Modèles et outils pour le déploiement d'applications de Réalité Virtuelle sur des architectures distribuées

Sylvain Jubertie

Modèles et outils pour le déploiement d'applications de Réalité Virtuelle sur des architectures distribuées hétérogènes

Thèse

présentée à l'Université d'Orléans pour obtenir le grade de
Docteur de l'Université d'Orléans
discipline : Informatique
présentée et soutenue publiquement par

Sylvain Jubertie

le 14 décembre 2007

Membres du jury

Président :	M. Olivier Coulaud	Directeur de recherche INRIA-Futurs
Rapporteurs :	M. Bruno Arnaldi	Professeur INSA de Rennes
	M. Frédéric Desprez	Directeur de recherche INRIA-LIP ENS Lyon
Examinateurs :	M. Sébastien Limet	Professeur Université d'Orléans
	M. Bruno Raffin	Chargé de recherche INRIA Rhônes-Alpes
Directeur :	M. Gaétan Hains	Professeur LACL-Université Paris 12
Encadrant :	M. Emmanuel Melin	Maître de conférence Université d'Orléans

Remerciements

Je tiens à remercier Bruno Arnaldi et Frédéric Desprez pour avoir accepté de rapporter mon travail de thèse ainsi que les autres membres de mon jury, Olivier Coulaud, Bruno Raffin et Sébastien Limet. Je remercie également mon directeur de thèse, Gaétan Hains, et plus particulièrement Emmanuel Melin qui m'a encadré et conseillé pendant ces trois années de thèse.

Mon intérêt pour le parallélisme et la réalité virtuelle est apparu lors de ma licence suite à la présentation d'applications de réalité virtuelle développées par des étudiants de maîtrise effectuant leur stage au LIFO. Ces applications, basées sur l'environnement NetJuggler conçu au LIFO, exploitaient un grappe de PC pour fournir un calcul ainsi qu'un rendu distribué en stéréo. Impressionné par ces démonstrations et le mélange entre théorie et aspects système, j'ai donc continué mes études en espérant pouvoir un jour travailler dans ce domaine à la croisée de nombreuses disciplines. Je remercie donc les membres de l'équipe de parallélisme/réalité virtuelle du LIFO : Sophie Robert, Sébastien Limet et Emmanuel Melin, pour m'avoir proposé un stage de DEA puis un sujet de thèse, ainsi que les membres du LIFO et du département d'informatique qui sont trop nombreux pour que je les cite ici ! Merci également à mes collègues de bureau : Radia, pour les nombreuses discussions sur les modèles de coût que nous avons eu, et Jérémie, pour m'avoir apporté son aide sur la programmation par contraintes, et pour leur bonne humeur et humour ;-).

Cette thèse est l'aboutissement d'un long périple qui n'a pas été toujours évident. Je remercie particulièrement les membres de l'IUT d'Informatique d'Orléans pour m'avoir accueilli pendant l'année entre la fin de mon DEA et le début de ma thèse qui fut pour moi l'occasion de m'initier aux joies de l'enseignement.

Enfin, je ne serais pas arrivé jusqu'ici sans le soutien de ma maman et de mes grand-parents ainsi que de mes amis. Mes remerciements sont également adressés à Anne-Laure qui a parfois du subir mes rédactions nocturnes et qui me soutient toujours.

Table des matières

Chapitre 1

Introduction

Contexte

Réaliser une expérience dans le monde réel est une tâche soumise à de nombreuses contraintes physiques, d'espace, de temps, de coût, etc. De nombreuses expériences sont donc difficilement réalisables, voir irréalisables de part les moyens nécessaires à leur mise en oeuvre. La solution proposée par la réalité virtuelle est de produire une simulation de l'expérience que l'on souhaite réaliser par l'utilisation de moyens informatiques, et d'utiliser des interfaces homme-machine pour pouvoir interagir avec cette simulation. Une définition de la réalité virtuelle est ainsi donnée par Arnaldi dans le *Traité de la Réalité Virtuelle*[17] :

> *"La réalité virtuelle est un domaine scientifique et technique exploitant l'informatique et des interfaces comportementales en vue de simuler dans un monde virtuel le comportement d'entités 3D, qui sont en interaction en temps réel entre elles et avec un ou des utilisateurs en immersion pseudo-naturelle par l'intermédiaire de canaux sensori-moteurs."*

Nous pouvons dégager de cette définition les trois aspects nécessaires à la réalisation d'une expérience de réalité virtuelle. En premier lieu, nous avons besoin d'un **modèle** informatique pour décrire les entités et leur comportement dans le monde virtuel. Nous devons ensuite définir une **représentation** du monde virtuel afin de pouvoir l'appréhender. Finalement, une **interaction en temps réel** est nécessaire entre le modèle, sa représentation et l'utilisateur afin que celui-ci puisse se sentir immerger dans le monde virtuel. Nous décrivons maintenant plus en détail ces trois aspects.

Un modèle peut être expérimental, c'est à dire créé à partir d'observations réelles, ou bien il peut être imaginaire. Dans le premier cas on cherche à reproduire la réalité. On trouve par exemple dans cette catégorie les simulateurs de conduite, les visites virtuelles ou les simulations chirurgicales. Dans le second cas on dispose d'un modèle imaginaire que l'on souhaite expérimenter réellement. Cette approche est par exemple utilisée en astrophysique afin de tester des modèles cosmologiques purement théoriques. La Réalité Virtuelle permet donc de s'affranchir de l'espace, du temps et de la réalité ce qui ouvre de vastes possibilités pour de nombreuses applications. Il devient ainsi possible d'expérimenter des modèles sans avoir besoin de créer de représentations physiques réelles, de changer d'échelle afin d'observer des phénomènes microscopiques, de partager une simulation et de collaborer avec plusieurs personnes éventuellement distantes ou encore de reproduire des évènements passés ou simuler des évènements futurs. Les

implications sont donc importantes et intéressent de nombreux domaines, citons par exemple la formation, le prototypage, la visualisation scientifique, et l'art.

Un autre point important de la réalité virtuelle est la représentation qui est faite du modèle considéré. Cette représentation peut chercher à produire un environnement naturel le plus réaliste possible, ou à l'opposé utiliser une représentation symbolique. Le premier cas est par exemple très utilisé pour la formation par simulateurs car l'utilisateur doit s'entraîner dans les conditions les plus réalistes possible. Dans d'autres cas l'information que l'on souhaite représenter est symbolique et ne dispose pas de représentation. Il est donc nécessaire de la matérialiser afin que l'utilisateur puisse l'appréhender par ses sens. Il est ainsi possible d'utiliser des niveaux de couleurs afin de représenter et donc de pouvoir étudier la diffusion de chaleur dans un matériau. La réalité virtuelle permet donc de s'abstraire de la représentation qui est donnée à un modèle afin de pouvoir choisir celle qui a le plus de sens pour l'utilisateur.

La réalité virtuelle implique également une interaction entre l'utilisateur et le modèle considéré par l'intermédiaire des différents sens de l'utilisateur afin qu'il puisse agir sur l'environnement virtuel et recevoir une réaction en retour. L'interaction doit donc être crédible et cohérente. Ces critères sont subjectifs et dépendent de l'utilisateur, de l'application considérée ainsi que des périphériques utilisés. Prenons comme exemple une simulation d'écoulement de fluides. Dans un simulateur naval destiné à la formation cet écoulement doit se comporter comme dans la réalité, c'est à dire produire un résultat dont la différence avec le résultat réel ainsi que le laps de temps nécessaire pour sa production ne sont pas perceptibles par l'utilisateur. Dans ce cas l'interactivité dans le monde virtuel est calquée sur l'interactivité réelle du phénomène. A l'opposé, une telle simulation peut être utilisée pour observer un phénomène précis. La production des résultats peut être très lente et donc non interactive mais ce qui intéresse l'utilisateur c'est avant tout la manipulation de la représentation de l'écoulement de fluide.

Les possibilités offertes par les applications de réalité virtuelle sont étroitement liées aux capacités des matériels utilisés pour effectuer l'interaction, la visualisation et les simulations. Afin de produire des environnements virtuels toujours plus riches en information, il est nécessaire de manipuler toujours plus de données et donc de disposer d'ordinateurs plus puissants en terme de calcul et de stockage.

Problématique

Les architectures à mémoire distribuées de type grappes de PC représentent une solution intéressante pour la réalité virtuelle. D'une part les composants des PC sont de plus en plus performants, et leur production à grande échelle les rend peu coûteux. D'autre part les grappes de PC sont évolutives car les composants sont standards, et extensibles par simple connexion de nouveaux PC. Les avantages sont donc nombreux sur les architectures parallèles telles que les stations SGI historiquement utilisées pour la réalité virtuelle. Les grappes de PC actuelles possèdent différents niveaux de parallélisme : au niveau du noeud, plusieurs processeurs accèdent à une mémoire partagée (SMP, multi-coeurs) alors qu'au niveau de la grappe la mémoire est répartie et les noeuds collaborent par échange de messages à travers des réseaux haut-débit de type gigaEthernet ou Myrinet. De nombreux outils existent pour la programmation d'applications parallèles et distribuées afin de tirer parti de l'architecture de ces grappes.

Certains s'attachent à aider le programmeur à paralléliser ses codes, d'autres à faciliter leur distribution et leur couplage.

Afin d'exploiter au mieux les grappes de PC il faut néanmoins scinder les applications de réalité virtuelle en différentes tâches. Dans la réalité, les phénomènes de notre environnement se déroulent de manière simultanée et interagissent. Partant de cette analogie nous pouvons découper les applications de réalité virtuelle en un ensemble de tâches (simulations, interactions, visualisation, etc.) capables de s'exécuter de manière asynchrone sur différent processeurs et de communiquer entre elles via les bus et réseaux. Chaque tâche est également susceptible d'être parallélisée afin d'améliorer encore la performance. De part son aspect pluridisciplinaire, une application de réalité virtuelle résulte souvent du travail de développeurs issus de différents domaines. Ce découpage du code apporte donc également un sens au niveau du génie logiciel car il facilite sa conception et sa réutilisation.

Ces tâches doivent ensuite être placées sur les différents noeuds de la grappe. Si chacune d'elles est placée sur un noeud différent alors leur exécution est optimisée mais leurs communications passent par le réseau qui est le principal goulot d'étranglement de ces architectures et qui introduit également une latence supplémentaire. A l'opposé, plusieurs tâches placées sur un même noeud peuvent communiquer efficacement par la mémoire partagée mais cela implique un partage des ressources (processeurs, mémoire, entrées/sorties) qui peut ralentir leur exécution. Les performances dépendent donc de trois facteurs :

1. l'architecture, qui dispose d'une puissance de calcul et des capacités de communication ;
2. l'application dont les codes communiquent et se synchronisent, la performance d'un code peut donc dépendre de celle d'un autre code ;
3. le placement qui détermine la performance de chaque code sur les noeuds ainsi que celle des communications.

Le choix d'un bon placement, capable de fournir l'interactivité escomptée, est donc déterminant pour les performances de l'application. Le processus pour obtenir un tel placement consiste à élaborer un premier placement, le tester sur la grappe afin de vérifier l'interactivité et la stabilité des performances, puis le modifier et recommencer les tests s'il n'est pas satisfaisant. La phase de conception est donc longue et suppose une connaissance approfondie des caractéristiques de la grappe, une analyse précise de l'architecture de l'application et repose en grande partie sur l'expérience du développeur. Elle devient encore plus difficile lorsque le nombre de tâches dépasse le nombre de processeurs, il faut alors gérer la concurrence entre les tâches, ou lorsque la grappe considérée est hétérogène, c'est à dire qu'elle comporte des noeuds et des réseaux différents.

Un modèle capable d'évaluer la performance d'un placement à partir, d'une part d'une modélisation de l'architecture et d'autre part d'une modélisation de l'application, permettrait de supprimer la phase de test et d'accélérer ainsi le processus de conception de placements. Il deviendrait alors possible de comparer "a priori" les performances de plusieurs placements pour déterminer le meilleur mais également de guider le développeur dans ses choix à partir des informations de performance fournies par le modèle. Cette approche permettrait également d'abstraire la création de placements de l'architecture matérielle et de prévoir ainsi le comportement d'une application sur une architecture virtuelle afin, par exemple, de justifier l'ajout de noeuds à notre architecture pour atteindre le niveau de performance souhaité.

Dans un deuxième temps un tel modèle pourrait servir de base à un outil de génération et d'optimisation automatique de placements. Notons que le nombre de placements potentiels peut subir une explosion combinatoire lorsqu'on augmente le nombre de parties de l'application ainsi que le nombre de noeuds de la grappe. Il devient alors nécessaire d'envisager l'utilisation combinée de ce modèle de performance et de méthodes d'optimisation.

Organisation du mémoire

La première partie de ce mémoire présente les approches utilisées pour la construction d'applications de réalité virtuelle, puis les différents types d'architectures parallèles et distribuées, les outils pour la programmation sur ces architectures, ainsi que les modèles de performance pour les applications parallèles.

Un modèle pour l'évaluation de performance de placements d'applications distribuées est ensuite présenté en partie II. Il se base sur des modélisations de l'application, de l'architecture et du placement (chapitre 6), qui sont ensuite utilisées pour étudier les synchronisations (chapitre 7), la concurrence (chapitre 8) et les communications (chapitre 9) entre les différents codes de l'application et déterminer ainsi leur performance.

La partie III est consacrée à l'optimisation des placements en utilisant la programmation par contrainte (chapitre 10). Une modélisation de notre problème de placement sous forme de problème de contraintes est ensuite proposée (chapitre 11).

La partie IV présente différentes expérimentations réalisées sur différentes applications afin de valider notre modèle de performance ainsi que notre approche basée sur l'utilisation de contraintes pour la résolution et l'optimisation de notre problème de placement.

Finalement nous dressons le bilan de notre approche ainsi que les perspectives qu'elle nous permet d'envisager.

Première partie

Etat de l'art

Chapitre 2

La Réalité Virtuelle

Sommaire

C'est vers la fin des années 1970 que les expériences qualifiées alors de "réalité artificielle" commencent à se développer grâce à la mise au point des premiers bras à retour de force et casques de visualisation. Le terme "réalité virtuelle" apparaît plus tard, dans les années 1980, mais ce n'est qu'à partir de la fin de cette décennie que la réalité virtuelle va connaître un essor considérable. En effet les ordinateurs deviennent plus puissants, de nouveaux périphériques sont développés pour l'interaction haptique et la visualisation ouvrant ainsi la voie à un large champ d'applications dans de très nombreux domaines. Les médias ont également largement contribué à populariser cette discipline notamment au travers des oeuvres de science-fiction.

La réalité virtuelle propose à l'utilisateur d'expérimenter un environnement décrit par un programme informatique par l'intermédiaire d'interfaces matérielles. Les possibilités offertes par les applications de réalité virtuelle sont donc liées aux performances des processeurs et aux possibilités des interfaces homme-machine. Ainsi les premiers environnements virtuels des années 1980 étaient très simplifiés afin de pouvoir être exécutés de manière interactive sur les machines de l'époque. L'évolution des matériels permet d'espérer des applications capables de manipuler toujours plus d'information afin d'obtenir, par exemple, des environnements virtuels contenant un plus grand nombre d'objets, des simulations plus précises sur des domaines plus vastes, l'immersion d'en un même environnement d'un plus grand nombre d'utilisateurs, etc. Cependant, cette évolution entraîne l'intégration de toujours plus de codes au sein d'une même application. Ces codes sont souvent hétérogènes, par exemple ceux de simulation et de visualisation scientifique, de gestion interfaces homme-machine et de l'intelligence artificielle ne nécessitent pas la même puissance de calcul ni la même quantité de mémoire. De plus,

ils sont souvent élaborés par différents développeurs spécialisés dans ces différents domaines. L'architecture d'une application de réalité virtuelle doit donc faciliter sa conception par plusieurs développeurs, l'intégration de nouveaux codes, leur maintenance ainsi que leur réutilisation.

Nous commençons par présenter les concepts que l'on retrouve dans les approches couramment utilisées pour structurer et organiser les applications de réalité virtuelle. Puis, à un plus bas niveau, nous présentons les outils logiciels qui permettent de simplifier l'écriture des différents codes de visualisation, d'interaction et de simulation.

2.1 Architecture d'une application de RV

Certains concepts sont communs à de nombreuses approches employées pour le développement d'application de réalité virtuelle. Nous présentons dans cette section les principaux concepts utilisés. Nous nous intéressons plus particulièrement aux différentes propositions pour le découpage d'une application de réalité virtuelle afin de simplifier le développement mais également pour permettre de distribuer les différents codes et ainsi améliorer les performances.

2.1.1 La boucle interactive

Une application de réalité virtuelle est susceptible de s'exécuter indéfiniment, ou du moins jusqu'à ce que l'utilisateur décide de l'arrêter. Sa programmation s'exprime donc assez naturellement par une boucle.

Afin de produire une interaction avec l'utilisateur, une application de RV comporte trois étapes répétées dans la boucle du programme :

1. capture des actions de l'utilisateur ;
2. traitement de l'action par la simulation et production d'une réponse ;
3. émission de la réponse vers l'utilisateur.

L'utilisateur peut interagir à tout moment avec l'application, ces étapes constituent ce que l'on appelle la *boucle interactive*, représentée à la figure 2.1. La fréquence et la latence de cette boucle déterminent l'interactivité de l'application.

Notons que ce schéma se retrouve notamment dans la librairie GLUT basée sur OpenGL, et est couramment utilisé dans le domaine des jeux vidéo, dans ce dernier cas cette structure est nommée boucle de jeu.

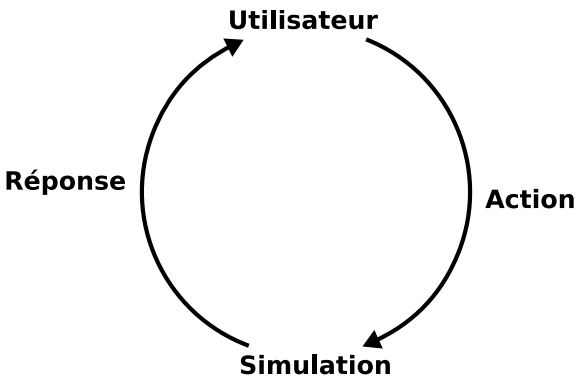

FIG. 2.1 – La boucle interactive

Cette conception présente cependant plusieurs inconvénients. Premièrement, cela impose une synchronisation de ces étapes alors que conceptuellement les codes n'ont pas toujours besoin d'un couplage fort. Deuxièmement, l'intégration de simulations plus complexes entraîne une augmentation du temps d'exécution de la boucle. Une possibilité pour résoudre ce problème consiste à paralléliser les étapes de la boucle, mais cette solution ne fait que repousser le problème. L'intégration des différentes étapes de l'application au sein d'une même boucle synchrone n'est donc pas une approche satisfaisante pour considérer des applications plus complexes en conservant de bonnes performances. .

2.1.2 Modèles pour les applications de réalité virtuelle

La réalité virtuelle est un outil qui concerne de nombreux domaines scientifiques et, dans de nombreux cas, les applications de réalité virtuelle sont développées de manière ad hoc pour un problème donné. Par exemple, dans [14], une application de réalité virtuelle est spécifiquement développée pour effectuer des manipulations microscopiques. Cependant, leur conception requiert une analyse spécifique qui les rend souvent difficiles à maintenir et à étendre, par exemple pour piloter un nouveau périphérique ou exploiter des architectures multi-processeurs ou des grappes de PC. Il est alors nécessaire de définir des modèles pour fournir un cadre à leur développement et ainsi simplifier leur conception et faciliter leur extension et distribution.

Le modèle *Decoupled Simulation Model*, introduit par Shaw et al. [45, 46], propose de découper une application en quatre parties : le modèle géométrique, la présentation (affichage, son), l'interaction et la simulation, mais seulement deux tâches peuvent s'exécuter simultanément, l'une contient la simulation, l'autre les parties restantes. En effet une simulation évolue d'elle-même lorsque aucune action de l'utilisateur n'est produite. Une simulation est ainsi également modélisable sous forme d'une boucle autonome possédant un état interne qui évolue à chaque étape de celle-ci. La simulation et l'interaction sont donc deux processus qui peuvent être implantés de manière asynchrone. Ce découplage permet d'expérimenter des simulations qui ne sont pas interactives de par leur temps de calcul, on parle dans ce cas de *computational steering*. Notons qu'une implentation de ce modèle, appelée MR Toolkit, est également proposée dans [45, 46].

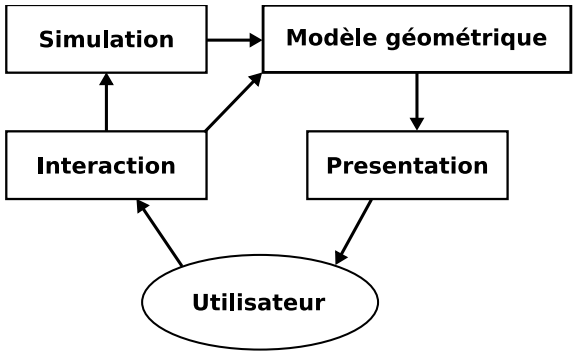

FIG. 2.2 – La décomposition proposée par le *Decoupled Simulation Model*

Le modèle VRID (*Virtual Reality Interface Design*), proposé par Tanriverdi et al. [49], va plus loin que l'approche précédente en permettant de découpler complètement les différentes parties de l'application. L'application est toujours décomposée en quatre parties à l'instar du

Decoupled Simulation Model. Le couplage et les communications entre ces parties est pris en charge par une cinquième partie, le *mediateur*. L'organisation de ces différentes parties est présentée figure 2.3 Le développeur peut dans ce cas configurer le *mediateur* afin de définir le couplage qu'il souhaite entre ses différents codes. Cette approche permet donc de s'abstraire de la politique de couplage. Cependant, l'efficacité de ce modèle est limitée par la centralisation du couplage et des communications et donc par la performance du *mediateur*.

