, École Doctorale MATISSE (IFSIC/IRISA), France, 2004.

[79] M. Sato S. Sekiguchi S. Matsuoka, H. Nakada. Design issues of Network Enabled Server Systems for the Grid. In *Grid Computing – GRID 2000*, volume 1971 of *Lecture Notes in Computer Science*, pages 4–17. Springer-Verlag, 2000.

[80] M. Sato, H. Nakada, S. Sekiguchi, S. Matsuoka, U. Nagashima, and H. Takagi. Ninf : A network based information library for a global world-wide computing infrastracture. In *HPCN'97 (LNCS-1225)*, pages 491–502, 1997.

[81] D. Schmidt, A. Gokhale, T. Harrison, D. Levine, and C. Cleeland. Tao : A high-performance endsystem architecture for real-time corba, 1997.

[82] M. Sloman. Policy driven management for distributed systems. *Journal of Network and Systems Management*, 2 :333–360, 1994.

[83] C. Szyperski. *Component Oriented Programming*. Addison Wesley, 1997.

[84] Athapascan-1 team. *Athapascan-1*. Projet APACHE, Grenoble.

[85] R. Wolski, N. T. Spring, and J. Hayes. The network weather service : A distributed resource performance forecasting service for metacomputing. In *Future Generation Computing Systems, Metacomputing Issue*, volume 15, pages 757–768, October 1999.

[86] F. Zara. *Algorithmes parallèles de simulation physique pour la synthèse d'images : application à l'animation de textiles*. PhD thesis, Institut National Polytechnique de Grenoble, France, dec 2003.

Couplage à hautes performances de codes parallèles et distribués

Résume

L'accroissement rapide de la puissance des calculateurs actuels et leur interconnexion en grappes et grilles de calcul à l'aide de réseaux rapides, permettent d'envisager, en mode de production, l'utilisation de plusieurs codes de calculs numériques couplés pour la simulation de phénomènes physiques plus complexes. Dans le but d'obtenir des résultats toujours plus précis, un nouveau type de simulation numérique, dont l'objectif est de simuler plusieurs physiques en même temps, est apparu. Ce type d'application est appelé "couplage de code". En effet, plusieurs codes (physiques) sont couplés ou interconnectés afin qu'ils communiquent pour réaliser la simulation.

Cette thèse s'intéresse aux problématiques liées au couplage à hautes performances de codes parallèles et distribués. L'obtention des performances repose sur la conception d'applications distribuées dont certains composants sont parallélisés et dont les communications sont efficaces. L'idée de bas de cette thèse est d'utiliser un langage de programmation parallèle orienté flot de données (ici Athapascan) dans deux modèles de conception d'applications distribuées ; "modèle appel de procédure à distance (RPC)" et "modèle orienté flux de données (stream-oriented)". Les contributions apportées par ce travail de recherche sont les suivants :

- **Utilisation d'un langage de flot de données dans un grille RPC de calcul ;**
  Dans le cadre de projet HOMA, les extensions au modèle RPC ont porté d'une part sur la sémantique de contrôle et de communication et d'autre part sur les supports exécutifs pour mieux exploiter le parallélisme. Les résultats théoriques de ces extensions pour une implantation sur le bus logiciel CORBA à l'aide du moteur exécutif KAAPI d'Athapascan et pour l'architecture homogène comme grappe de PC, sont présentés sous la forme d'un modèle de coût d'exécution. Les expériences (élémentaires et sur une application réelle) ont validé ce modèle de coût.
- **Extension d'un modèle mémoire partagée pour couplage de codes ;**
  Afin d'étendre la sémantique d'accès aux données partagées du langage Athapascan, nous avons proposé la notion de "collection temporelle". Ce concept permet de décrire la sémantique d'accès de type flux de données. La "collection spatiale" permet de mieux exploiter les données parallèles. Pour préciser la sémantique associée à ces nouvelles notions, nous avons donné une nouvelle définition pour la donnée partagée. Puis dans le cadre de cette définition, nous avons défini trois types de données partagées ; "séquentielle", "collection temporelle" et "collection spatiale".

High performance coupling of parallel and distributed codes

Abstract

The quick power's growth of current computers and their interconnection in clusters and computational grids using high speed networks, allow to simulate more complex physical phenomena coupling several numerical calculation codes. In the hope to obtain more precise results, a new type of numerical simulation appeared which simulate several physics at the same time. This type of application is called "codes coupling". In fact, several codes (physics) are coupled or interconnected so that they communicate to achieve the simulation.