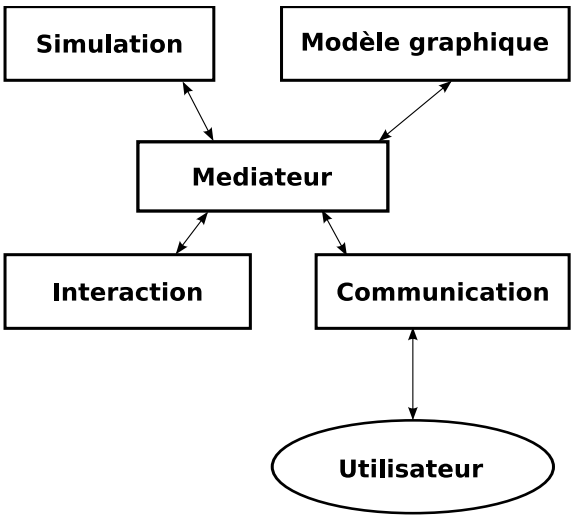

FIG. 2.3 – Le découplage du modèle VRID

Ces modèles de haut niveau facilitent la conception du code et sa distribution sur des machines parallèles par l'utilisation d'une structure donc les tâches peuvent être découplées, ce qui permet d'améliorer les performances et d'envisager des simulations plus lourdes. Cependant ces modèles sont limités à un nombre défini de tâches, soit de par la conception du modèle, soit par le fait de centraliser la gestion des tâches, et ne permettent donc pas d'envisager un passage à des applications plus complexes.

2.1.3 Plateformes de réalité virtuelle

Les plateformes de réalité virtuelle proposent de simplifier le développement en fournissant une implantation d'un modèle d'application, ainsi qu'un environnement d'exécution pour les différentes tâches de rendu, d'interaction et de simulation. L'utilisateur peut ensuite étendre cette implantation en ajoutant le code spécifique à son application.

CAVELib

La bibliothèque CAVELib [23] fut développée afin de piloter les plateformes de réalité virtuelle de type CAVE qui nécessite un rendu sur de multiples surfaces. Afin de permettre un rendu distribué il est possible d'utiliser de multiples cartes graphiques au sein d'un même ordinateur, en utilisant par exemple les stations SGI Onyx. L'application est également scindée en plusieurs processus effectuant la gestion des interfaces, la simulation, et le rendu pour tirer parti de ces architectures multi-processeurs. Cependant cette approche est limitée au nombre de cartes pouvant être installées sur une même machine. Une solution permettant une distribution plus importante consiste à distribuer chaque rendu partiel à une machine différente. CAVELib

permet ces deux approches : dans le premier cas plusieurs processus de rendu sont créés, alors que dans le second cas l'application est répliquée sur les différents noeuds. Cependant, cette approche est très liée à l'utilisation de plateformes de type CAVE.

VRJuggler

L'environnement VRJuggler [20] fut créé pour permettre une plus grande abstraction des matériels utilisés par une application de réalité virtuelle. VRJuggler définit un modèle de programmation qui décompose une application en un ensemble de cinq *managers* :

Config : gère la configuration de l'application et permet sa modification pendant l'exécution ;

Performance : permet d'obtenir les statistiques de performance de l'application ;

Input : contrôle les différents périphériques d'entrées ;

Display : gère l'affichage ;

Draw : implante le rendu.

Chaque *manager* possède un ensemble de classes que l'utilisateur peut étendre afin d'intégrer son propre code. Cette approche permet d'abstraire l'application des logiciels utilisés pour effectuer le rendu ou gérer les périphériques d'interaction. VRJuggler fournit également par défaut un ensemble de classes prédéfinies pour l'utilisation de périphériques haptiques, via des classes basées sur VRPN ou Trackd, ainsi que différents moteurs de rendu, par exemple OpenGL, Performer, ou OpenSceneGraph. Ces facilités permettent au développeur de se concentrer sur chacun de ses codes et de ne pas se préoccuper de leur organisation.

L'exécution de ces *managers* est cependant effectuée séquentiellement par un micro-noyau au sein d'une boucle interactive. Une application *VRJuggler* reste donc limitée par ce fonctionnement trop synchrone. De par sa conception, la plateforme VRJuggler se destine aux applications de réalité virtuelle ne nécessitant pas de simulations lourdes. Elle propose une couche d'abstraction pour les périphériques d'interaction, et les dispositifs d'affichage. La reconfiguration d'une application VRJuggler pour l'utilisation d'un autre périphérique se fait par simple modification d'un fichier de configuration.

NetJuggler

La plateforme NetJuggler [12] fut développée au Laboratoire d'Informatique Fondamentale d'Orléans et a été la première extension de VRJuggler apportant le support de l'exécution sur grappe de PC. Cette fonctionnalité a ensuite été également introduite dans l'extension ClusterJuggler [39] développée par l'équipe de développement de VRJuggler et est désormais intégrée en standard dans VRJuggler 2.0.

Dans ces différentes plateformes, la distribution du calcul sur les noeuds est effectuée, à l'instar de CAVELib, par réplication du code de l'application. Le code exécuté sur chaque machine est donc identique, seuls changent les fichiers de configuration pour la visualisation qui permettent de spécifier des points de vue différents pour chaque machine.

Cette approche n'améliore donc pas les performances de l'application, par contre elle permet de distribuer l'affichage. On peut ainsi porter facilement une application VRJuggler d'un environnement de type *CAVE* à un *workbench* ou un mur d'image.

Une solution qui peut être apportée pour paralléliser la simulation consiste à utiliser une bibliothèque de communication de type MPI [29]. Cependant, la structure même de VRJuggler ne permet pas de désynchroniser facilement le code de simulation de celui de visualisation. Une autre solution est apportée par l'environnement Syzygy que nous présentons maintenant.

Syzygy

L'environnement Syzygy [42] est spécialement conçu pour tirer parti des architectures de type grappes de PC. Une application Syzygy est construite par l'assemblage de différents composants fournis par l'environnement pour gérer le rendu visuel, sonore, les entrées/sorties. Syzygy propose deux modes de fonctionnement :

- le mode *graphe de scène distribué* consiste à répartir les différentes parties de l'application sur les noeuds concernés ;
- le mode *maître/esclave* consite, à l'instar de NetJuggler, à répliquer le code de l'application et à synchroniser l'exécution des différentes instances ainsi créées.

Ces modes reposent sur un système distribué nommé *Phleet* qui prend en charge le lancement et l'arrêt des processus, la gestion des communications et des synchronisations entre les différents processus ainsi que la reconfiguration dynamique de l'application. Le système *Phleet* propose ainsi des fonctionnalités comparables à celles d'un système d'exploitation distribué et permet de s'abstraire du type de noeud utilisé et du système d'exploitation. Syzygy permet ainsi de répartir une application sur des architectures distribuées hétérogènes.

L'utilisation de cet environnement en mode *graphe de scène distribué* permet de répartir les calculs et ainsi d'améliorer les performances. Cependant, pour exploiter ce mode il convient d'utiliser les composants définis par Syzygy. Le développement est donc restreint à ce cadre applicatif et il est par exemple impossible d'exploiter un code de rendu différent sans avoir à reprogrammer Syzygy. Dans ce cas il faut utiliser le mode *maître/esclave* qui présente les mêmes inconvénients que NetJuggler.

OpenMASK

OpenMASK [5, 25, 28], (*Open* {*Multi-threaded* | *Modular*} *Animation and Simulation* {*Kernel* | *Kit*}) est issue de la combinaison de l'environnement GASP [26] et du travail de thèse de Margery [36]. Le modèle définit par OpenMASK propose une approche de plus haut niveau, par rapport aux modèles cités précédemment. L'application n'est plus décomposée en un ensemble de parties spécialisées mais composée d'objets nommés OpenMASK Simulated Objects (OSO) qui peuvent indifféremment implanter une simulation, la description d'un objet virtuel ou encore un type d'interaction. Par exemple dans un environnement virtuel simulant le comportement d'une foule, chaque individu sera représenté par un objet OpenMASK. Il s'agit donc d'une approche orientée multi-agents. Toutefois, le niveau de granularité d'un objet est laissé au choix du développeur. Par exemple un véhicule peut être considéré comme un objet ou comme un ensemble d'objets : un moteur, une carosserie, des roues, etc.

La création et l'exécution des objets ainsi que leur couplage et interconnexion sont pris en charge par des objets appelés *kernels*. L'organisation d'une application OpenMASK est décrite figure 2.4. Des *kernels* permettant l'exécution de codes parallèles ou distribués sont fournis par OpenMASK ce qui lui permet de supporter l'exécution sur des stations multiprocesseurs

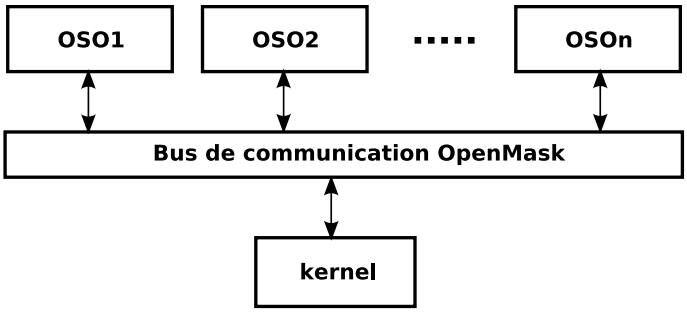

FIG. 2.4 – Schéma d'une application OpenMASK

ou sur des grappes de PC. La plateforme OpenMASK apporte également une abstraction du système et de l'architecture en proposant des *kernels* pour les systèmes Irix, Linux, MacOS et Windows.

Afin de faire communiquer les objets entre eux, chaque objet dispose d'une interface constituée de points d'entrées/sorties qui permettent de l'interconnecter aux autres objets de l'application. Ainsi le point d'entrée d'un objet est relié à un point de sortie d'un autre objet. Des points d'accès spéciaux permettent également de régler les paramètres de contrôle d'un objet ou de capturer des évènements. Dans ce dernier cas un code spécifique est déclenché afin de produire une réponse adaptée à l'évènement reçu. Dans le cas d'une application distribuée, un objet accède à un objet distant par l'intermédiaire d'une représentation locale de cet l'objet distant. On distingue alors l'objet *référentiel*, effectuant le calcul de son *miroir* qui contient uniquement une copie de l'interface du référentiel et de ses valeurs. Les données du référentiel et de ses miroirs sont alors synchronisées pour assurer la cohérence de l'application.

Concernant l'activation des objets OpenMASK, ceux-ci peuvent être activés de deux manières différentes :

- périodiquement, par exemple dans le cas d'une simulation qui fournit un résultat à intervalle régulier ;
- ponctuellement, lors de la réception d'un évènement particulier, par exemple une action de l'utilisateur.

Dans le premier cas, les objets possèdent une fréquence d'activation définie par le développeur dans le cadre de l'application considérée. L'exécution d'une application OpenMASK consiste alors en une suite d'étapes d'activation dont la durée est déterminée à partir de la fréquence d'activation des objets. Chaque objet n'est activé qu'une seule fois par étape. Du point de vue de l'ordonnancement, OpenMASK est donc un modèle synchrone [37].

OpenMASK apporte également des facilités pour la création dynamique d'objets sur l'initiative de l'utilisateur ou par le noyau sur demande des autres objets de l'application, par exemple pour créer un objet miroir dans le cas d'une requête sur un objet distribué. La possibilité de migration d'objets est également envisagée d'après les travaux de Zamar [54].

A l'instar de VRJuggler, de nombreux objets prédéfinis sont intégrés en standard à OpenMASK pour former la plateforme VR-OpenMASK. Parmi ces objets on trouve par exemple un objet de visualisation intégrant les bibliothèques Performer, OpenSG et plus récemment Ogre,

un objet pour l'interaction basé sur VRPN, ou encore un objet intégrant la bibliothèque audio OpenAL.

2.2 Outils pour la RV

Les applications de réalité virtuelle sont constituées de codes pour la visualisation, l'interaction et la simulation et cela indépendamment de leur organisation suivant les modèles décrits précédemment. Afin de factoriser les développements de ces différents codes et de faciliter leur réutilisation, des outils de haut niveau ont été développés.

Nous présentons premièrement les outils utilisés pour le calcul du rendu et pour la visualisation, puis ceux utilisés pour la gestion des interactions. Finalement nous étudions les approches utilisées pour le développement de simulations.

2.2.1 Rendu

Le rendu est l'étape qui consiste à transformer les informations de l'environnement virtuel en une image exploitable par les périphériques de visualisation. Cette étape peut être effectuée de manière logicielle ou prise en charge par des cartes graphiques qui contiennent des processeurs dédiés afin de décharger le processeur central et de permettre un rendu interactif.

Plusieurs approches existent afin de réaliser cette étape. Celles de plus bas niveau consistent à utiliser directement l'interface OpenGL qui est fournie par les pilotes des cartes graphiques. L'interface OpenGL définit un ensemble de structures de données et de fonctions afin de définir des primitives graphiques et de les manipuler. Le développeur est dans ce cas responsable de la conversion des informations en primitives graphiques et de leur mise en forme. La bibliothèque Chromium propose une implantation d'OpenGL permettant de distribuer l'affichage sur des dispositifs tels que des murs d'image, et ce de manière transparente pour l'utilisateur.

Les approches de plus haut niveau fournissent une couche d'abstraction à OpenGL par l'utilisation d'une structure arborescente, appelée graphe de scène, dont les noeuds sont associés à des objets du monde virtuel. Cette approche est implantée notamment par les bibliothèques OpenSceneGraph et Performer. Le développeur peut ainsi organiser et manipuler facilement les informations à afficher. De plus cette structure permet d'implanter facilement et efficacement des optimisations telles l'élimination des objets non visibles afin de n'effectuer que le rendu des objets qui seront visibles. L'utilisation de techniques de rendu distribué de type *sort-first* : on détermine les primitives visibles dans la scène avant de les envoyer à la carte graphique, est également simplifiée par cette structure. Les approches basées sur des graphes de scène sont adaptées à la création d'environnement complexes comportant de multiples objets.

D'autres outils dédiés à la visualisation scientifique existent également. Par exemple la bibliothèque VTK [43] (*Visualization ToolKit*) intègre des algorithmes de visualisation pour la représentation de champs de vecteurs, de tenseurs, ou pour effectuer du rendu volumique. Notons que les applications VTK peuvent être distribuées en utilisant la bibliothèque ParaView [30].

2.2.2 Interactions

Afin de faciliter le développement et de pouvoir interchanger les matériels d'interaction sans modifier le code des applications, des bibliothèques de haut-niveau ont été développées au-dessus des pilotes spécifiques fournis par les constructeurs de périphériques. Citons par exemple VRPN [31] et Trackd [52] qui permettent, par exemple, de remplacer un bras haptique par une souris 3D ou un gant de données simplement en modifiant des fichiers de configuration.

L'architecture client-serveur utilisée par ces approches permet également de s'abstraire du système d'exploitation. En effet, certains pilotes de périphériques d'interaction requièrent des configurations logicielles spécifiques, certains ne sont disponibles que pour un système d'exploitation donné.

2.2.3 Simulations

Les simulations sont les programmes qui définissent le comportement et l'évolution d'une application de réalité virtuelle. Les algorithmes utilisés dépendent donc des types de comportements que l'on souhaite simuler. Par exemple, les simulations mettant en jeu des entités individuelles (véhicules, personnages) sont généralement implantées à l'aide de systèmes de type multi-agents alors que les simulations physiques sont basées sur des solveurs d'équations.

Les simulations sont souvent les codes les plus exigeant en terme de calcul. Afin de considérer des simulations manipulant toujours plus d'information, une solution consiste à les paralléliser. Suivant le type de code considéré cette parallélisation est plus ou moins aisée. Les simulations basées sur des agents multiples modélisés sous forme de tâches communicantes se prêtent facilement à une parallélisation par répartition des tâches sur différents processeurs ou à une distribution sur plusieurs machines car les volumes de données échangés entre les agents sont généralement faibles. Au contraire, les simulations physiques sont généralement parallélisées par découpage du domaine étudié et les algorithmes employés nécessitent une forte cohérence entre les calculs sur les différents sous-domaines engendrés ce qui engendre un couplage fort et l'échange d'importants volumes de données. Des bibliothèques, telles que Petsc, fournissent des routines et des structures de données pour la parallélisation de solveur d'équations des simulations physiques. Notons que les simulations physiques sont souvent développées en utilisant le langage de programmation Fortran.

2.3 Conclusion

L'utilisation d'outils génériques facilite le développement des différents codes de visualisation, interaction et simulation en proposant une couche d'abstraction au-dessus de la gestion du matériel. De plus les modèles et plateformes présentés dans ce chapitre apportent un cadre au développement d'applications de réalité virtuelle afin de simplifier l'organisation de leurs différents codes. Notre étude nous amène à distinguer deux principaux types d'approches.

D'une part nous avons les modèles tels que VRID ou celui proposé par VRJuggler qui consistent à fournir à l'utilisateur un patron dans lequel insérer les codes pour les différentes

tâches de l'application. Cette approche facilite le développement d'applications de réalité virtuelle mais est limitée aux applications qui rentrent dans le cadre défini. Dans le cas de NetJuggler nous avons vu que la parallélisation et la désynchronisation du code de simulation impose de contourner le modèle de VRJuggler.

D'autre part nous avons un modèle de plus haut-niveau, OpenMASK, qui ne fait pas de distinction entre les différents codes spécifiques d'une application de réalité virtuelle. Cette approche permet d'éviter la restriction imposée par un cadre applicatif et de pouvoir distribuer un nombre plus important d'objets. De plus les communications entre les objets sont prises en charge par la plateforme, le développeur peut donc se concentrer sur la conception des objets. OpenMASK propose donc un modèle plutôt destiné aux applications de type multi-agents.

Les fonctionnements proposés par ces deux approches restent cependant très synchrones ce qui représente une limitation pour la conception d'applications distribuées de plus grande taille et intégrant des objets fonctionnant à des fréquences très différentes. Nous devons donc envisager l'utilisation d'approches permettant de considérer un couplage de code plus fin pour les applications distribuées.

Nous nous intéressons maintenant aux différentes architectures parallèles afin d'étudier leur fonctionnement et le niveau de performance qu'elles sont susceptibles d'apporter aux applications de réalité virtuelle. Nous présentons ensuite les outils pour la programmation parallèle, la distribution et le couplage de codes sur ces architectures et nous étudions leur utilisation pour les applications de réalité virtuelle.

Chapitre 3

Matériels pour le calcul haute-performance

Sommaire

De nombreuses solutions matérielles existent afin d'augmenter la performance des applications. D'une part les processeurs ne cessent d'évoluer en intégrant toujours plus de transistors, des jeux d'instructions spécialisés et en atteignant des fréquences plus élevées. D'autre part les capacités et performances des bus et réseaux augmentent parallèlement ce qui permet d'élaborer des architectures faisant communiquer un ensemble de processeurs afin de mettre en commun leur puissance.

Afin d'étudier l'apport de ces matériels pour les applications de réalité virtuelle, nous allons tout d'abord les présenter séparément. Puis, nous préciserons les architectures que nous allons considérer dans le cadre de notre étude.

3.1 Processeurs

Les processeurs sont les éléments de calcul de base. Leur puissance dépend de leur architecture interne qui détermine le nombre de cycles nécessaires pour effectuer des opérations ainsi que de leur fréquence qui conditionne le nombre de cycles effectués par seconde.

Afin d'améliorer les performances des applications scientifiques des unités parallèles sont intégrées au sein même des processeurs. Une autre technologie émergente est l'intégration de plusieurs coeurs de processeurs au sein d'une même enveloppe physique afin de pouvoir effectuer plusieurs tâches indépendantes simultanément ou découper un calcul en plusieurs tâches parallèles.

3.1.1 Les unités SIMD

Une première solution pour accélerer certains types de calculs est apportée par l'ajout d'unités de type SIMD (Single Instruction Multiple Data) capables d'effectuer des opérations simultanées sur des données différentes. Cette approche utilise donc le paradigme du parallélisme de donnée. L'idée générale est d'améliorer les calculs sur les opérations vectorielles qui sont très utilisées dans les calculs pour le rendu 3D et pour les simulations scientifiques. Ainsi l'addition de deux vecteurs de quatre composantes nécessite quatre opérations d'addition indépendantes. Les instructions SIMD proposent d'utiliser dans ce cas quatre additionneurs effectuant l'addition de chaque composante en parallèle. On peut donc espérer dans ce cas un gain de performance d'un facteur quatre.

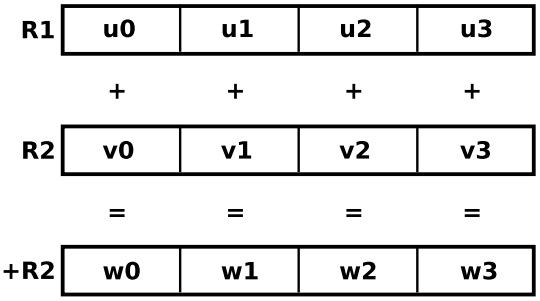

FIG. 3.1 – Addition de 2 vecteurs à 4 composantes

Implantations

Plusieurs jeux d'instructions sont apparus chez les différents fabricants de processeurs. Citons par exemple les plus connus :

- AltiVec développé par IBM, Apple et Freescale et implanté sur les processeurs PowerPC ;
- MMX (Matrix Math eXtensions) et SSE (Streaming SIMD Extensions) introduites par Intel à partir des processeurs Pentium ;
- 3DNow ! développé par AMD et intégré sur les processeurs de la marque à partir du K6-2.

Ces jeux diffèrent sur les instructions et les structures de données qu'ils permettent de traiter. Par exemple, les instructions MMX sont limitées à des opérations sur des données de 64 bits, par exemple quatre entiers de 16 bits, alors que les instructions SSE peuvent traiter des opérations sur 128 bits. Les performances offertes varient également suivant l'implantation matérielle choisie. Certaines séries de processeurs chez AMD et Intel partagent les mêmes jeux d'instructions SSE, cependant leurs implantations diffèrent ainsi que leur performance.

Programmation

Afin de tirer parti d'une unité SIMD il est nécessaire d'utiliser les structures de données et les instructions correspondantes. Le portage d'un jeu d'instruction à un autre requiert donc une réécriture et une recompilation du programme. Le compilateur d'Intel, et plus récemment le compilateur GCC, permettent une analyse du code à la compilation afin de détecter les portions susceptibles d'être vectorisées afin de tirer parti des unités SIMD.

Une approche de plus haut-niveau consiste à utiliser des bibliothèques de routines spécialisées qui constituent une surcouche pour masquer les structures de données et les jeux d'instructions considérés. La bibliothèque ATLAS, destinée au calcul sur les matrices et vecteurs, fournit une implantation de BLAS optimisée pour les instructions SIMD de différents processeurs. Le code est ainsi portable et ne nécessite pas de recompilation grâce à l'utilisation de bibliothèques dynamiques.

3.1.2 Les coeurs multiples

Le développement de nouvelles générations de processeurs est un processus très complexe et long. Afin d'augmenter la performance, de rentabiliser les investissements en recherche et développement, et de diminuer les coûts de fabrication, une solution consiste à intégrer plusieurs coeurs de processeurs au sein d'une même enveloppe. Chaque coeur peut exécuter une tâche différente, de plus un bus interne ainsi qu'une mémoire cache de haut-niveau commune permet de les faire communiquer efficacement.

L'exploitation des performances de ce type d'architecture nécessite de scinder ses programmes en tâches capables de s'exécuter en parallèle. Pour cela le développeur peut utiliser des *threads* afin de répartir les tâches de son programme sur les différents processeurs ou de répartir un même calcul sur des données différentes à la manière des instructions SIMD.

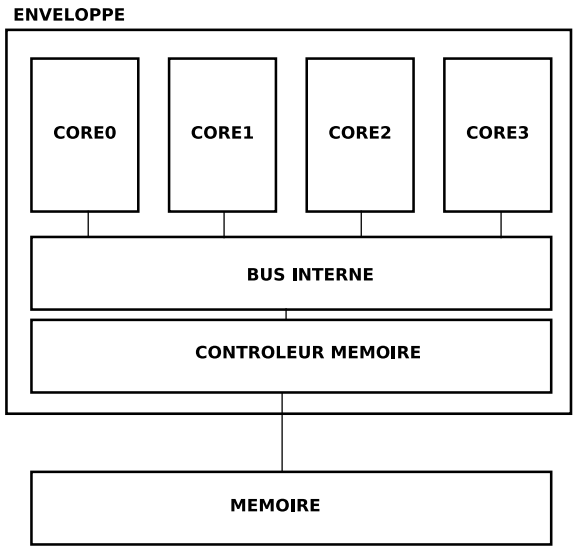

FIG. 3.2 – Processeur doté de 4 coeurs et d'un controlleur mémoire intégré

3.1.3 Perspectives

L'architecture des processeurs évolue vers toujours plus de parallélisme, que ce soit par l'utilisation d'instructions SIMD capables d'effectuer des opérations parallèles sur des données,

ou par l'utilisation de plusieurs coeurs capables d'exécuter plusieurs tâches simultanément. Les processeurs multicoeurs actuels intègrent de 4 à 8 coeurs identiques et des prototypes comportant jusqu'à 80 coeurs existent déjà. Une autre possibilité offerte par cette technologie est l'intégration de coeurs spécialisés dans l'exécution de certaines opérations. Cependant, la finesse de gravure des composants se rapproche des limites de la physique classique et il devient de plus en plus difficile d'intégrer toujours plus de transistors au sein d'un même processeur, notamment du fait de leur dégagement thermique.

Dans le but de fournir toujours plus de puissance pour les applications de réalité virtuelle, une solution est de considérer des architectures multiprocesseurs plus vastes.

3.2 Architectures à mémoire partagée

Les architectures à mémoire partagée sont constituées de processeurs accédant à une mémoire commune. On distingue plusieurs variantes des ces architectures suivant la manière dont la mémoire est accédée.

3.2.1 Architecture à accès mémoire uniforme

Les architectures à mémoire partagée de type UMA (Uniform Memory Access), également appelées architecture SMP (Symmetric Multiprocessing), sont composées de plusieurs processeurs accédant à une mémoire commune. Les processus peuvent s'exécuter de manière asynchrone sur ces différents processeurs et utiliser cette mémoire afin de communiquer entre eux. Le coût de ces communications inter-processus est donc faible puisque égal au temps d'accès des processeurs à la mémoire. Cependant l'accès concurrent à la mémoire nécessite un mécanisme de protection afin d'éviter les écritures concurrentes qui consiste à ne laisser qu'un processus écrire en mémoire et à bloquer les autres. Des contrôleurs et des bus d'interconnexion spécialement conçus pour ces architectures permettent d'éviter un blocage global des accès mémoires.

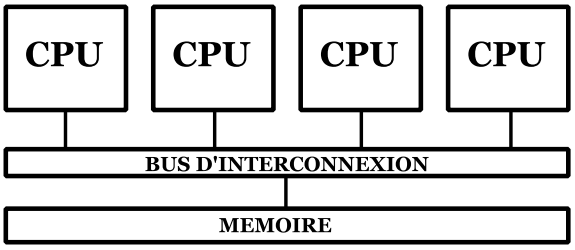

FIG. 3.3 – Architecture SMP de 4 processeurs

Ce type d'architecture entraîne également des problèmes de cohérence de cache entre les processeurs. Lorsqu'un processeur accède à une zone mémoire celle-ci est recopiée dans son cache interne pour des raisons de performance. Cette même zone mémoire peut être également mise en cache par un autre processeur. Toute modification effectuée par un processeur sur une donnée dans son cache invalide le cache des autres processeurs possédant la même donnée. Chaque modification doit donc être répercutée sur la mémoire cache des autres processeurs afin d'assurer une cohérence globale des données. Les architectures à mémoire partagée doivent donc intégrer un mécanisme de cohérence de cache. La performances des applications dépend donc de la manière dont ils accèdent aux données. Si les processeurs mettent en cache les mêmes zones mémoire alors le mécanisme de cohérence de cache va devoir effectuer beaucoup

de communications entre les caches des différents processeurs afin d'assurer la cohérence. L'exécution peut donc s'avérer inefficace. Le schéma d'accès des processeurs aux données conditionne donc les performances des programmes sur ces architectures.

Les architectures UMA présente l'avantage d'être simples à programmer et offre de très bonnes performances à condition d'utiliser des algorithmes et des structures de données limitant les accès mémoires concurrents et les invalidations de cache. Dans ce cas les performances peuvent bénéficier d'une amélioration des performances par un facteur correspondant au nombre de processeurs de la machine. Cependant l'augmentation du nombre de processeurs sur ce type d'architecture entraîne une augmentation des conflits d'accès à la mémoire et ne passent donc pas à l'échelle. Les calculateurs qui utilisent cette approche ont donc généralement un nombre limité de processeurs.

3.2.2 Architecture à accès mémoire non uniforme

L'approche NUMA (Non Uniform Memory Access) se situe juste au-dessus de l'approche UMA. En effet, une machine basée sur une architecture NUMA peut être considérée comme un assemblage de blocs UMA communiquant par un bus. L'accès à la mémoire est donc non uniforme car le bus introduit une latence supplémentaire lorsqu'un processeur accède à une mémoire distante. Cette architecture présente l'avantage de pouvoir intégrer plus de processeurs et de mémoire au sein d'une même machine. Chaque sous-bloc UMA peut donc accéder à sa mémoire locale de manière asynchrone.

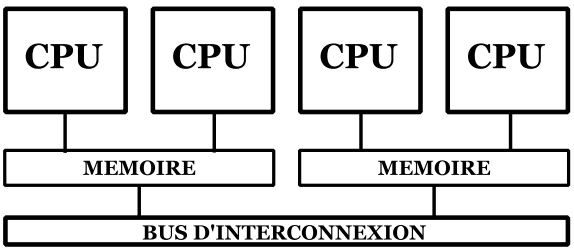

FIG. 3.4 – Architecture NUMA de 2 ensembles de 2 processeurs

Cependant il reste toujours le problème de cohérence de cache entre processeurs et d'accès mémoire concurrent inhérent à l'architecture UMA. Certaines machines NUMA n'intègrent pas de mécanisme de cohérence de cache et c'est au développeur qu'est laissé la tâche de garantir l'absence d'accès concurrents aux données afin d'éviter l'indéterminisme. D'autres intègrent un tel mécanisme de cohérence de cache sont qualifiées de ccNUMA (cache coherent NUMA). A l'instar des architectures UMA, leur performance dépend donc fortement des accès aux données effectués par l'application. Cette approche permet de repousser la limite d'intégration de l'approche UMA mais possède toujours une limite de part son architecture.

3.3 Architectures à mémoire distribuée

Les architectures à mémoire distribuée sont constituées de couples processeur-mémoire, appelés noeuds, interconnectés par un réseau.

Cette approche présente de nombreux avantages sur les architectures à mémoire partagée :

- conception simplifiée ;
- extensibilité par simple ajout de noeuds sur le réseau ;
- gestion des pannes.

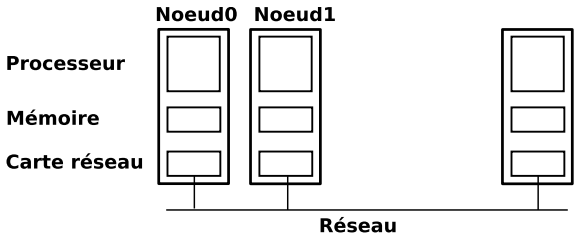

FIG. 3.5 – Schéma d'une architecture à mémoire distribuéé

Cependant leur programmation est plus complexe que sur les architectures à mémoire partagée car il faut utiliser des réseaux pour échanger de l'information, ce qui introduit une latence et des temps de communication que l'on ne peut plus négliger ainsi que des problèmes d'accès concurrents susceptibles d'engendrer une contention et donc de faire chuter les performances.

3.3.1 Configurations

Suivant le niveau de couplage et de communication entre les machines on peut distinguer différentes approches basées sur le principe des architectures distribuées :

- les réseaux pair-à-pair, utilisés pour des calculs faiblement couplés et générant peu de communications, citons par exemple les projets de décodage du génome ou de traitement de signaux de radio-astronomie SETI@Home qui utilisent des ordinateurs personnels reliés à Internet ;
- les supercalculateurs, actuellement majoritairement conçus sur la base de cette architecture [7] comptent plusieurs milliers de processeurs interconnectés par des réseaux haute-performance, leur puissance est utilisée pour des effectuer des calculs lourds nécessaires par exemple aux simulations climatiques ou nucléaires.
- les grappes, ensemble d'ordinateurs connectés par des réseaux locaux, peuvent être dédiées à une tâche de calcul lourd, par exemple une grappe de rendu, ou partagées par différents utilisateurs pour déporter leurs calculs ;
- les grilles de calcul, qui peuvent être vues comme des grappes constituées elles-mêmes de grappes ou de supercalculateurs, permettent de considérer des problèmes encore plus vastes.

Les réseaux pair-à-pair ne disposent pas de réseaux suffisamment performants (débits, qualité de service) pour pouvoir les utiliser pour les applications de réalité virtuelle. Les supercalculateurs et grilles de calcul sont utilisés pour effectuer des simulations complexes nécessitant des temps de calculs importants et nécessitent la mise en place d'infrastructures lourdes. Nous envisageons donc plus particulièrement l'utilisation de grappes de PC pour la réalisation d'applications de réalité virtuelle.

3.3.2 Grappes de PC pour la réalité virtuelle

Les PC sont constitués de composants définis par des standards, fabriqués et diffusés en grande quantité donc peu coûteux. De plus leurs processeurs et cartes graphiques sont performants et ont rapidement intégré les innovations autrefois réservées aux processeurs des calculateurs et atteignent aujourd'hui des niveaux de performance comparables. Les assembler pour constituer une grappe et ainsi mettre en commun leur performance représente donc une solution très intéressante par rapport aux calculateurs basés sur des architectures propriétaires. Un autre avantage est la possibilité d'étendre une grappe par simple ajout de noeuds

FIG. 3.6 – Plateformes de réalité virtuelle Immersia (à gauche), et GrImage (à droite) basées sur des grappes de PC

supplémentaires.

Les grappes de PC représentent donc une solution intéressante pour fournir plus de puissance aux applications de réalité virtuelle. Ces grappes sont souvent constituées de noeuds et réseaux hétérogènes car elles évoluent par ajout de noeuds plus récents. De plus certaines machines peuvent être utilisées pour effectuer des tâches spécifiques par exemple une sous-partie de la grappe peut être dédiée au calcul du rendu de l'environnement virtuel.

Dans de nombreux laboratoires, les grappes de PC remplacent progressivement les stations propriétaires SGI. Par exemple la plateforme de réalité virtuelle Immersia (figure 3.6) de l'IRISA est équipée d'une grappe de trois PC bi-processeurs. La plateforme GrImage de l'INRIA Rhônes-Alpes est constituée de deux sous-grappes dotées de respectivement 11 et 16 noeuds bi-processeurs.

3.4 Conclusion

L'évolution des ordinateurs est limitée par des problèmes d'intégration. Les architectures actuelles évoluent donc vers des configurations intégrant de plus en plus de processeurs ainsi que différents niveaux de parallélisme. Les grappes de PC sont particulièrement adaptées aux besoins des applications de réalité virtuelle de par leur puissance, leur évolutivité et leur coût. Ces architectures sont généralement hétérogènes pour correspondre aux différents besoins des codes de ces applications.

Nous avons vu dans le chapitre précédemment que différents outils existent pour faciliter la distribution du code ainsi que leur couplage. Cependant la décomposition des applications basées sur ces approches est limitée soit par le modèle utilisé, soit par la présence d'un élément centralisateur.

Afin de tirer parti de grappes de PC ces différents codes doivent être décomposés plus efficacement. Nous présentons maintenant les outils utilisés pour la programmation d'applications distribuées et nous étudions plus particulièrement leur adéquation pour la programmation d'applications hétérogènes de réalité virtuelle.

Chapitre 4

Programmation d'applications parallèles et distribuées

Sommaire

L'évolution des architectures, que ce soit celles des ordinateurs personnels, des stations de travail ou des supercalculateurs, impose de sortir du modèle de programmation séquentiel afin de pouvoir accroître les performances des programmes.

Il existe de nombreuses approches pour la parallélisation et la distribution de codes sur architectures distribuées. Certaines sont de plus haut niveau et facilitent l'écriture et la distribution du code alors que d'autres adoptent une approche bas-niveau afin d'optimiser les performances. Leur utilisation dépend donc fortement du type d'application considéré.

Nous présentons ici les approches les plus communément utilisées et nous étudions les contextes les plus adaptés à leur utilisation. Nous étudions également plus précisément leur adéquation pour la programmation d'applications de réalité virtuelle.

4.1 Programmation par passage de messages

La programmation distribuée par passage de messages consiste à exprimer les communications et synchronisations entre processus dans le code des programmes par des appels explicites à des fonctions d'émission et de réception de messages. Il s'agit donc d'une approche bas-niveau, la gestion des communications, des synchronisations ainsi que de la répartition et l'équilibrage des calculs étant laissées à la charge du développeur.

4.1.1 Implantations

Plusieurs interfaces de programmation existent, notamment PVM (Parallel Virtual Machine) et MPI (Message Passing Interface), chacune définissant son propre ensemble de primitives pour les opérations de communication et de synchonisation. Ces interfaces permettent d'écrire des programmes SPMD (Single Program Multiple Data), dans ce cas les processus exécutent tous le même programme, ou MPMD (Multiple Programs Multiple Data), des programmes distincts sont alors exécutés sur des données différentes. Le code engendré est portable puisqu'il peut fonctionner indifféremment sur une architecture distribuée, une architecture à mémoire partagée, ou même sur une machine monoprocesseur.

La programmation par passage de messages repose sur l'utilisation de bibliothèques de communications. L'interface MPI est implantée par des bibliothèques telles que MPICH, LAM/MPI ou plus récemment OpenMPI.

4.1.2 Remarques

L'utilisation de bibliothèques de passage de message dans une application introduit des risques d'interblocages et d'indéterminisme. Toute réception doit donc correspondre à une émission. En effet une routine de réception peut ne jamais recevoir de message et donc bloquer le processus correspondant. De même, le fonctionnement asynchrone des processus peut entraîner différents entrelacements des communications, le résultat obtenu peut donc varier d'une exécution à une autre. Les applications basées sur ces librairies utilisent donc le plus souvent des schémas réguliers de communication et de synchronisation afin d'éviter les risques d'interblocages et l'indéterminisme. Un autre point important de ces bibliothèques est que les schémas de communication et de synchronisation sont intégrés au code et toute modification requiert une réécriture de celui-ci et une recompilation.

La conception d'applications de réalité virtuelle, intrinsèquement constituées de codes hétérogènes disposant de schémas de communication et de synchronisation complexes, apparaît donc difficilement réalisable à partir de cette approche car il devient difficile de garantir l'absence d'interblocages et d'indéterminisme. De plus la necessité de modifier et de recompiler le code lors d'une modification des schémas de communication et de synchronisation rend difficile la réutilisation du code.

4.2 Migration de processus

Sur une architecture à mémoire partagée, la migration de processus permet de déplacer des processus d'un processeur à un autre afin de mieux répartir leurs charges et d'optimiser ainsi les performances des applications. Ce processus est géré par le système d'exploitation et son ordonnance de manière totalement transparente pour l'utilisateur.

L'extension de cette approche aux architectures à mémoire distribuée permet de les considérer comme une architecture à mémoire partagée. La migration de processus est donc une approche de haut-niveau qui permet de masquer la gestion des communications et du placement des processus.

4.2.1 Implantations

Les systèmes MOSIX, OpenMOSIX, le pendant libre de MOSIX, ou encore Kerrighed implantent cette approche de manière logicielle grâce à une extension du noyau Linux. Le placement des processus sur les différents noeuds est alors effectué par un ordonnanceur global de manière transparente pour l'utilisateur. Si l'un des noeuds est trop chargé ses processus sont migrés vers d'autres noeuds afin de répartir au mieux les charges.

Les applications ne nécessitent pas de modifications ni de recompilation pour bénéficier de cette approche. L'ordonnanceur est basé sur des algorithmes adaptatifs de gestion des ressources qui contrôlent la charge des processeurs et les besoins des processus. Cela permet de tenir compte des performances de chaque noeud lorsque l'architecture est hétérogène.

4.2.2 Applications

La migration de processus permet de tirer parti d'une architecture de type grappe dans le cadre d'une utilisation multi-utilisateur de manière simple et transparente. Dans ce cas les processus sont indépendants et ne communiquent pas entre eux, on peut donc obtenir de bonnes performances.

Les applications distribuées, composées de plusieurs processus peuvent également exploiter l'équilibrage de charge global afin d'optimiser leur répartition sur les processeurs. Cependant la migration de processus introduit une latence lorsqu'un processus est déplacé. Dans le cas d'applications de réalité virtuelle manipulant de grosses quantités de données, celles-ci doivent accompagner le processus les utilisant lors d'une migration vers un autre noeud, ce qui implique une latence non négligeable. Or les applications de réalités virtuelles doivent garantir des performances constantes pour ne pas perturber l'expérience de l'utilisateur. De plus les placements de certains processus tels que ceux d'interaction ou de visualisation sont liés à des matériels, la migration de processus n'apparaît donc pas comme une solution adéquate pour le déploiement d'applications de réalité virtuelle.

4.3 Composants distribués

Afin de répartir une application, une approche de plus haut niveau consiste à combiner des appels de procédures à distance, les RPC (*Remote Procedure Call*), avec la programmation objet.