This thesis is interested to the problematic of high performance coupling of parallel and distributed codes. Performance achievement is based on the conception of distributed applications where certain components are parallel and communicate efficiently. The basic idea of this thesis is to construct distributed application using a data-flow oriented parallel programming language (Athapascan). Our contribution is situated in two programming models ; "remote procedure call (RPC) model" and "stream-oriented model". The contributions brought by this research are the following ones :

- **Utilisation of a data-flow oriented language on a RPC-based computational grid ;**
  In HOMA project, the extensions to RPC model carried on one hand on the control and communication semantic and on the other hand on the execution supports to better exploit the parallelism. In the homogenous architecture such as clusters, the theoretical results of these extensions are presented in the form of an execution cost model for an implementation on CORBA distributed system using KAAPI ; the execution kernel of Athapascan parallel language. The experiences (elementary and on a real application) validated this cost model.
- **Extensions to a shared memory model for codes coupling ;**
  In order to extend the consistency model of Athapascan virtual shared memory, we proposed "time collection". This notation allows to describe the stream-like semantics of access to shared data. Spatial distribution of shared data, such as needed for "data parallel" codes, is described by "space collection". To specify the semantic associated to these new notations, we gave a new definition for shared data. Then in the framework of this definition, we defined three types of shared data ; "sequential", "time collection" and "space collection".

Couplage à hautes performances de codes parallèles et distribués

Cette thèse s'intéresse aux problématiques liées au couplage à hautes performances de codes parallèles et distribués. L'idée de bas de cette thèse est d'utiliser un langage de programmation parallèle orienté flot de données (ici Athapascan) dans deux modèles de conception d'applications distribuées ; "modèle appel de procédure à distance (RPC)" et "modèle orienté flux de données (stream-oriented)". Les contributions apportées par ce travail de recherche sont les suivants :

- **Utilisation d'un langage de flot de données dans un grille RPC de calcul ;** Dans le cadre de projet HOMA, les extensions au modèle RPC ont porté d'une part sur la sémantique de contrôle et de communication et d'autre part sur les supports exécutifs pour mieux exploiter le parallélisme. Les résultats théoriques de ces extensions pour une implantation sur le bus logiciel CORBA à l'aide du moteur exécutif KAAPI d'Athapascan et pour l'architecture homogène comme grappe de PC, sont présentés sous la forme d'un modèle de coût d'exécution.
- **Extension d'un modèle mémoire partagée pour couplage de codes ;** Afin d'étendre la sémantique d'accès aux données partagées du langage Athapascan, nous avons proposé la notion de "collection temporelle". Ce concept permet de décrire la sémantique d'accès de type flux de données. La "collection spatiale" permet de mieux exploiter les données parallèles. Pour préciser la sémantique associée à ces nouvelles notions, nous avons donné une nouvelle définition pour la donnée partagée. Puis dans le cadre de cette définition, nous avons défini trois types de données partagées ; "séquentielle", "collection temporelle" et "collection spatiale".

Mots-clés : Langage de flot de données, Grille RPC de calcul, Systems orientés flux de données.

High performance coupling of parallel and distributed codes

This thesis is interested to the problematic of high performance coupling of parallel and distributed codes. The basic idea of this thesis is to construct distributed application using a data-flow oriented parallel programming language (Athapascan) in two programming models of distributed applications ; "remote procedure call (RPC) model" and "stream-oriented model". The contributions brought by this research are the following ones :

- **Utilisation of a data-flow oriented language on a RPC-based computational grid ;** In HOMA project, the extensions to RPC model carried on one hand on the control and communication semantic and on the other hand on the execution supports to better exploit the parallelism. In the homogenous architecture such as clusters, the theoretical results of these extensions are presented in the form of an execution cost model for an implementation on CORBA distributed system using KAAPI ; the execution kernel of Athapascan parallel language.
- **Extensions to a shared memory model for codes coupling ;** In order to extend the consistency model of Athapascan virtual shared memory, we proposed "time collection". This notation allows to describe the stream-like semantics of access to shared data. Spatial distribution of shared data, such as needed for "data parallel" codes, is described by "space collection". To specify the semantic associated to these new notations, we gave a new definition for shared data. Then in the framework of this definition, we defined three types of shared data ; "sequential", "time collection" and "space collection".

Key-words : Data flow language, RPC-based computional Grids, Stream-oriented systems.