4.3.1 Appels de procédures à distance

Le principe de l'appel de procédure à distance consiste à déporter des codes sur des machines distantes et de pouvoir les appeler comme une procédure locale. De cette manière l'exécution est reportée sur la machine distante ce qui libère la machine locale pour d'autres tâches. Les appels de procédures à distance masquent ainsi les communications explicites des approches par passage de messages. Les communications sont donc effectuées de manière totalement transparente pour l'utilisateur et le développement d'une application distribuée s'ap-

parente à celui d'une application séquentielle. Notons qu'afin de faciliter le développement, il est possible d'adapter les appels de procédure à distance à une approche orientées objet [50].

Un appel de procédure à distance suppose un client, qui effectue une requête contenant les paramètres d'appel, et un serveur, qui fournit la procédure demandée et l'exécute en fonction de la requête. Le résultat est alors retourné au client qui attendait le résultat pour poursuivre son exécution. Certaines implantations d'appels de procédures à distance permettent d'effectuer des appels asynchrones [15] afin de ne pas bloquer l'objet appelant. Les performances sont ainsi améliorées par le recouvrement des calculs et des communications.

La communication par appel de procédure implique l'utilisation d'un langage commun, appelé IDL (*Interface Description Language*), pour décrire l'interface des procédure et le format des paramètres. Ce langage permet de faire appel à des procédures indépendamment du langage utilisé pour les programmer. Cependant le codage/décodage des paramètres entre le format IDL et le format utilisé par l'architecture engendre une certaine latence.

Il existe de nombreuses implantations des appels de procédures à distance, citons entre autres :

- ONC RPC, originellement développée par Sun, est l'implantation la plus répandue ;
- RMI (Remote Method Invocation) est une implantation pour le langage Java ;
- XML-RPC utilise le langage XML pour encoder les appels et le protocole HTTP pour les transmettre ;
- MSRPC est l'implantation de Microsoft pour les systèmes Windows.

Ces implantations ne sont pas cependant pas compatibles entre elles car elles utilisent de langages IDL différents et certaines sont limitées à certains systèmes.

4.3.2 Composants

Le composant est un objet associé à une description de son interface. Les opérations de déploiement, d'exécution et de communication sont prises en charge par des conteneurs qui servent donc d'intermédiaire entre les composants. Cette approche permet de masquer les opérations de lancement et de connexions et donc d'abstraire la conception des composants du système et des protocoles de communication utilisés.

La norme la plus répandue pour la conception d'applications basées sur des composants est CORBA (*Common Object Request Broker Architecture*) développé par l'*Object Management Group* [3] dont plusieurs implantations libres existent telles que OmniORB [4] et ORBit [2]. Un composant CORBA est un programme associé à une description XML qui fournit des informations sur le lancement de ce programme ainsi qu'une liste de ports d'entrées/sorties pour permettre les connexions avec les autres composants de l'application. Une application se construit par assemblage de ses briques logicielles en connectant les ports des différents composants.

4.3.3 Remarques

Concevoir une application à base de composants distribués apporte de nombreux avantages en terme de développement. D'une part, la programmation objet utilisée pour décrire les composants facilite la conception et la maintenance du code. Les composants peuvent être

utilisés indépendamment de leur langage de programmation. Un composant est donc facilement réutilisable dans d'autres applications. D'autre part l'appel de procédure à distance est transparent pour le développeur. Afin d'utiliser cette approche pour des applications interactives il convient néanmoins d'optimiser l'appel de procédure en le rendant asynchrone et en optimisant la conversion des messages.

4.4 Le modèle FlowVR

Le modèle FlowVR [10, 11, 13] fut développé afin de faciliter la conception et le couplage d'applications de réalité virtuelle. L'objectif est de fournir au développeur une approche de haut-niveau et d'extraire les schémas de communication et de couplage du code de l'application. Le modèle FlowVR de part son approche n'est pas limité aux applications de réalité virtuelle ou interactives, c'est pourquoi nous le considérons avant tout comme une plateforme pour la distribution d'applications.

FlowVR désigne également l'implantation du modèle du même nom. Il s'agit d'un intergiciel disponible sur les plateformes Linux et MacOS et les architectures 32 et 64 bits d'Intel et AMD ainsi que sur les processeurs PowerPC. Pour la programmation d'applications, FlowVR fournit une bibliothèque écrite en C++. Un ensemble d'outils est également disponible pour spécifier le couplage des codes et faciliter leur déploiement ou encore analyser les traces d'exécution.

4.4.1 Modélisation d'une application distribuée

Pour construire une application, le modèle FlowVR définit trois types de composants correspondant à des fonctions différentes :

- les modules, qui contiennent les différents codes de l'application ;
- les filtres effectuent des transformations sur les messages ;
- les synchroniseurs implantent la politique de couplage entre les composants.

Nous présentons maintenant plus en détail ces différents composants ainsi que la manière dont ils interagissent.

Modules

Chaque module contient un code de l'application et est chargé de son exécution sur une machine cible. Dans une application, les modules sont repérés par un identifiant unique. Par exemple si nous reprenons un découpage classique d'une application de réalité virtuelle, notre application sera découpée en trois modules : *Interaction*, *Simulation* et *Visualisation*.

Afin de communiquer avec les autres modules de l'application, chaque module dispose d'un ensemble de ports d'entrées et de sorties pour respectivement recevoir et émettre des messages vers d'autres modules. Chaque message FlowVR comporte deux parties. D'une part un entête contenant des informations telles que l'identifiant du module émetteur du message et le numéro d'itération de celui-ci. Ces informations, ou estampilles dans la terminologie FlowVR, sont utilisées pour le filtrage et le couplage des modules par les synchroniseurs. Un message contient ensuite les données produites par le module.

La figure 4.1 présente un module d'interaction disposant de deux ports de sorties pour émettre la position du périphérique. Notons que tous les modules disposent également de deux ports nommés *BeginIt* et *endIt*, utilisés pour le couplage avec d'autres modules. Un module n'a donc aucune information sur les autres modules et a seulement accès aux messages qu'il reçoit.

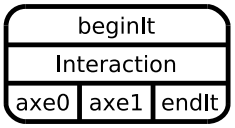

FIG. 4.1 – Exemple de module d'interaction

Les modules sont constitués d'une boucle. A chaque itération de cette boucle, le module fait appel à trois méthodes :

1. *wait()* génère une attente tant qu'un message n'est pas présent sur chacun des ports connectés ;
2. *get()* permet de récupérer les messages sur les ports d'entrées ;
3. *put()* effectue l'envoi d'un message sur un port de sortie.

Notons que les envois ne sont pas bloquants.

Un même module peut être instancié plusieurs fois dans une même application ce qui apporte la possibilité de distribuer son calcul. Par exemple le module *Simulation* peut être instancié plusieurs fois sur différentes machines. Chaque instance est ainsi responsable d'une partie des calculs. Notons que les communications entre instances d'un même module ne sont pas possibles au sein d'une itération. Dans ce cas il est nécessaire de faire appel à une bibliothèque de communication externe telle que, par exemple, MPI.

Le découpage de l'application en plusieurs modules et éventuellement en plusieurs instances de ces modules introduit la nécessité d'effectuer des opérations sur les messages. En effet un module a un nombre fixe de ports définit à sa conception et les connexions sont point-à-point. Il est donc impossible de connecter plusieurs ports de sorties à un même port d'entrée ou un port de sortie à plusieurs ports d'entrées. Nous devons dans ce cas utiliser d'autres composants : les filtres.

Filtres

Les filtres sont des composants chargés de transformer les messages qu'ils reçoivent. Ils disposent, comme les modules, de ports d'entrées et de sorties mais leur fonctionnement n'impose pas d'attente sur les ports d'entrées.

La librairie FlowVR définit un ensemble de filtres couramment utilisés pour modifier les messages, par exemple :

- *broadcast* réplique le message reçue sur ses ports de sorties ;
- *scatter* découpe un message ;
- *gather* assemble les messages reçus en un seul nouveau message.

FlowVR fournit également une interface de programmation pour faciliter la création de filtres.

Afin de pouvoir diriger les connexions FlowVR sur des réseaux différents, un autre type de filtres, les noeuds de routage, sont placés sur les connexions entre les composants. Ces filtres prennent en paramètre l'interface réseau sur laquelle la connexion doit transiter. L'utilisation de réseaux multiples permet ainsi de répartir les connexions afin d'optimiser les temps de communication. Une combinaison avec des filtres de type *scatter* et *gather* permet également de découper les messages en plusieurs parties empruntant des réseaux différents ce qui permet de diminuer les temps de communication.

Les filtres sont indépendants des modules ce qui permet de changer le schéma de communication de l'application sans avoir à modifier le code de l'application.

Synchroniseurs

Les synchroniseurs implantent les politiques de couplage entre les modules à partir des informations fournies par les estampilles des messages. Ils sont donc connectés aux autres composants par des connections spéciales ne transmettant que les estampilles afin de minimiser les communications. Nous présentons maintenant deux types de synchronisations fournis en standard par FlowVR.

La fréquence d'un module peut ainsi être réglée en connectant ses ports *beginIt* et *endIt* à un synchroniseur de type $maxfrequency$. Dans ce cas, le synchroniseur délivre un signal à intervalle régulier sur le port d'entrée *beginIt* du module synchronisé. Le module reste donc bloqué après chacune de ses itérations dans l'attente d'un message du synchroniseur et respecte donc la fréquence imposée.

La combinaison d'un filtre et d'un synchroniseur permet d'implanter un schéma de couplage appelé *greedy* permettant de faire communiquer des modules possédants des fréquences différentes. Le principe de ce schéma consiste à utiliser un filtre ne gardant en mémoire que le dernier message provenant du composant émetteur. Ce message n'est envoyé au récepteur que lorsque celui-ci signale la fin de son itération. Les conséquences du schéma *greedy* sur la sémantique des communications dépendent de la différence de fréquence entre le composant émetteur et récepteur. Si l'émetteur est plus rapide, alors certains messages sont perdus car le récepteur n'est pas assez rapide pour tous les traiter. Ce schéma permet ainsi d'éviter l'accumulation des messages sur les ports d'entrées du récepteur et le dépassement des tampons de réception qui se produit lors de l'utilisation de connexions FIFO. Si le récepteur est plus rapide que l'émetteur alors il peut recevoir plusieurs fois un même message car le filtre *greedy* fournit toujours le dernier message disponible. Suivant la sémantique que le développeur souhaite apporter au schéma *greedy* le comportement du filtre peut être modifié pour obtenir une interpolation ou extrapolation des messages.

D'autres politiques de synchronisation peuvent être implantées par la création des synchroniseurs correspondant à partir de l'API FlowVR.

4.4.2 Implantation

Chaque module FlowVR correspond à un processus. Un processus particulier, appelé *démon*, est utilisé afin de faire communiquer et de synchroniser les processus de l'application. Dans le cas d'application FlowVR réparties sur une architecture distribuée, un processus *démon* est

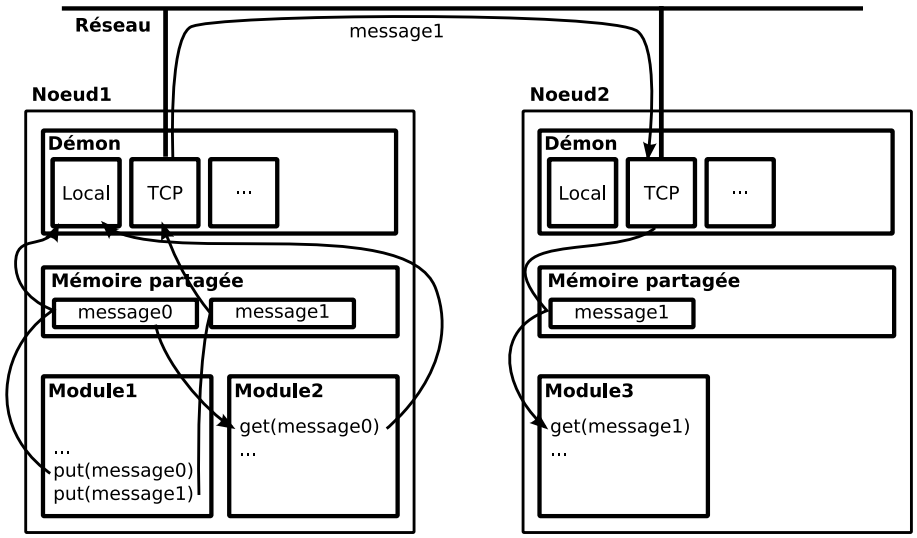

FIG. 4.2 – Schéma de l'architecture FlowVR : démon et modules

exécuté sur chaque noeud. Le *démon* implante les communications inter-processus soit par l'utilisation d'une mémoire partagée lorsque les modules sont sur un même noeud, soit par le protocole TCP entre des modules distants. Les filtres et les synchroniseurs sont implantés sous forme de librairies dynamiques chargées par le *démon* en fonction du type d'opération de filtrage ou de synchronisation requis. Une description de l'organisation des démons, des modules et de leur accès à la mémoire partagée est présenté à la figure 4.2. Dans cet exemple le module $module_1$ émet deux messages (appels à la commande *put()*), l'un à destination d'un autre module placé sur la même machine, le message est alors placé en mémoire partagée et lorsque le module $module_2$ demande à recevoir un message (appel à la fonction *get()*), un pointeur vers celui-ci lui est renvoyé. Lorsque le message est envoyé par le réseau, le démon fait appel à la librairie dynamique de gestion TCP qui effectue l'envoi sur le réseau. Lorsque le démon sur la machine distante reçoit le message, celui-ci le place en mémoire partagée, lorsque le module $module_3$ demande ce message, le démon lui retourne un pointeur vers celui-ci. Cette conception sous forme de bibliothèques dynamiques permet d'ajouter simplement de nouvelles fonctionnalités au démon FlowVR.

4.4.3 Extensions

Plusieurs extensions de FlowVR permettent de faciliter la création d'applications de visualisation ou de réalité virtuelle distribuées :

- FlowVRRender fournit un module pour la visualisation distribuée ;
- FlowVR MPlayer est une surcouche de FlowVRRender permettant de distribuer l'affichage de vidéos sur des supports de type mur d'images ;
- VTK FlowVR permet d'utiliser les fonctionnalités de FlowVRRender pour les applications développées à l'aide de VTK.

D'autres extensions pour intégrer notamment VRPN au sein d'un module FlowVR sont également en cours de développement.

4.5 Conclusion

Différentes approches peuvent être utilisées pour le développement d'applications distribuées. D'un côté les bibliothèques de communication de type MPI/PVM permettent de définir des schémas de communication et de synchronisation précis et sont très performantes. Cependant leur mise en oeuvre est difficile car leur utilisation entraîne des risques d'interblocages et d'indéterminisme. Les programmes basés sur ces bibliothèques utilisent donc généralement le paradigme du parallélisme de données avec des schémas de communication réguliers. Les approches de plus haut-niveau telles que CORBA présentent à l'opposé une facilité de développement ainsi qu'une abstraction des architectures et des systèmes qui sont cependant rendues possibles au détriment des performances. L'approche basée sur FlowVR nous apparaît donc la plus adaptée pour le développement d'applications distribuées de réalité virtuelle. En effet, elle présente l'avantage des approches à composants en terme de développement ainsi qu'une implantation performante et permettant de définir un couplage fin entre les différents parties d'une application.

Cependant, comme pour les autres approches, les performances offertes dépendent du placement des codes, effectué par le développeur, sur les processeurs de l'architecture. L'obtention d'un placement offrant les performances escomptées nécessite de prendre en compte la performance de l'architecture, mais également les performances des codes de l'application ainsi que les synchronisations et les communications entre eux. Cette tâche est donc généralement effectuée par des spécialistes qui ont une connaissance approfondie de l'architecture et de l'application considérée. C'est un processus long qui consiste en une succession d'essais/erreurs et qui est d'autant plus difficile lorsque l'architecture et l'application sont hétérogènes, par exemple dans le cas d'applications de réalité virtuelle sur grappes de PC. Nous proposons donc maintenant d'étudier des modèles de performances pour déterminer ceux qui pourraient nous permettre de simplifier l'évaluation des performances d'un placement sans avoir à effectuer des essais réels sur l'architecture cible.

Chapitre 5

Modèles de performances pour applications distribuées

Sommaire

Nous présentons dans ce chapitre différents modèles de performances afin d'étudier leur adéquation pour la prévision de performances d'applications de réalité virtuelle sur des architectures de type grappes de PC.

5.1 PRAM

Le modèle PRAM, *Parallel Random Access Machine*, est un modèle théorique qui décrit une machine abstraite composée d'une infinité de processeurs accédant à une mémoire partagée également infinie. Les processeurs d'une machine PRAM fonctionnent de manière synchrone, chacun d'eux effectue un calcul local puis écrit dans une mémoire partagée. Pour résoudre les problèmes d'accès concurrents à la mémoire, plusieurs catégories de machines PRAM sont spécifiées donc voici les trois principales :

- EREW, Exclusive Read Exclusive Write, un seul processeur peut accéder à la mémoire pour lire ou écrire une donnée ;
- CREW : Concurrent Read Exclusive Write, l'écriture en mémoire n'est accordée qu'à un processeur alors que tous peuvent y accéder en lecture ;
- CRCW : Concurrent Read Concurrent Write, les processeurs peuvent lire et écrire dans la mémoire simultanément.

PRAM est un modèle de haut-niveau dont l'objectif n'est pas d'être réaliste mais d'aider à réfléchir à la conception d'algorithmes parallèles indépendamment des problèmes de communications. Cette approche n'est donc pas adaptée aux architectures distribuées puisque les communications ne sont pas modélisées, et elle ne permet pas non plus d'appréhender efficacement les applications hétérogènes car elle ne tient pas compte des synchronisations.

5.2 BSP

Le modèle BSP, Bulk Synchronous Parallel, introduit par Valiant [51], fourni un modèle simple pour la programmation d'algorithmes parallèles. Une machine BSP est composée d'un ensemble de couples processeur-mémoire reliés entre eux par un réseau d'interconnexion et est caractérisée par seulement trois paramètres :

- le nombre de processeurs ;
- le temps nécessaire à un échange collectif de messages ;
- le temps nécessaire à une synchronisation globale.

Une grappe de PC homogène peut donc être considérée comme une machine BSP.

5.2.1 Modèle de programmation

Un programme BSP consiste en une suite d'étapes appelées super-étapes.

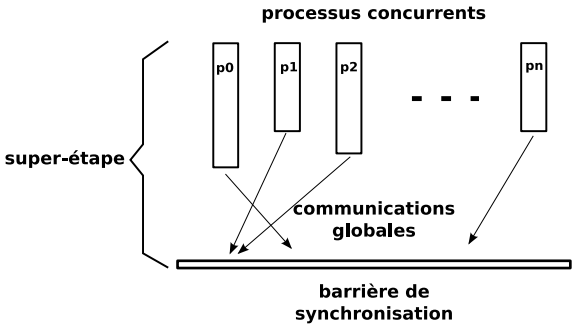

FIG. 5.1 – Description d'une super-étape BSP

Chaque super-étape comporte des processus qui s'exécutent en parallèle et effectuent trois opérations dans l'ordre suivant :

1. un calcul,
2. des communications avec les autres processus de la même super-étape,
3. une barrière de synchronisation commune à tous les processus de la super-étape.

La figure 5.1 représente la décomposition d'un programme suivant le schéma de programmation BSP.

Ce modèle de programmation présente des caractéristiques intéressantes comme l'indépendance de l'architecture considérée et la prédiction de performance pour une architecture donnée. Il garantie également l'absence de blocage de part l'utilisation de communications globales séparées de la synchronisation globale.

5.2.2 Modèle de coût

Le coût d'une super-étape BSP correspond au temps maximum pour effectuer les calculs, effectuer les communications globales, et synchroniser les processus avant la super-étape suivante. Pour chaque processus on évalue donc son temps de calcul ainsi que le temps de communication nécessaire pour émettre et recevoir ses messages. Le temps de communication est calculé en tenant compte de la bande passante du réseau. On détermine ensuite le processus le plus lent. On ajoute également le temps nécessaire à la synchronisation globale de chaque super-étape. Le coût total du programme est obtenu en additionnant le coût de chacune de ses super-étapes.

5.2.3 Remarques

La structure d'une application BSP en super-étapes implique une fréquence identique pour tous les processus d'une même super-étape. De plus les processus d'une même super-étape doivent attendre le processus le plus lent ce qui peut conduire à une exécution inefficace si les processus sont mal équilibrés. Or les applications de RV possèdent intrèsequement des parties déséquilibrées qui fonctionnent à des fréquences très différentes, ce modèle ne se révèle donc pas adapté dans ce cas.

Le modèle BSPWB [41], BSP Without Barrier, propose de relâcher le couplage entre processus en supprimant la barrière de synchronisation globale. La notion de super-étape est remplacée par celle de m-étape composée d'une phase de calcul suivie d'une phase de communication. Les processus n'échangent alors que les données dont ils ont besoin pour commencer l'étape suivante. Cependant les processus ne peuvent communiquer qu'avec les processus dans les m-étapes adjacentes. La communication implique donc toujours une synchronisation forte.

5.3 LogP

LogP [24] est un modèle développé spécifiquement pour les architectures distribuées constituées de machines possédant chacune un processeur et une mémoire et reliées entre elles par un réseau, les communications entre les machines se faisant point à point. L'objectif est d'obtenir un coût réaliste en tenant compte des paramètres de l'architecture considérée. Les architectures distribuées étant intrèsequement asynchrones, le modèle proposé se doit donc de l'être également afin de tirer parti au mieux des performances en n'imposant pas les synchronisations globales coûteuses d'un modèle comme BSP sur ce type d'architectures.

5.3.1 Modélisation de l'architecture

L'architecture considérée par le modèle logP est un ensemble de modules homogènes constitués par une unité de traitement, une mémoire et une interface réseau, et reliés entre eux par un réseau commun, ce qui correspond à une architecture de type grappe de PC. Le

modèle LogP doit son nom aux paramètres de l'architecture qu'il considère pour l'évaluation des performances :

- L : la latence du réseau, c'est à dire le temps de transit d'un mot à travers le réseau.
- o : le surcoût du processeur pour gérer les communications.
- g : le temps minimal entre deux transmissions (ou réceptions) consécutives, ce qui correspond à l'inverse de la bande passante.
- P : le nombre de modules (couples processeur/mémoire) disponibles.

En fonction du type de réseau on peut éventuellement réduire le nombre de paramètres si certains deviennent négligeables par rapport aux autres.

5.3.2 Graphe des dépendances et calcul du coût

Un calcul est représenté par le modèle LogP sous forme d'un graphe : chaque noeud représente un calcul local sur une machine donnée et chaque arête symbolise une communication point à point entre deux calculs. Deux calculs sont donc asynchrones s'ils n'appartiennent pas au même chemin dans l'arbre. L'exécution du programme résulte donc de l'exécution des différents calculs locaux en suivant le schéma de communication induit par le graphe des dépendances. Le résultat du calcul est obtenu une fois le graphe des dépendances parcouru intégralement. Il n'y a donc pas de cycles dans le graphe.

A chaque noeud du graphe des dépendances est associé un coût pour le calcul local exprimé en fonction des données d'entrée et du nombre de processeurs, de même les arêtes sont pondérées par le coût des communications entre deux noeuds distincts exprimé en fonction des paramètres du réseau et du volume de données à communiquer. Le coût se détermine à partir du graphe des dépendances en instanciant les paramètres de l'architecture et en calculant un plus long chemin dans le graphe.

5.3.3 Remarques et limitations

Le modèle LogP ne prend pas en compte les noeuds SMP dotés de plusieurs processeurs. On peut proposer deux approches pour contourner cette limitation : soit utiliser localement des threads, soit ajouter des noeuds supplémentaires pour représenter des calculs effectués en réalité sur plusieurs processeur d'un même noeud. Dans cette dernière configuration certains arcs correspondent à des communications locales et ne sont donc pas associés à un coût si l'on suppose que le coût d'un accès à la mémoire partagée est négligeable devant les communications par le réseau. A l'instar des grappes SMP, les architectures hétérogènes ne sont pas prises en compte par le modèle. En effet LogP ne permet pas de modéliser des grappes disposant de machines dotées de performances différentes. Il faudrait pour cela tenir compte des différentes puissances de traitement, du coût variable des communications (latence et bande passante).

LogP ne prend pas en compte la topologie du réseau, or celle-ci peut influer lourdement sur les performances. Dans le cas d'une grappe de PCs interconnectées à l'aide d'un switch, le coût des communications point à point étant constant, le modèle LogP ne pose pas de problèmes. Notons que LogP considère des communications de messages courts entre les processeurs, une extension pour gérer les messages longs est proposée par Alexandrov et al [9].

Concernant le design d'algorithmes parallèles, l'approche LogP semble bien fonctionner dans les exemples présentés dans [24], mais elle n'apparaît pas évidente, comme montré par Loh

dans [35], pour des problèmes plus larges comme les applications hétérogènes rencontrées en réalité virtuelle. En effet cette approche est peu maniable, la décomposition du programme est donc très dépendante des paramètres de l'architecture, alors que dans les approches basées sur des composants le graphe des dépendances est déterminé à priori par le programmeur qui indique explicitement les parties du code à distribuer et paralléliser.

5.4 Athapascan

Athapascan [22] est une librairie de programmation en C++ qui repose sur 2 niveaux distincts :

- Athapascan-0, une librairie bas-niveau de communication (MPI) et de gestion de tâches(processus légers POSIX)
- Athapascan-1, qui permet d'exprimer le parallélisme à un plus haut niveau.

Le parallélisme est exprimé par la création de plusieurs tâches à l'aide de la commande *fork*, les synchronisations sont basées sur l'analyse des accés aux données partagées identifiées par le type *shared*.

5.4.1 Graphe de dépendance

La dépendance entre les tâches est représentée par un graphe construit en suivant la structure séquentielle du code. Chaque noeud du graphe représente une tâche et chaque arc symbolise une dépendance de donnée entre deux tâches. Quand la méthode *fork* est appelée, une tâche est créée et un noeud est ajouté au graphe de dépendance. Si cette nouvelle tâche contient une référence à une variable partagée par une tâche déjà présente dans le graphe alors un arc est ajouté entre ces deux tâches. Comme dans le modèle LogP le coût d'une exécution est définie par la longueur du chemin le plus long dans le graphe de dépendance.

Dans le cas général, le graphe de dépendance est construit statiquement au début de l'exécution. Les applications de RV contiennent des boucles infinies ce qui suppose un graphe de dépendance infini et donc impossible à construire statiquement. Dans ce cas il est possible de borner la taille de l'arbre de dépendance. Un exemple de l'utilisation d'Athapascan dans ce cas est donné par Zara [27] pour la réalisation d'une simulation interactive de tissu. Chaque étape de la simulation est alors associée à un nouveau graphe dont la création est appelée explicitement.Les tâches sont ensuite placées sur les différents processeurs par l'ordonnanceur suivant une politique de placement basée sur des heuristiques.

5.4.2 Remarques

Athapascan peut ainsi s'intégrer à une application de RV afin de réaliser des simulations. Dans le cadre de notre étude nous considérons des applications de réalité virtuelle distribuées dont les différentes parties comportent des boucles infinies s'exécutant de manière asynchrones. On peut envisager d'utiliser Athapascan pour réaliser chacune de ces différentes parties. Cependant, ces parties doivent communiquer ce qui implique l'utilisation d'une donnée de type *shared* qui impose leur synchronisation et ne permet donc pas un faible couplage.

5.5 SCP

Le modèle de programmation SCP *Structural Clock Programming*, proposé par Melin [38], est associé à un langage de type SPMD *Single Program Multiple Data* ce qui signifie que le programme est le même sur tout les processeurs, seules changent données pour chaque instance du programme.

5.5.1 Graphe des dépendances et coût

Les dépendances entre les instructions du langage sont définies par l'ordre de la syntaxe. Il s'agit donc d'un modèle de programmation à grain fin. Un programme SCP peut donc être représenté par un graphe acyclique construit en suivant l'ordre de la syntaxe.

Le modèle de coût associé à SCP est proposé par Rebeuf [40]. Chaque chemin dans le graphe représente une exécution possible du programme. Le coût d'une exécution est déterminé de manière symbolique par le coût du chemin le plus long dans le graphe pour un contexte initial donné. Contrairement aux modèles précédents, le coût dépend donc de la valeur des données en entrée et pas uniquement de leur taille.

5.5.2 Remarques

Ce modèle est basé sur la syntaxe du langage qui donne le schéma de synchronisation et l'ordre d'exécution des instructions. Cela limite son utilisation aux programmes écrits dans ce langage et impose un schéma de synchronisation. De plus, ce langage est de type SPMD ce qui ne facilite pas le développement et la maintenance d'applications hétérogènes. Ce modèle possède comme élément de base l'instruction, or les applications de réalité virtuelle se placent à un plus gros grain.

5.6 Conclusion

Les modèles que nous venons de présenter présentent de nombreuses limitations qui ne permettent pas d'envisager leur utilisation dans le cadre d'applications de réalité virtuelle sur grappes de PC.

L'architecture considérée est généralement constituée de noeuds identiques, comme dans les approches BSP et LogP, alors que nous avons montré que les grappes de PC pour la réalité virtuelle sont généralement hétérogènes. Ensuite certains modèles, tels que BSP, imposent l'utilisation d'un schéma de programmation synchrone qui ne correspond pas au comportement d'une application de réalité virtuelle qui possède des codes fonctionnant à des fréquences très différentes. Enfin, une approche telle que SCP impose un langage de programmation ainsi qu'une structure SPMD qui ne se prête pas au développement de codes de réalité virtuelle.

Nous proposons donc de définir un nouveau modèle de performance. Celui-ci doit être capable de modéliser les architectures de type grappes de PC hétérogènes en capturant les principaux paramètres de celles-ci (puissance des processeurs, capacités et topologies des réseaux) afin de fournir des performances réalistes. La modélisation de l'application doit également intégrer les schémas de communication et de synchronisation afin de pouvoir déterminer précisément leurs effets sur les performances. Finalement ce modèle doit tenir compte du placement de l'application sur les performances.

Deuxième partie

Modèle pour l'évaluation des performances

Chapitre 6

Modélisation de l'architecture matérielle et logicielle

Sommaire

Le modèle de performance présenté dans cette partie a fait l'objet de publications à NPC 2007 [34] ainsi qu'à PDPTA 2007 [33]

La performance d'une application séquentielle dépend des instructions qui la composent et de la vitesse d'exécution du processeur qui l'exécute. Il est dans ce cas relativement simple de déterminer la performance en analysant l'enchaînement des instructions et la vitesse de traitement de celles-ci par le processeur. La performance d'un programme séquentiel varie donc uniquement en fonction de la puissance offerte par la machine. Si on considère une application parallèle exécutée sur une architecture à mémoire partagée, cette tâche devient plus complexe. La performance dépend alors de celle de chacune des parties ainsi que des synchronisations entre elles nécessaires pour résoudre les problèmes d'accés concurrents à la mémoire partagée.

Considérons maintenant une application distribuée sur une architecture à mémoire répartie de type grappe de PC. L'évaluation de performance de codes hétérogènes distribués sur des architectures également hétérogène nécessite de prendre en compte de nombreux facteurs.

D'une part, nous disposons d'une architecture distribuée qui possède des capacités de calcul, de stockage et de communication que l'on souhaite exploiter. D'autre part, nous avons une application distribuée composée de différentes parties qui communiquent et se synchronisent. La performance de l'application dépend de l'architecture, de la performance des parties de l'application, du couplage et des communication entre celles-ci, ainsi que du placement de ces parties sur les différents noeuds de la grappe.

De plus, une architecture distribuée de type grappe de PC peut être composée de noeuds hétérogènes et les performances d'un composant peuvent donc varier d'un noeud à l'autre. De la même manière il est possible d'utiliser plusieurs réseaux d'interconnexion possédant des performances et des topologies différentes. Les temps de communication peuvent donc varier en fonction du placement des connexions sur les différents liens réseaux.

Afin de vérifier la performance d'une application distribuée il est possible de générer des placements et de les exécuter sur l'architecture cible afin de pouvoir les comparer. Cette approche nécessite un accés physique à l'architecture et monopolise son utilisation pendant la durée des tests. Nous proposons donc d'utiliser un modèle capable de prédire les performances d'une application placée afin d'éviter une longue et laborieuse phase de tests. De plus un modèle permettrait de tester les performances d'une application sur une architecture virtuelle afin de déterminer a priori la configuration nécessaire à l'obtention de performance pour une application donnée.

Dans cette partie nous nous attachons a l'évaluation de performance d'un placement donné d'une application sur une grappe de PC. Cela signifie que pour chaque module et chaque connexion nous connaissons respectivement son placement sur un des noeuds de la grappe et le lien réseau emprunté.

Nous proposons donc dans un premier temps de modéliser les noeuds de la grappe ainsi que leurs interconnexions par différents réseaux. Nous présentons ensuite la manière dont nous modélisons une application FlowVR ainsi que les informations additionnelles nécessaires à notre modèle. Finalement nous décrivons le placement des composants et des connexions d'une application respectivement sur les noeuds et liens réseaux de notre grappe de PC.

6.1 Modélisation de la grappe de PC

Une grappe de PC est constituée d'un ensemble de noeuds, noté $Nodes$, interconnectés à l'aide d'un ensemble de réseaux, noté $Networks$. Par exemple la figure 6.1 décrit une grappe constituée de quatres noeuds interconnectés par deux réseaux. Nous avons dans ce cas $Nodes = \{node_1, node_2, node_3, node_4\}$ et $Networks = \{net_0, net_1\}$.

Ces différents éléments peuvent être hétérogènes. Ainsi la grappe peut contenir des noeuds mono-processeurs ou SMP, possédant des architectures 32 ou 64bits et des fréquences différentes. Les cartes graphiques influent également sur les performances pour la visualisation ou peuvent servir à assister le processeur dans le cas d'une utilisation GPGPU (General-Purpose computation on GPUs). La performance d'un composant est donc différente suivant son placement.

De même, on peut disposer de réseaux hétérogènes comme par exemple des réseaux Ethernet offrant des débits de 100 ou 1000Mb/s, ou des réseaux haute-performance de type Myrinet. Ces réseaux peuvent interconnecter l'ensemble de la grappe ou juste un sous-ensemble. La performance des communications va donc dépendre des caractéristiques du réseau utilisé.

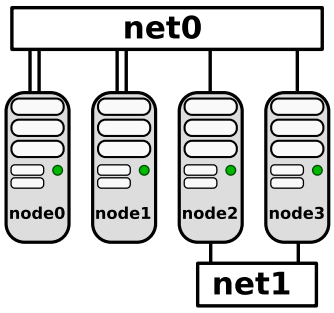

FIG. 6.1 – Exemple de topologie d'interconnexion réseau

Nous décrivons maintenant plus en détail la modélisation des noeuds et des réseaux d'une grappe.

6.1.1 Noeuds de la grappe

On considère que chaque noeud $n \in Nodes$ de la grappe possède un type $type_n$ et que les noeuds physiquement identiques sont associés au même type. Ainsi, un composant a les mêmes performances sur les noeuds d'un même type et peut avoir des performances différentes sur des noeuds de type distincts. Cette modélisation permet de s'abstraire du type de processeur utilisé car la performance d'un composant peut dépendre d'autres périphériques. Par exemple la performance d'un composant utilisé pour l'affichage va dépendre plus fortement de la carte graphique que du processeur central. On choisit donc cette approche plus générale afin d'englober l'ensemble du matériel. Dans notre exemple, figure 6.1, nous avons deux types de noeuds différents qui sont décrits dans le tableau 6.1.

noeud	type
$node0$	$type0$
$node1$	$type0$
$node2$	$type1$
$node3$	$type1$

TAB. 6.1 – Différents types de noeuds

Au cours de notre étude nous avons également besoin de considérer les différents processeurs d'une machine (SMP et/ou multi-coeurs) pour étudier les phénomènes de concurrence et le placement des composants sur les processeurs. Nous associons donc à chaque noeud n une liste de processeurs notée $CPUs_n$. Dans notre exemple, si le noeud $node0$ possède deux processeurs alors $CPUs_{node0} = \{cpu^0_{node0}, cpu^1_{node0}\}$. Notons que les architectures SMP et multi-coeurs supposent que les processeurs d'un même noeud soient tous identiques.

6.1.2 Réseaux de la grappe

Chaque réseau $net \in Networks$ est doté d'une bande passante BW_{net} exprimée en mégabits par seconde, ainsi que d'une latence L_{net} exprimée en microsecondes. Nous considérons que les réseaux utilisés disposent de connexions point-à-point en full-duplex et que la gestion des communications est prise intégralement en charge par le processeur de la carte réseau afin de ne pas générer de surcharge sur les processeurs centraux. Ces hypothèses

semblent raisonnables si l'on considère les caractéristiques et le fonctionnement des réseaux actuels. Nous définissons également un réseau local afin de modéliser les communications entre des composants placés sur un même noeud. Dans l'implémentation de FlowVR les communications locales se font par l'intermédiaire de mémoire partagée. Les composants s'échangent donc juste des pointeurs vers des zones mémoires. Nous supposons donc que les communications locales ont un coût nul et que nous avons ainsi $BW_{local} = \infty$ et $L_{local} = 0$.

Afin de connecter les noeuds aux réseaux nous définissons un ensemble appelé $Netlinks$ de liens réseaux. Chaque lien $netlink \in Netlinks$ connecte un noeud $n \in Nodes$ à un réseau $net \in Networks$. Un lien $netlink$ représente une connexion physique d'un noeud à un switch par un cable réseau. On peut ainsi modéliser l'ensemble $Netlinks$ des liens par une table associant un lien $netlink$ à un noeud et à un réseau. Pour chaque lien $netlink \in Netlinks$, nous utilisons la notation $node_{netlink}$ pour désigner le noeud connecté par $netlink$, et nous notons $net_{netlink}$ le réseau associé au lien.

net	$\mathbf{BW_{net}}$ (Mb/s)	$\mathbf{L_{net}}$ (μs)
$local$	∞	0
$net0$	80	47
$net1$	200	3.5

TAB. 6.2 – Description de l'interconnexion de la grappe

Par exemple, le cluster représenté par la figure 6.1 comporte deux réseaux $net0$ et $net1$ dont les caractéristiques sont décrites dans le tableau 6.2. Les liens réseaux sont définis dans le tableau 6.3. Ainsi, tous les noeuds sont reliés à un même réseau $net0$, les noeuds $node0$ et $node1$ possèdent deux liens vers ce réseau alors que les noeuds $node2$ et $node3$ possèdent une seconde connexion vers un autre réseau $net1$.

netlink	$\mathbf{node_{netlink}}$	$\mathbf{net_{netlink}}$
$netlink0$	$node0$	$net0$
$netlink1$	$node0$	$net0$
$netlink2$	$node1$	$net0$
$netlink3$	$node1$	$net0$
$netlink4$	$node2$	$net0$
$netlink5$	$node2$	$net1$
$netlink6$	$node3$	$net0$
$netlink7$	$node3$	$net1$

TAB. 6.3 – Description de l'interconnexion de la grappe

Cette modélisation permet de spécifier des réseaux hétérogènes et également de définir précisément la topologie de chaque réseau. Dans notre étude nous supposons que la capacité de l'adaptateur réseau sur un noeud est égale à celle du réseau ou supérieure. On s'aligne alors dans ce dernier cas sur la capacité du réseau. Notons que nous ne pouvons pas modéliser le cas où une carte réseau possède une capacité inférieure à celle du réseau, par exemple lorsqu'une carte Ethernet 100Mb/s est connectée à un switch Ethernet 1Gb/s. Une modélisation plus poussée intégrant les cartes réseaux permettrait de considérer ce cas mais compte tenu de notre architecture matérielle ainsi que pour des raisons de simplicité nous n'avons pas retenu ce choix.

6.2 Modélisation de l'application

L'application peut être modélisée sous forme d'un graphe $G_{appl}(V, E)$ dont l'ensemble V des noeuds contient les composants : modules, filtres ou synchroniseurs, et l'ensemble E des arcs contient les connexions entre les ports de ces composants. L'ensemble des modules est noté $Modules$, c'est un sous-ensemble de V. De la même manière nous notons respectivement $Filters$ et $Synchronizers$ l'ensemble des filtres et des synchroniseurs. Notons que :

$$\begin{aligned} Modules \cup Filters \quad &\cup \quad Synchronizers = V \\ Modules \cap Filters = Modules \quad &\cap \quad Synchronizers = Filters \cap Synchronizers = \emptyset \end{aligned}$$

L'ensemble des connexions est défini par E. La figure 6.2 présente un graphe d'application que nous allons utiliser comme exemple dans cette partie. Cette application est composée des ensembles d'objets suivants :

- $Modules$=$\{module1, module2/0, module2/1, module3/0, module3/1\}$;
- $Filtres$=$\{broadcast0, broadcast1, broadcast2, gather, greedy/in/O, greedy/filter/0\}$;
- $Synchronizers$=$\{greedy/sync/0, greedy/sync/1\}$;
- E=$\{c_0, c_1, \ldots, c_{17}\}$.

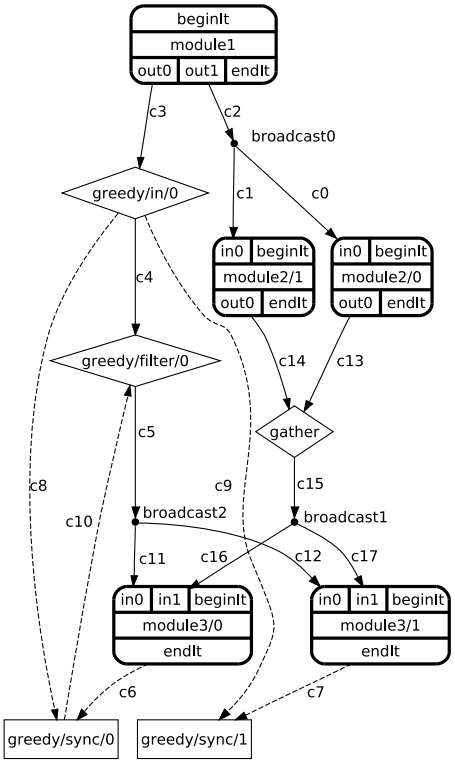

FIG. 6.2 – Exemple d'application FlowVR

Nous présentons maintenant une modélisation des composants de l'application ainsi que des connexions FlowVR.

6.2.1 Informations sur les modules

Pour effectuer son calcul, un module nécessite un certain temps pour s'exécuter qui dépend des performances du noeud sur lequel il est placé. Il génère également une occupation, aussi appelée charge, sur le processeur qui dépend des opérations d'entrées-sorties qu'il effectue. Enfin, pour une application donnée, chaque module émet par itération une quantité de données fixée.

Temps d'exécution et charge

Chaque module $m \in Modules$ sur un noeud cible $n \in Nodes$ est caractérisé par son temps d'exécution $T^{exec}_{m,n}$ et sa charge LD_mn. Le temps d'exécution correspond au temps nécessaire à un module pour effectuer le calcul d'une itération de la boucle FlowVR et intègre donc le temps de calcul pur ainsi que le temps nécessaire pour effectuer les opérations d'entrées/sorties. Ce temps correspond donc au temps écoulé entre la sortie de la fonction *wait()* et l'appel suivant à cette fonction.

La charge indique le pourcentage d'utilisation du processeur par le module. Ces valeurs dépendent de la performance du processeur du noeud ainsi que des controlleurs d'entrées/sorties. On peut donc avoir des valeurs différentes de $T^{exec}_{m,n}$ et de LD_mn en fonction du type $type_n$ du noeud n considéré. Pour chaque module, nous pouvons représenter ces valeurs sous forme d'un tableau en fonction du type de noeud considéré comme nous le montrons dans le tableau 6.4.

$\mathbf{type_n}$	$\mathbf{T^{exec}_{m,n}}$ (ms)	$\mathbf{LD_m n}$
$type_0$	20	60
$type_1$	15	55

TAB. 6.4 – Temps d'execution et charge d'un module m en fonction du type de noeud

Il existe plusieurs méthodes possibles afin de déterminer les valeurs de ces paramètres sur les différents noeuds d'une grappe. Une première méthode consiste à exécuter chaque module sur chaque noeud, indépendamment des autres modules de l'application, et de mesurer le temps d'exécution et la charge. Il est également possible d'évaluer ces valeurs sur une machine cible et de les extrapoler en fonction d'un facteur de performance défini entre chaque machine. Etant donné que FlowVR permet de spécifier les modules indépendamment de l'application et de les réutiliser, les temps d'exécution et les charges sont souvent déjà déterminées et disponibles. En fonction de ces valeurs, le développeur peut faire des choix de placement.

Dans cette partie de notre étude nous considérons un placement donné par le développeur. Pour chaque module m nous connaissons donc le noeud n associé et donc son type $type_n$. Les valeurs de $T^{exec}_{m,n}$ et de LD_mn sont donc fixées par ce choix. Afin de simplifier l'écriture nous utiliserons les notations T^{exec}_m et de LD_m lors de l'étude d'un placement donné.

Volume de données émis

Chaque module m possède une liste de ports de sorties notée $ports^{out}_m$. Le module $module1$ de la figure 6.2 possède ainsi trois ports de sorties :

$$ports^{out}_{module1} = \{out0, out1, endIt\}$$

Chacun de ses ports $p \in ports_m^{out}$, émet par itération un volume de données Vol_m^p fixé pour l'application considérée. Cette donnée nous permet de déterminer la quantité de données envoyée à chaque itération à travers les connexions.

Nous étudions maintenant les filtres et synchroniseurs qui peuvent respectivement influer sur la taille des messages et sur leur fréquence d'émission.

6.2.2 Comportements des filtres et synchroniseurs

Les filtres réalisent des opérations de transformation sur les messages. Par exemple le filtre *gather*, utilisé dans l'application décrite figure 6.2, récupère des messages provenant de plusieurs modules et les combine en un seul message les regroupant. Le filtre *scatter* réalise quant à lui l'opération inverse. Ces opérations sont relativement simples et peu coûteuses par rapport aux opérations effectuées par les modules car il s'agit uniquement d'opérations sur la mémoire. De plus la fréquence de fonctionnement des filtres est liée à celles des modules qui leurs sont connectés. Ainsi le filtre *scatter* traite les messages à la fréquence de l'objet auquel son port d'entrée est relié. Le filtre *greedy* quant à lui fonctionne à la fréquence de l'objet le plus rapide auquel il est connecté. Dans notre approche, nous considérons donc que le fonctionnement des filtres usuels engendre une charge négligeable sur les noeuds de la grappe. Néanmoins, dans le cas de filtres plus complexes dont la charge ne serait pas négligeable, il est possible de les intégrer dans notre approche en les considérant comme des modules, c'est à dire en leur attribuant un temps d'exécution et une charge.

Par contre les filtres ont une grande influence sur le volume des données qu'ils transfèrent. Nous devons donc tenir compte du type de filtre utilisé afin de déterminer les volumes de données émis vers des objets se trouvant sur d'autres noeuds. Par exemple un filtre *broadcast* réplique un message reçu sur son port d'entrée vers ses ports de sorties alors qu'un filtre *gather* récupère les messages sur ses ports d'entrée et les concatène en un seul message.

Les synchroniseurs implémentent les politiques de couplage, leurs décisions sont basées uniquement sur les valeurs des estampilles. Comme pour les filtres, la fréquence de fonctionnement d'un synchroniseur dépend de celle des objets qui y sont connectés. De plus, les calculs mis en oeuvre pour la prise de décision sont très simples, la charge engendrée sur le processeur est donc négligeable par rapport aux calculs effectués par les autres modules. Par contre la politique de synchronisation mise en place par les synchroniseurs peut avoir une forte influence sur les performance. Par exemple un synchroniseur *maxfrequency* permet de limiter la frequence d'un module. Notre approche doit donc considérer les politiques mises en oeuvre par les synchroniseurs.

6.2.3 Connexions entre composants

Une connexion FlowVR relie les ports de deux composants, un émetteur et un récepteur, à travers un canal FIFO. Chaque connexion $c \in E$ possède donc un composant source, noté src_c et un composant destination noté $dest_c$. Le volume de communication envoyé à travers une connexion $c \in E$ est noté Vol_c et dépend du volume de données émis par src_c sur son port de sortie connecté à c. Nous nous intéressons maintenant aux caractéristiques des composants qui vont déterminer pour chaque connexion c le volume Vol_c correspondant :

- Si $src_c \in Modules$ alors Vol_c dépend du volume $Vol_{src_c}^p$ du port $p \in ports_{src_c}^{out}$, auquel c est connectée : $Vol_c = Vol_{src_c}^p$.
- Si $src_c \in Filters$ alors Vol_c dépend du type de filtre utilisé.

  - Dans le cas d'un filtre de type $broadcast$ possédant une connexion c_0 en entrée et deux connexions c_1, c_2 en sortie, la quantité de données émise à travers c_1 et c_2 est égale à la quantité de données envoyée par c_0. Nous avons donc $Vol_{c_1} = Vol_{c_2} = Vol_{c_0}$.
  - Si on considère un filtre $merge$ binaire avec deux connexions entrantes c_0 et c_1 et une connexion en sortie c_2, alors $Vol_{c_2} = Vol_{c_0} + Vol_{c_1}$.
  - Dans le cas d'un filtre $scatter$ binaire avec une connexion c_0 en entrée et deux connexions en sortie c_1 et c_2 nous avons $Vol_{c_1} = Vol_{c_2} = Vol_{c_0}/2$
  - La transformation effectuée par d'autres filtres doit être explicitée de la même manière que précedemment.
- Si $src_c \in Synchronizers$, nous choisissons dans notre approche de considérer que les volumes de données échangés avec les synchroniseurs sont négligeables. En effet, les estampilles échangées par les synchroniseurs représentent un faible volume de données comparé à celui engendré par les messages des modules. Pour chaque connexion c telle que $src_c \in Synchronizers$ ou $dest_c \in Synchronizer$ nous définissons donc $Vol_c = 0$.

Initialement, la seule donnée disponible sur les volumes de communication est fournie par les modules et la taille des messages qu'ils émettent sur leurs ports de sorties. Afin d'évaluer les valeurs de Vol_c pour chaque connexion c nous commençons donc par les connexions telles que $src_c \in Modules$. Dans notre exemple, figure 6.2, nous évaluons premièrement les connexions en sortie des modules, c'est à dire c_2, c_3, c_{13} et c_{14} :

$$\begin{aligned} Vol_{c_2} &= Vol^{out1}_{module1} \\ Vol_{c_3} &= Vol^{out0}_{module1} \\ Vol_{c_{13}} &= Vol^{out0}_{module2/0} \\ Vol_{c_{14}} &= Vol^{out0}_{module2/1} \end{aligned}$$

Puis nous propageons ces valeurs à travers le graphe de l'application en les transformant en fonction des filtres traversés. Après le filtre $gather$ nous obtenons ainsi :

$$Vol_{c_{15}} = Vol_{c_{13}} + Vol_{c_{14}} = Vol^{out0}_{module2/0} + Vol^{out0}_{module2/1}$$

Les filtres $greedy$ ne modifient pas la taille des messages, nous avons donc :

$$Vol_{c_5} = Vol_{c_4} = Vol_{c_3} = Vol^{out0}_{module1}$$

Après les filtres de type $broadcast$ nous avons finalement :

$$\begin{aligned} Vol_{c_1} &= Vol_{c_0} = Vol_{c_2} = Vol^{out1}_{module1} \\ Vol_{c_{16}} &= Vol_{c_{17}} = Vol_{c_{15}} = Vol^{out0}_{module2/0} + Vol^{out0}_{module2/1} \\ Vol_{c_{11}} &= Vol_{c_{12}} = Vol_{c_5} = Vol^{out0}_{module1} \end{aligned}$$

6.3 Modélisation du placement

Nous disposons de modélisations pour notre architecture matérielle ainsi que pour notre application. Il nous reste maintenant à définir un placement, c'est à dire à associer chaque composant de l'application à un noeud, et chaque connexion distante à deux liens d'un même réseau si les composants sont placés sur des noeuds distincts.

Le placement des composants à un impact très important sur la performance de l'application. Ainsi, si deux modules m_0 et m_1 sont placés sur une même machine $node_0$, c'est à dire si $node_{m_0} = node_{m_1} = node_0$, alors ils peuvent être en concurrence sur le même processeur et voir leur performance modifiée. Si nous considérons l'application représentée figure 6.2, on peut définir par exemple le placement des composants comme décrit dans le tableau 6.5.

composant	noeud
$module1$	$node0$
$module2/0$	$node0$
$module2/1$	$node1$
$module3/0$	$node2$
$module3/1$	$node3$
$broadcast0$	$node0$
$broadcast1$	$node1$
$broadcast2$	$node2$
$gather$	$node1$
$greedy/in$	$node0$
$greedy/filter$	$node0$
$greedy/sync/0$	$node0$
$greedy/sync/1$	$node0$

TAB. 6.5 – Placement des modules, filtres et synchroniseurs

Le placement des composants conditionne ensuite le placement des connexions. Si deux composants communiquent, c'est à dire s'ils sont reliés par un arc dans le graphe de l'application, et s'ils sont placés sur des noeuds différents $node_0$ et $node_1$ alors il faut choisir un réseau commun aux deux noeuds. Si un tel réseau n'existe pas alors il faut reconsidérer le placement car il n'est pas possible de faire communiquer les deux modules. Une fois le réseau sélectionné pour une connexion c, par exemple $net_c = net_0$, il reste à associer la connexion à un couple de liens qui connectent $node_0$ et $node_1$ à net_0. S'ils sont placés sur un même noeud alors la connexion est locale $net_c = local$.

Dans notre précédent exemple, tableau 6.3, on peut choisir par exemple les couples $(netlink_0, netlink_2)$, $(netlink_0, netlink_3)$, $(netlink_1, netlink_2)$, ou bien encore $(netlink_1, netlink_3)$. Un exemple de placement de quelques connexions de l'application décrite figure 6.2 est présenté tableau 6.6.

6.4 Critères de performance

Nous présentons maintenant les différents indices de performances que notre modèle va permettre de déterminer. Tout d'abord nous avons besoin de connaître le comportement de chaque module qui varie en fonction de la concurrence et de la synchronisation avec les autres

connexion	réseau	placement
c_0	$local$	-
c_1	$net0$	$(netlink_1, netlink_3)$
c_2	$local$	-

TAB. 6.6 – Exemple de placement des connexions

modules. Cela nous permet de déterminer la fréquence des modules. A partir du volume de données transmis par itération sur chaque connexion et de la fréquence nous serons ensuite capable de calculer le volume de données échangé sur les différents liens des réseau et vérifier si ce volume ne dépasse pas les capacités matérielles. Il est ensuite possible de déterminer la latence entre les composants, c'est à dire le temps nécessaire à une information pour se propager d'un composant émetteur à un composant récepteur. Les applications de RV imposent des valeurs minimales pour la fréquence et la latence. Ainsi la fréquence d'un module de visualisation doit être supérieure à 15Hz pour assurer la fluidité de l'affichage. De même le temps de latence entre un module d'interaction et de visualisation doit être infèrieure à 0.1 seconde afin d'assurer l'interactivité de l'application. Si ces critères ne sont pas respectés alors le placement n'est pas satisfaisant et il faut en considérer un autre. Nous souhaitons également étudier l'utilisation des processeurs afin, par exemple, de pouvoir comparer plusieurs placements aux performances similaires et déterminer celui qui utilisera le moins de ressources.

6.4.1 Performance des modules

Notre modèle a pour objectif de déterminer, pour un placement donné, deux indices de performances pour chaque module m de l'application qui sont le temps d'itération, noté T_m^{it}, et le temps d'exécution concurrente, noté T_m^{conc}. Ces deux indices nous permettrons ensuite de déterminer la fréquence de chaque module.

Temps d'exécution concurrente Le temps d'exécution concurrente est le temps nécessaire à un module pour effectuer son calcul lorsqu'il se trouve en concurrence avec d'autres modules. On peut le définir comme le temps écoulé entre la sortie de la fonction *wait()* et l'appel suivant à cette fonction. Il dépend de la répartition de la charge processeur entre les différents modules concurrents. Cette allocation est généralement effectuée par l'ordonnanceur du système d'exploitation mais peut être également définie par le développeur. Notons que nous avons dans tous les cas $T_m^{conc} \geq T_m^{exec}$ puisqu'on ne peut pas attribuer plus de 100% du processeur à un module.

Temps d'itération Le temps d'itération correspond au temps écoulé entre deux appels successifs à la méthode *wait()*. Contrairement au temps d'exécution concurrente il englobe également le temps d'attente dans la méthode *wait()*. Ce temps d'itération peut être affecté par le temps d'exécution concurrente du module et également par les synchronisations avec les autres modules qui vont influer sur le temps d'attente dans la méthode *wait()*. En effet pour qu'un module puisse sortir de la fonction *wait()* il est nécessaire que chacun de ses ports d'entrées ait reçu un message. Nous avons donc besoin de connaître les performances des modules se trouvant en amont du flot de données reçu par un module.

Fréquence Le temps d'itération d'un module est une information très importante car il correspond à l'inverse de la fréquence de fonctionnement du module :

$$F_m = \frac{1}{T_m^{it}} \tag{6.1}$$

La fréquence est la mesure couramment utilisée pour exprimer la performance d'un périphérique haptique ou encore le rafraîchissement d'un dispositif d'affichage. On peut donc facilement comparer les performances des différents modules et les comparer aux capacités des matériels utilisés.

6.4.2 Utilisation des processeurs

Afin de déterminer T_m^{conc} nous avons besoin d'étudier la charge LD_m^c attribuée à chaque module sur un processeur par l'ordonnanceur du système d'exploitation. Cette information nous permettra de connaître la charge totale affectée à chaque processeur et donc de définir les taux d'utilisation et l'efficacité de notre placement. Le développeur peut ainsi choisir le placement qui minimise le nombre de noeuds afin de libérer des noeuds pour d'autres applications.

6.4.3 Performance des communications

Volumes de données échangés Nous souhaitons également déterminer les volumes de données transmis à travers les connexions FlowVR de notre application. Cela nous permet de vérifier que le volume de données transitant au travers des différents liens réseaux de la grappe ne dépasse pas les capacités physiques des réseaux.

Latence La latence exprime le temps de traitement nécessaire à une information pour être transmise à travers les différentes parties de l'application. Elle dépend du temps de traitement des composants de l'application ainsi que du temps de transmission des messages à travers le réseau. Les applications de RV nécessitent une faible latence afin d'assurer un fonctionnement interactif à l'utilisateur.

6.5 Conclusion

La composition et la structure de l'application sont définies par le modèle FlowVR. Nous avons ajouté à cela les informations sur les performances des différents composants. FlowVR facilite la réutilisation des modules, leurs informations de performance sont donc souvent déjà connues. Dans le cas contraire, FlowVR permet d'exécuter chaque module indépendamment pour évaluer ses performances.

Nous fournissons également une modélisation de l'architecture qui permet, contrairement à celles des modèles BSP et LogP, de considérer des grappes de PC hétérogènes, constitués de noeuds SMP, et interconnectées par des réseaux également hétérogènes de par leurs caractéristiques et leur topologie.

Nous disposons maintenant des modélisations de l'application, de l'architecture cible ainsi que du placement de l'application sur l'architecture. Nous pouvons donc passer à l'étude des effets des synchronisations, de la concurrence et des communications sur les performances afin de déterminer les différents critères que nous avons décrit précédemment.

Chapitre 7

Modèle pour les synchronisations

Sommaire

Les synchronisations entre composants sont nécessaires afin d'assurer la cohérence de l'application. Par exemple le rendu distribué d'une même scène sur différents écrans nécessite une synchronisation, appelée *swaplock*, entre les composants effectuant le rendu afin de garantir que les images d'une même scène sont affichées simultanément. Dans ce cas on parle de synchronisations explicites. Des synchronisations peuvent également apparaître lors de communications. Par exemple, dans un programme MPI, l'appel à la commande MPI_Recv est bloquant et impose donc une synchronisation avant de continuer à exécuter le code suivant cette commande.

Dans le modèle FlowVR, les synchronisations entre différents modules peuvent résulter de synchroniseurs placés par l'utilisateur ou alors des communications entre eux. En effet, les synchroniseurs sont placés par le développeur de manière explicite afin d'obtenir un comportement donné. Il est ainsi possible de limiter la fréquence de fonctionnement d'un module à l'aide d'un synchroniseur *maxfrequency* fournit en standard avec FlowVR. Le développeur peut également créer ses propres synchroniseurs à l'aide de l'API FlowVR, dans ce cas il doit définir l'effet de ceux-ci sur les fréquences des modules qui y sont connectés. Dans notre étude, nous nous intéressons aux synchroniseurs définis par FlowVR qui sont fréquemment utilisés dans les applications. Nous cherchons également à obtenir un modèle capable d'intégrer d'autres types de synchroniseurs. Les synchronisations peuvent également intervenir lorsque les modules communiquent. Si l'on considère un module dont les ports d'entrées sont connectés, l'appel

à la commande *wait()* est bloquant tant qu'un message n'est pas disponible sur chacun des ports d'entrées du module. Nous avons alors une synchronisation implicite du modules avec ses modules précédents.

Le modèle FlowVR permet de synchroniser les itérations des modules mais ne fournit pas la possibilité d'effectuer des communications ni des synchronisations à l'intérieur d'une même itération. Cette possibilité peut s'avérer utile si l'on considère par exemple un module de simulation de fluide répartie comme décrit figure 7.1. Chaque itération produit un nouvel état du fluide, mais le calcul de cet état nécessite des communications entre les différents sous-domaines afin d'assurer une cohérence au résultat. Il est alors possible d'utiliser d'autres librairies de communication telles que MPI afin de contourner cette limitation. Notons que dans ce cas le développeur doit alors préciser les communications ainsi que les synchronisations entre les instances d'un module de ce type. Ces communications peuvent être matérialisées par l'ajout d'arêtes dans le graphe FlowVR et les synchronisations entre ces instances peuvent également être représentées par l'ajout d'un synchroniseur comme le montre la figure 7.1.

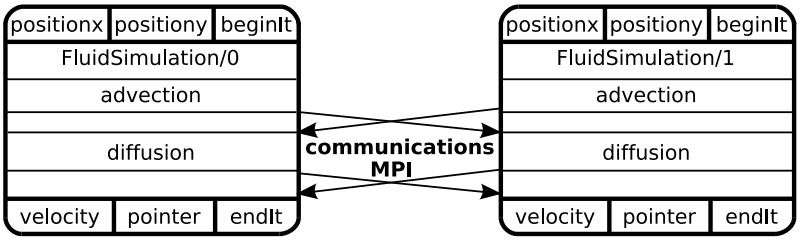

FIG. 7.1 – Module de simulation de fluide comportant des communications en dehors du schéma FlowVR

Dans cette section nous étudierons premièrement le comportement des modules qui ne sont pas synchronisés. Nous verrons ensuite comment sont affectées les performances des modules lorsqu'ils sont synchronisés soit pas des communications FlowVR, soit par des synchroniseurs. Finalement nous nous intéresserons aux communications et synchronisations en dehors du schéma FlowVR.

7.1 Modules sans ports d'entrées connectés

Certains modules ne disposent pas de ports d'entrées, ou plus exactement disposent uniquement du port d'entrée $beginIt$ défini par défaut. C'est par exemple le cas d'un module effectuant une lecture sur un périphérique (souris, un clavier ou caméra). On peut également désirer utiliser un module sans connecter ses ports d'entrées. Par exemple dans le cas d'une simulation disposant d'un port d'entrée pour l'interaction, il est possible de ne pas connecter ce port si on ne souhaite par interagir. Cette propriété permet de développer des modules facilement réutilisables dans des contextes différents.

Les modules ne possédant pas de ports d'entrées ou n'ayant pas de ports d'entrées connectés n'attendent jamais de nouveaux messages. L'appel à la fonction *wait()* dans ce cas ne provoque aucune attente. Dans ce cas seule la concurrence avec d'autres modules peut influencer ses performances. Le temps d'itération d'un module m de ce type dépend donc uniquement de

son temps d'exécution concurrente :

$$T_m^{it} = T_m^{conc} \tag{7.1}$$

Si de plus ce module m est seul sur le noeud où il est placé nous avons l'égalité suivante :

$$T_m^{it} = T_m^{exec}, \forall m_i \in Modules, m_i \neq m, node_{m_i} \neq node_m \tag{7.2}$$

7.2 Filtres greedy

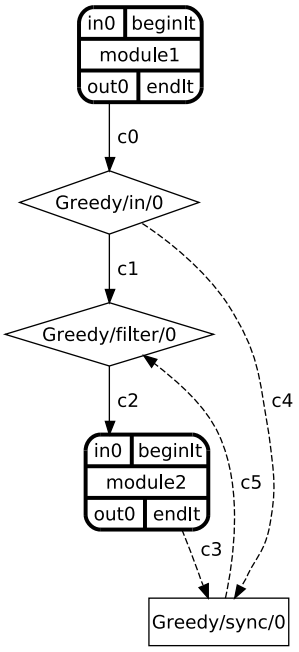

FIG. 7.2 – Schéma de connexion $greedy$ entre deux modules

Comme décrit à la section 4.4.1 , les filtres *greedy* permettent de faire communiquer des modules tout en leur permettant de fonctionner de manière asynchrone. De cette manière un module récepteur dispose toujours de la dernière information disponible en provenance de l'émetteur. On peut donc considérer qu'un module m recevant des données d'un filtre *greedy* se comporte comme un module ne disposant pas de connexions sur ses ports d'entrées et qu'il fonctionne donc toujours à sa fréquence maximale, son temps d'itération correspond donc à son temps d'exécution concurrent et est donc donné par l'équation 7.1.

7.3 Composantes synchrones

Les filtres *greedy* n'obligent pas les modules émetteurs et récepteurs qui y sont connectés à se synchroniser, nous pouvons restreindre notre étude au sous-graphe de l'application obtenu en supprimant du graphe de l'application G_{appl} les filtres *greedy* et les synchroniseurs associés. Nous notons le graphe obtenu $G_{sync}(V_{sync}, E_{sync})$. Dans certains cas le graphe G_{sync} n'est plus connexe, on obtient donc un ensemble de plusieurs composantes noté $Comps_{sync}(G_{sync})$. La figure 7.4 montre une transformation effectuée sur le graphe d'application présenté figure 7.3. Dans ce cas on obtient deux composantes connexes : $Comps_{sync}(G_{sync}) = \{C_1, C_2\}$

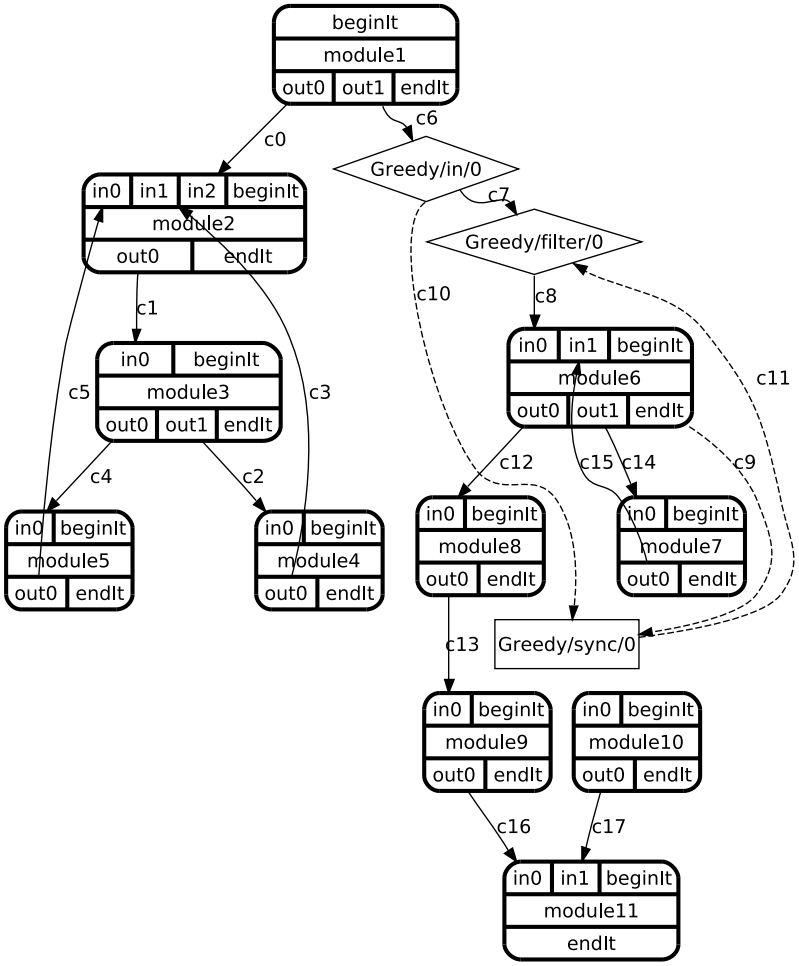

FIG. 7.3 – Graphe d'application G_{appl}

L'étape suivante consiste à étudier la configuration de G_{sync} ou de chacune de ses composantes si le graphe n'est plus connexe. Si on parcours la composante C_2 en simulant une exécution de ses modules à partir de $module6$ on s'aperçoit que lorsque $module7$ commence son itération alors $module6$ doit à son tour attendre la fin de l'exécution de $module7$ avant de commencer son itération suivante. Ces deux modules ne s'exécutent donc jamais simultanément. Nous pouvons généraliser ce raisonnement à tous les modules des cycles synchrones car, pour commencer son itération, chaque module doit attendre que les modules qui le précèdent dans le cycle se terminent. Les cycles synchrones impliquent donc un comportement différent par rapport aux autres modules. Afin de les étudier séparément nous proposons de les extraire du graphe G_{sync}.

La détection des cycles dans le graphe est obtenue par un parcours en profondeur en marquant les sommets parcourus. Notons que certains sommets peuvent appartenir à plusieurs cycles. Dans ce cas nous avons des cycles imbriqués. Dans notre exemple, figure 7.4, la composante C_1 comporte deux cycles contenant chacun trois modules. Le premier contient l'ensemble de modules E_1=$\{module2,\ module3,\ module4\}$ et le second l'ensemble E_2=$\{module2,\ module3,\ module5\}$. L'intersection de ces deux ensembles est non vide : $E_1 \cap E_2 = \{module2, module3\}$ donc on considère les modules de l'ensemble $E_1 \cup E_2$ comme appartenant à un même cycle imbriqué. La composante C_2 contient également un cycle com-

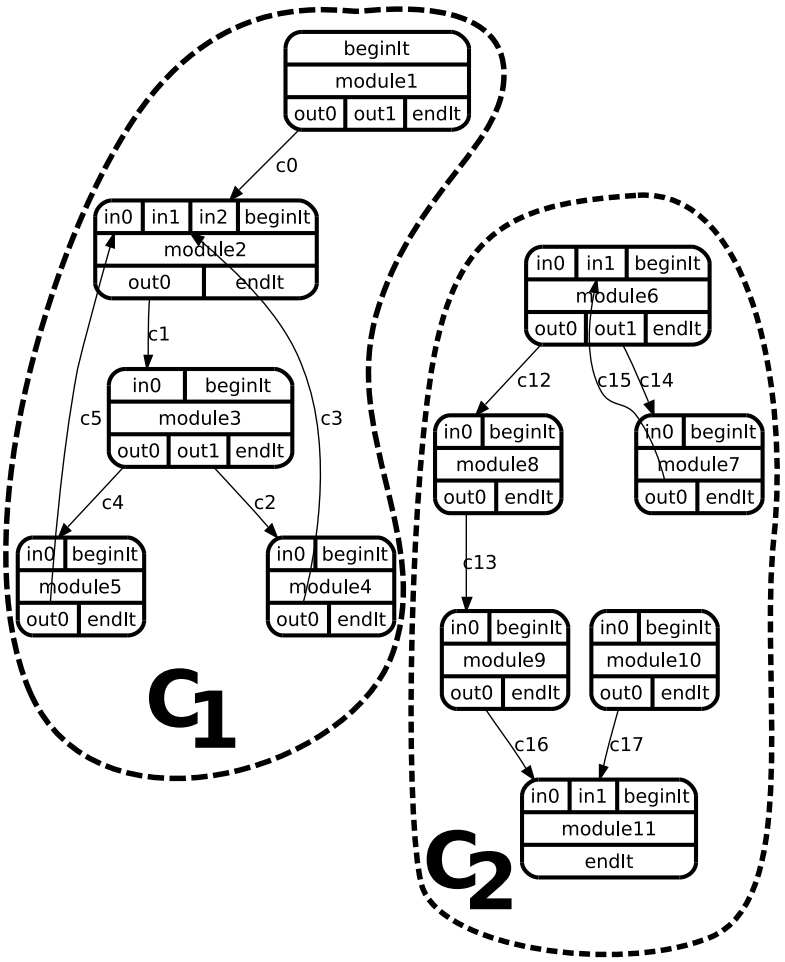

FIG. 7.4 – Graphe G_{sync} correspondant à l'application figure 7.3

portant les modules $module6$ et $module7$. Une fois les cycles, éventuellement imbriqués, identifiés on les retire du graphe, par définition, le sous-graphe restant ne contient alors plus que des graphes acycliques (DAG). Dans les composantes C_1 et C_2 il reste respectivement le module $module1$ et le DAG comportant les modules $module8$ et $module9$. Le résultat de ce découpage est présenté figure 7.5. Nous constituons donc deux ensembles à partir des ces configurations. Le premier ensemble est constitué des cycles synchrones, le deuxième de graphes acycliques. On étudie ensuite séparément ces deux ensembles en commençant par les DAG.

7.3.1 Graphes acycliques synchrones

Avant de considérer un graphe acyclique général dans son intégralité, nous nous intéressons premièrement à un graphe acyclique simple composé de seulement deux modules : un module et son module précédent, afin de déterminer l'influence de la communication synchrone sur le temps d'itération. Nous considérons ensuite une configuration comportant plusieurs modules précédents. Finalement nous étudions l'effet de la communication synchrone à travers le graphe acyclique complet.

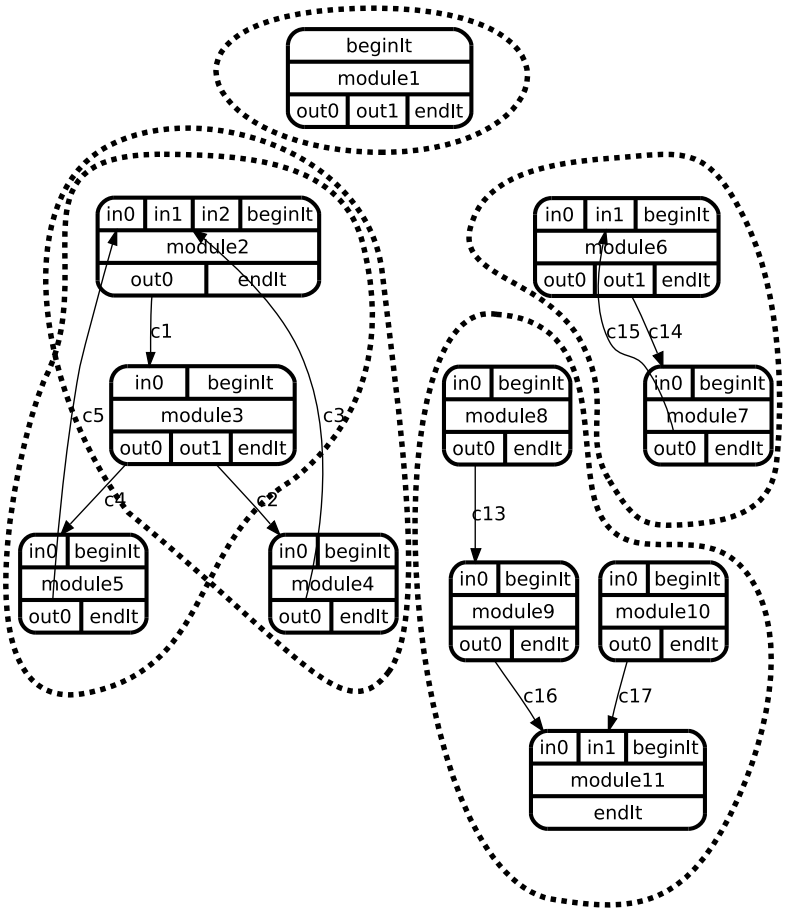

FIG. 7.5 – Décomposition de G_{sync} en DAG et cycles

Cas simple : 2 modules synchronisés

Lorsque deux modules m_1 et m_2 sont connectés par une connexion FIFO alors nous devons considérer deux cas différents. Soit le module émetteur m_1 est plus rapide que le module récepteur m_2, c'est à dire que $T^{it}_{m_1} > T^{conc}_{m_2}$. Soit le module émetteur est plus lent et nous avons $T^{it}_{m_1} \leq T^{conc}_{m_2}$.

Dans le premier cas le temps d'itération du module dépend de son temps d'exécution concurrent et nous avons donc $T^{it}_{m_2} = T^{conc}_{m_2}$. Mais le module m_2 n'est pas capable de traiter les messages à la fréquence imposée par m_1 car il est plus lent. Les messages de m_1 s'accumulent alors dans le tampon de réception du démon se trouvant sur le même noeud que m_2 jusqu'à ce que le tampon soit plein et que le démon génère une erreur d'allocation. Nous sommes alors capables d'indiquer au développeur que l'application provoquera une erreur lors de son exécution et de localiser précisément la cause de cette erreur. Il est alors possible de résoudre ce problème en ralentissant le module émetteur m_1 ou en accélérant le module m_2, par exemple en le plaçant sur un noeud plus performant pour diminuer $T^{exec}_{m_2}$. Le développeur peut aussi choisir de remplacer la connexion FIFO par une connexion *greedy*.

Dans le second cas le module récepteur m_2 est plus rapide, cependant sa méthode *wait()* l'oblige à attendre les messages en provenance de m_1. Etant donné qu'un nouveau message

est envoyé à chaque itération de m_1, le temps d'itération de m_2 doit s'aligner sur celui de m_1. Nous avons donc dans ce cas $T^{it}_{m_2} = T^{it}_{m_1}$.

Dans le cas de deux modules synchronisés, m_1 émettant vers m_2, nous pouvons donc exprimer le temps d'itération de m_2 de la manière suivante :

$$T^{it}_{m_2} = max(T^{conc}_{m_2}, T^{it}_{m_1}) \tag{7.3}$$

Modules précédents multiples

Si nous considérons maintenant, dans le cas plus général, un module m disposant de plusieurs modules précédents alors m doit attendre le module précédent le plus lent, celui avec le plus grand temps d'itération. Nous proposons donc la définition suivante, plus générale, du temps d'itération d'un module m :

$$T^{it}_{m} = max(T^{conc}_{m}, max_{\forall i \in IM_s(m)}(T^{it}_{i})) \tag{7.4}$$

Notons que, si toutefois m est plus lent que ses modules précédents alors nous avons toujours $T^{it}_{m} \leq T^{conc}_{m}$. Nous pouvons donc également prédire dans ce cas que les messages des modules précédents m sont s'accumuler et générer un dépassement de tampon. Mais dans le cas où m possède plusieurs modules précédents ce n'est pas le seul cas où un dépassement de tampon peut se produire. En effet si m est plus rapide que ses modules précédents alors il doit attendre le plus lent, par contre les autres modules précédents peuvent être eux plus rapide et leurs messages vont alors s'accumuler. Le seul moyen de prévenir tout dépassement de tampon est donc d'avoir le même temps d'itération pour tous les modules précédents.

Graphe acyclique complet

Avec les configurations précédentes, nous pouvons maintenant considérer le graphe acyclique dans son ensemble. Pour chaque module du graphe nous pouvons déterminer son temps d'itération en fonction de ses modules précédents qui dépendent eux-mêmes de leurs modules précédents. Nous pouvons remonter de modules précédents en modules précédents jusqu'à atteindre des modules sans modules précédents car notre graphe est acyclique. Ces modules correspondent aux racines du graphe. Si nous voulons éviter les erreurs de dépassement de tampon précédemment évoquées nous avons vu que chaque module doit avoir le même temps d'itération que ses modules précédents. En remontant le graphe jusqu'à ces racines nous arrivons donc à la conclusion tous les modules d'un graphe acyclique doivent avoir le même temps d'itération et que celui-ci correspond à celui des ses racines.

Nous nous intéressons maintenant aux cycles synchrones qui peuvent être présents dans les composantes de G_{sync}.

7.3.2 Cycles synchrones

A l'intérieur d'un cycle synchrone non imbriqué chaque module attend des messages des modules le précèdent qui eux même attendent leurs modules précédents. Un cycle synchrone génère donc un blocage dans l'exécution de l'application car tous les modules du cycle s'attendent. Il faut donc qu'un des module du cycle commence par envoyer un premier message avant d'appeler la fonction *wait()* afin d'initialiser le cycle. Dans ce cas un seul module du cycle peut s'exécuter à la fois. En effet, une fois son exécution terminée un module doit attendre que tous les autres aient terminé. Cette information nous sera également utile pour l'optimisation du placement des modules sur les processeurs car des modules dans un cycle ne sont

jamais concurrents. Son temps d'itération dépend donc de son temps d'exécution concurrent, du temps d'itération des autres modules ainsi que des temps de communication nécessaires pour transmettre les messages entre les modules. Si aucun module du cycle $Cycle$ ne possède de module précédents en dehors du cycle, comme dans le cas du cycle représenté figure 7.5 comportant les modules $module_6$ et $module_7$, alors nous pouvons exprimer le temps d'itération de chaque module m_c dans $Cycle$ de la manière suivante :

$$T^{it}_{m_c} = \sum_{m \in Cycle} T^{conc}_{m} + \sum_{c} \left(\frac{Vol_c}{BW_{net_c})} + L_{net_c} \right) \tag{7.5}$$

Cela défini le temps d'itération propre du cycle noté T^{it}_{Cycle}. Si un module m_c de $Cycle$ possède un module précédent m en dehors du cycle alors le temps d'itération de m va influer sur celui du cycle :

$$T^{it}_{m_c} = max(T^{it}_{m}, T^{it}_{Cycle}) \tag{7.6}$$

Notons que si m est plus rapide que le cycle, c'est à dire si $T^{it}_{m} < T^{it}_{Cycle}$, alors nous pouvons prédire un dépassement de tampon à l'exécution de l'application car le cycle ne peut aller plus vite que son temps d'itération propre. De même si plusieurs modules du cycle possèdent des modules précédents en dehors du cycle alors ceux-ci doivent nécessairement posséder le même temps d'itération car les modules du cycle s'alignent sur le module le plus lent n'appartenant pas au cycle.

Cycles imbriqués Dans le cas de cycles imbriqués on distingue les modules communs à tous les cycles et les autres modules. Par exemple, dans le graphe représenté à la figure 7.5 nous avons deux cycles imbriqués comportant les modules $module_2$, $module_3$, $module_4$ et $module_5$. Les modules $module_2$ et $module_3$ sont communs aux deux cycles, tandis que les modules $module_4$ et $module_5$ ne font partie que d'un cycle. Les modules communs à tous les cycles imbriqués ne peuvent s'exécuter simultanément avec les autres modules du cycle. Pour les autres modules, soit ils appartiennent à une branche d'un même cycle, soit ils appartiennent à des branches différentes. Dans le premier cas, ils ne s'exécutent jamais en même temps que les modules de la même branche, par contre les modules placés dans deux branches distinctes, comme les modules $module_4$ et $module_5$ de notre exemple, peuvent s'exécuter simultanément. Pour calculer le temps d'itération on doit donc considérer le cycle le plus long parmi les cycles imbriqués, c'est à dire celui dont la valeur obtenue par l'équation 7.5 est la plus grande.

7.3.3 Modules et cycles prédécesseurs

Nous sommes maintenant capables d'exprimer le temps d'itération propre d'un cycle et celui des modules d'un graphe acyclique pour chaque composante de G_{sync}. Nous pouvons donc maintenant réassembler ces configurations pour reformer chaque composante de G_{sync}.

Lors de cette reconstruction, les configurations voient certains de leurs modules reliés à des modules d'autres configurations. Afin d'éviter les dépassements de tampon nous avons vu que les modules précédents d'un module devaient avoir le même temps d'itération. Comme le temps d'itération dans une configuration doit être le même, les modules de deux configurations connectées doivent donc avoir le même temps d'itération. Par reconstruction de la composante, les différentes configurations d'une même composante $Comp_{sync} \in Comps_{sync}(G_{sync})$ doivent donc posséder le même temps d'itération. En effet il existe au moins une connexion synchrone entre deux configurations d'une même composante sinon elles appartiendraient à

des composantes distinctes. Nous notons donc $T^{it}_{Comp_{sync}}$ le temps d'itération d'un composante $Comp_{sync} \in Comps_{sync}(G_{sync})$. Il nous reste maintenant à le déterminer.

Nous avons vu que le temps d'itération des modules ne disposant pas de ports d'entrées connectés ne dépend que du temps d'exécution concurrent de ceux-ci. De même pour le temps d'itération des modules dans les cycles synchrones lorsque ces modules ne disposent pas de modules précédents en dehors du cycle. Lorsque ceux-ci sont présents dans une composante $Comp_{sync} \in Comps_{sync}(G_{sync})$ alors $T^{it}_{Comp_{sync}}$ est déterminé en fonction de leur temps d'exécution concurrent respectivement par les équations 7.1 et 7.5. Nous nommons les modules dans $Comp_{sync}$ sans ports d'entrées les *prédécesseurs* de $Comp_{sync}$ et les cycles sans connexions extérieures les *cycles prédécesseurs* de $Comp_{sync}$. Il faut cependant respecter le fait que le temps d'itération doit être le même pour tous les modules de la composante. Donc si plusieurs prédécesseurs ou cycles prédécesseurs sont présents dans une même composante alors ils doivent posséder le même temps d'itération. Afin d'assurer un temps d'itération identique pour les prédécesseurs il est possible d'utiliser des synchroniseurs. Il est également possible, pour des modules parallèles basés sur MPI par exemple, d'avoir une synchronisation effectuée à l'aide d'une librairie parallèle. Lors du calcul des temps d'itération nous devons donc commencer par les prédécesseurs avant de considérer les autres modules dans les composantes.

On sait que le temps d'itération de chaque composante est déterminé par celui des prédécesseurs. Il reste donc à déterminer les temps d'exécution concurrents des prédécesseurs. On pourra ainsi vérifier que le temps d'exécution concurrent de chaque module est bien inférieur ou égal au temps d'itération afin d'éviter les dépassements de tampon.

7.4 Synchronisations explicites

7.4.1 Limitation de la fréquence

Le synchroniseur $MaxFrequency$ est utilisé pour limiter la fréquence d'un module. Il est par exemple utilisé pour limiter la consommation de ressource d'un module de visualisation. En effet on peut considérer qu'il n'est pas utile d'obtenir un taux de rafraîchissement supérieur à 60Hz car l'utilisateur n'est pas capable de percevoir de différence à l'affichage au-delà de cette fréquence. On diminue ainsi la charge sur le processeur, qui peut être disponible pour d'autres modules.

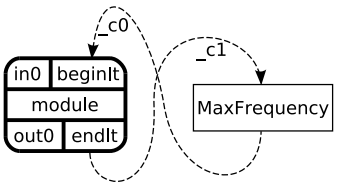

FIG. 7.6 – Filtre MaxFrequency connecté à un module

Ce synchroniseur se connecte sur les ports $BeginIt$ et $EndIt$ du module à contrôler comme décrit figure 7.4.1. A chaque fin d'itération, le module envoi un signal sur son port $EndIt$ à destination du synchroniseur. Celui-ci détermine le temps qui s'est écoulé entre ce signal et le signal envoyé à l'itération précédente et peut donc ainsi déterminer le temps d'itération et donc la fréquence du module. Si cette fréquence est inférieure à la limite fixée le synchroniseur

envoi immédiatement un signal sur le port $BeginIt$ du module afin qu'il commence une nouvelle itération. Si la fréquence est supérieure alors le synchroniseur produit une attente avant d'envoyer à nouveau un signal sur le port $BeginIt$ du module.

Le temps d'itération d'un module dépend de son temps d'exécution concurrente et du temps d'itération de ses modules précédents d'après l'équation 7.4. Si on branche le module sur un synchroniseur $MaxFrequency$ son temps d'itération dépend également de la valeur de la fréquence maximale imposée notée F_{max} et nous avons donc :

$$T_m^{it} = max(\frac{1}{F_{max}}, T_m^{conc}, max_{\forall i \in IM_m^s}(T_i^{it})) \tag{7.7}$$

7.4.2 Synchronisation de plusieurs modules

Il est possible de synchroniser plusieurs modules ensemble en utilisant un synchroniseur, comme décrit par la figure 7.7. On obtient dans ce cas une barrière de synchronisation.

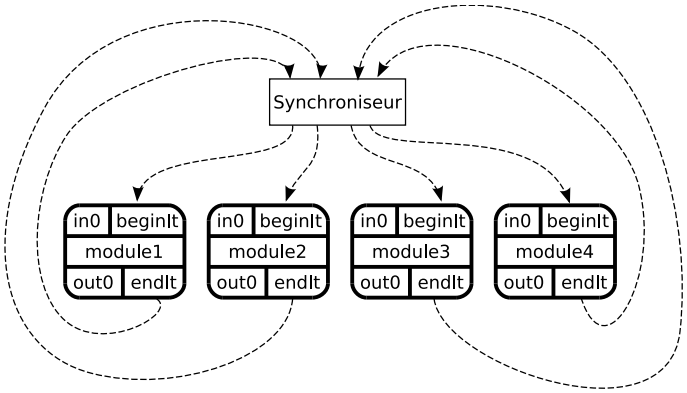

FIG. 7.7 – Synchronisation de plusieurs modules

Dans ce cas, pour un ensemble M_s de modules synchronisé, le temps d'itération de chaque module m est alignée sur celui du module le plus lent, c'est à dire celui avec le plus long temps d'exécution concurrente :

$$T_m^{it} = max(T_{m_i}^{conc}, m_i \in IM_s) \tag{7.8}$$

7.5 Synchronisations en dehors du schéma FlowVR

Certains modules peuvent être synchronisés en dehors du schéma FlowVR. Par exemple un module parallèle, basé sur MPI, instancié sur plusieurs machines peut effectuer des communications et des synchronisations au sein d'une itération. Dans ce cas le développeur doit préciser le comportement entre les instances de ce module. D'une part les communications peuvent être représentées de la même manière que des communications FlowVR, par l'ajout dans le graphe de l'application d'arêtes orientées associées à un volume de données transmis à chaque itération. D'autre part, les synchronisations peuvent être modélisées par l'ajout d'un synchroniseur FlowVR décrivant le comportement des modules. Un exemple de représentation pour un module de simulation parallèle est présenté figure 7.8.

De cette manière, on peut représenter un module parallélisé suivant un schéma BSP en utilisant une barrière de synchronisation comme décrit précédemment.

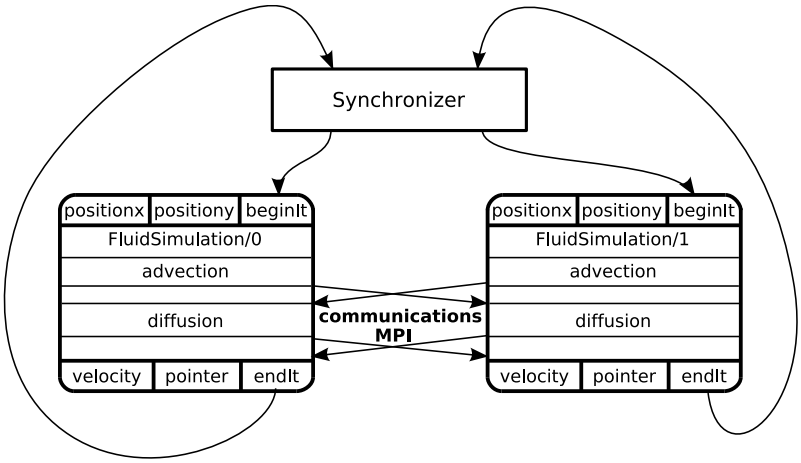

FIG. 7.8 – Modélisation des communications et synchronisation en-dehors du schéma FlowVR

7.6 Conclusion

L'étude du graphe G_{sync} nous a permis d'identifier les modules et cycles prédécesseurs qui déterminent le temps d'itération de tous les modules dans la même composante connexe de G_{sync}. La performance des ces modules dépend uniquement de leur temps d'exécution concurrent. Si ces modules ne sont pas en concurrence avec d'autres alors leur temps d'itération correspond à leur temps d'exécution concurrent et également à leur temps d'exécution. Pour un module prédécesseur pm nous avons donc, à partir de l'équation 7.1 :

$$T^{it}_{pm} = T^{conc}_{pm} = T^{exec}_{pm} \tag{7.9}$$

De la même manière, à partir de l'équation 7.5, si les modules d'un cycle prédécesseur $Cyclepred$ ne sont pas en concurrence avec d'autres modules, nous avons pour chaque module $pm_C \in Cyclepred$:

$$T^{it}_{pm_C} = \sum_{m \in Cyclepred} T^{exec}_{m} + \sum_{c \in Cyclepred} \left(\frac{Vol^{it}_{c}}{BW_{net_c}} + L_{net_c} \right) \tag{7.10}$$

Si les autres modules de l'application ne sont jamais en concurrence, par exemple si chaque module est associé à un processeur dédié, alors nous avons pour chacun d'eux :

$$T^{conc}_{m} = T^{exec}_{m} \tag{7.11}$$

Nous avons donc les informations suffisantes pour déterminer les volumes de communications échangés ainsi que la latence.

Dans les autres cas il nous reste maintenant à étudier les effets de la concurrence sur les modules afin de déterminer leurs temps d'exécution concurrent.

Chapitre 8

Modèles pour la concurrence

Sommaire

Un unique processeur ne peut pas exécuter plusieurs programmes simultanément. Si plusieurs programmes sont exécutés sur un même processeur alors le système d'exploitation les exécute partiellement et passe rapidement de l'un à l'autre ce qui donne l'illusion d'une exécution simultanée. Ce processus, appelé multitâche ou multiprogrammation [48], est géré par l'ordonnanceur du système d'exploitation qui tient compte de la priorité des processus afin de leur attribuer une allocation du processeur plus ou moins importante suivant la politique d'ordonnancement employée. Cette politique varie d'un système d'exploitation à un autre. Elle dépend également du type d'utilisation de la machine. Par exemple sur un poste de travail la priorité va être donnée aux processus interactifs au dépend des autres processus afin que l'utilisateur ne soit pas géné dans son interaction. Sur un serveur la politique est généralement inversée.

La politique d'ordonnancement va donc influer sur les performances des processus concurrents. Nous proposons donc dans une première partie de modéliser la politique d'ordonnancement du noyau Linux, qui est couramment utilisé sur les grappes de PC, afin de déterminer l'allocation des ressources processeurs effectuée par l'ordonnanceur et donc la performance des modules concurrents. Nous montrerons dans une seconde partie qu'il est également possible de définir une allocation des modules sur les processeurs à partir de l'étude globale des synchronisations de notre application. De cette manière on peut lier la politique d'ordonnancement à la performance que l'on souhaite obtenir.

8.1 Modélisation de la politique d'ordonnancement

Nous présentons premièrement l'ordonnanceur du noyau Linux ainsi que la politique associée. A partir de l'étude des critères sur lesquels se base cette politique d'ordonnancement, nous proposons ensuite un algorithme pour déterminer les temps d'exécution concurrent. Dans certains cas la politique de l'ordonnanceur peut conduire à des performances variables. Nous montrons qu'il est possible de prévoir ce comportement et d'y remédier dans certains cas.

8.1.1 Politique de l'ordonnanceur du noyau Linux

Nous décrivons l'ordonnanceur intégré jusqu'à la version 2.6.23 du noyau Linux [8, 21], appelé $O(1)$ car il opère en temps constant indépendamment du nombre de tâches à gérer.

Files de priorités

L'ordonnanceur est basé sur une structure constituée de 140 files d'attentes dont 100 sont utilisées pour les processus temps-réel et 40 pour les processus utilisateurs. Lorsqu'une tâche est créée, l'ordonnanceur lui attribue une priorité et la place dans la liste correspondante. Cette priorité détermine l'intervalle de temps que le processeur lui attribue pour son exécution avant de passer à une autre tâche. Ainsi, plus la tâche est prioritaire plus l'intervalle de temps alloué est important.

L'ordonnanceur commence par lancer l'exécution de la première tâche dans la liste la plus prioritaire. A la fin de l'intervalle de temps imparti, cette tâche est placée dans la file des tâches arrivées à expiration de leur intervalle de temps. Puis la tâche suivante dans la liste la plus prioritaire est lancée. Lorsque la liste est vide, l'ordonnanceur passe à la liste de priorité inférieure. Une fois toutes les listes traitées, les files actives et expirées sont interverties et le processus recommence. Le temps nécessaire pour trouver une nouvelle tâche à exécuter est donc dépendant du nombre de files d'attente et non pas du nombre de processus dans les listes d'attente ce qui permet d'obtenir une complexité en temps constant.

Notons que pour la gestion des machines SMP, chaque processeur possède sa propre file de priorité, contrairement à l'ordonnanceur du noyau 2.4 qui utilise une file commune. Cette solution permet d'éviter les accès concurrents à une file unique et le blocage des processeurs lorsque l'un d'entre eux y accède pour choisir la tâche à effectuer.

Attribution des priorités

La priorité d'un processus est attribuée en fonction du comportement de celui-ci. Plus un processus attend, c'est à dire effectue des opérations d'entrées-sorties, plus sa priorité va être élevée. A l'inverse un processus n'effectuant que du calcul va être pénalisé en lui attribuant un intervalle d'exécution court. Ce fonctionnement permet d'éviter qu'un processeur puisse s'approprier toute la ressource et pénalise ainsi les autres processus.

Un processus est ainsi plus prioritaire qu'un autre lorsque le temps qu'il passe à attendre est plus important, on favorise ainsi les processus les plus interactifs. Les processus n'effectuant pas d'opérations d'entrées-sorties, par exemple un code de calcul pur, sont donc les moins prioritaires et se retrouvent dans la liste de plus bas-niveau.

La priorité des processus est attribuée dynamiquement par l'ordonnanceur en fonction du temps d'attente effectué par chaque processus lors de son exécution précédente. Notons que seules les priorités des processus utilisateurs sont ajustées de la sorte.

Equilibrage sur architecture SMP

L'ordonnanceur permet un équilibrage de charge sur les différents processeurs d'une machine SMP. Une vérification des charges est effectuée régulièrement par l'ordonnanceur pour vérifier si les charges sont bien équilibrées. Si ce n'est pas le cas, des processus du processeur le plus chargé sont migrés dans la file d'attente d'autres processeurs.

8.1.2 Algorithme pour le calcul des temps d'exécution concurrent

La politique de l'ordonnanceur du noyau Linux est basée sur le temps d'attente des processus concurrents. Ainsi, plus un processus attend, plus l'ordonnanceur va le favoriser par rapport aux processus effectuant moins d'attente. Ce temps d'attente dépend des opérations d'entrées-sorties effectuées par un processus et peut être calculé en fonction de sa charge. Dans le cas des modules FlowVR, l'attente peut être également due à un blocage du module dans la méthode *wait()* lorsque celui-ci ne dispose pas de messages sur tous ses ports d'entrées.

Afin de déterminer les temps d'exécution de modules en concurrence, nous proposons de modéliser la politique de l'ordonnanceur Linux. Nous faisons l'hypothèse que les modules ont une charge et un temps d'exécution constants pendant la durée de l'exécution de l'application, par conséquent les priorités des modules sont supposées constantes et l'ordonnancement statique. Nous supposons également que l'architecture est dédiée à l'application et que les processus ne sont donc pas perturbés par ceux d'autres applications. Nous décrivons maintenant la démarche à appliquer sur un noeud afin de déterminer les temps d'exécution concurrente des modules placés sur celui-ci.

Calcul du temps d'attente

Le temps d'attente d'un module est également le temps de l'itération non utilisé pour le calcul. Dans ce cas, pour un module m nous pouvons déterminer son temps d'attente en fonction de son temps d'itération et de son temps d'exécution :

$$T_m^{I/O} = max_{\forall i \in IM_s(m)}(T_m^{exec}, T_i^{it}) - T_m^{exec} \times LD_m \tag{8.1}$$

Notons que si le module ne dispose pas de modules précédents nous avons $IM_s(m) = \emptyset$ et nous pouvons simplifier la formule précédente :

$$T_m^{I/O} = T_m^{exec} \times (1 - LD_m) \tag{8.2}$$

Nous sommes ensuite capable de comparer les temps d'attente des modules concurrents placés sur un noeud donné.

Classement des modules suivant leur priorité

Sur chaque noeud n nous pouvons ensuite ordonner les modules concurrents dans une liste notée l_n en commençant par le module avec le plus grand temps d'attente jusqu'à celui avec le plus petit temps d'attente. Notons que si plusieurs modules concurrents possèdent le même

temps d'attente nous pouvons les permuter et ainsi obtenir d'autres listes. Nous ne pouvons alors pas garantir l'ordre choisit par l'ordonnanceur. En pratique cela se traduit par un comportement variable des performances des modules. On observe ainsi soit un ordonnancement stable mais qui peut varier d'une exécution à une autre, soit une variation dynamique des performances pendant une même exécution. Dans ce cas nous sommes capable de détecter lorsque ce problème peut apparaître. On propose alors au développeur plusieurs solutions pour résoudre ce problème. Il est ainsi possible de déplacer les modules possédant le même temps d'attente sur des noeuds différents, ou de modifier leurs priorités par un réglage de la priorité laissée au soin du développeur. Dans ce dernier cas le développeur doit fournir la liste l_n correspondant à ses réglages.

Une fois la liste l_n définie, on attribue à chaque module une charge sur un processeur cible puis déterminer le temps d'exécution concurrente.

Attribution des charges et calcul des T^{conc}

Nous considérons un noeud n, disposant d'un ensemble de processeurs $CPUs_n$, sur lequel sont placés plusieurs modules ordonnés suivant la liste l_n. Notre objectif est d'attribuer à chaque module de la liste l_n un processeur dans $CPUs_n$ et une charge sur ce processeur. Pour cela nous utilisons l'algorithme 1 dont nous décrivons maintenant le principe.

Au départ les processeurs du noeud n ne sont pas chargés. Nous commençons par placer le premier module m de l_n (noté $m = head(l_n)$) qui est le plus prioritaire sur le premier processeur cpu_n^0 en lui attribuant une charge concurrente égale à sa charge : $LD_m^c = LD_m$ et nous augmentons d'autant la charge du processeur : $LD_{cpu_n^0} = LD_m^c$. Chaque module est ainsi placé sur le processeur le moins chargé, et il lui est attribué une partie de la charge restant libre du processeur correspondant au pourcentage représenté par sa charge.

Par exemple, considérons un module m tel que $T_m^{exec} = 10ms$ et $LD_m = 0,6$ que l'on souhaite placer sur un processeur cpu tel que $LD_{cpu} = 50$. La moitié du processeur est encore disponible pour placer le module. Le temps de calcul de ce module ($= T_m^{exec} \times LD_m$) va donc augmenter d'un facteur 2 ($= 1/(1 - 0.5)$). De manière générale nous définissons le temps d'exécution concurrent d'un module par l'équation suivante :

$$T_m^{conc} = \frac{T_m^{exec} \times LD_m}{1 - LD_p} + T_m^{exec}(1 - LD_m) \tag{8.3}$$

Notons que nous supposons que les temps nécessaires pour effectuer les opérations d'entrées-sorties sont indépendants de la charge du processeur, ce qui correspond à une prise en charge de ces opérations par des contrôleurs dédiés. Dans le cas où les modules effectuent le même type d'opérations, par exemple des accès disques, la concurrence peut introduire des temps d'accès plus longs.

Si le temps d'exécution concurrente du module dépasse le temps d'itération de ses prédécesseurs alors ce module devient plus lent que ses modules précédents, nous pouvons dans ce cas prévoir un dépassement de tampon à l'exécution. Dans le cas contraire nous calculons la charge qui lui sera allouée sur le processeur.

for all $cpu \in CPUs_n$ **do**
  $LD_{cpu} = 0$
end for
while $l(n) \neq \emptyset$ **do**
  $m = head(l(n))$
  $l(n) = tail(l(n))$
  $load = 1$
  for all $cpu \in CPUs(n)$ **do**
    if $LD_{cpu} < load$ **then**
      $p = cpu$
      $load = LD_{cpu}$
    end if
  end for
  if $load == 1$ **then**
    $exit$ "Plus de ressource processeur disponible"
  end if
  $T_m^{conc} = (T_m^{exec} \times LD_m)/(1 - LD_p) + T_m^{exec}(1 - LD_m)$
  if $T_m^{conc} \leq T_i^{it}, \forall i \in IM_s$ **then**
    $LD_m^c = (T_m^{exec} \times LD_m)/T_i^{it}$
  else
    exit "Problème de synchronisation : dépassement de tampon"
  end if
  $LD_p = LD_p + LD_m^c$
end while

Algorithme 1: Calcul de Tconc

8.1.3 Problème d'interdépendances

Le calcul de T_m^{conc}, pour un module $m \in Modules$, est déterminé par notre algorithme en fonction du temps d'attente $T_m^{I/O}$. Ce temps d'attente peut dépendre du temps d'itération des modules précédents de m d'après l'équation 8.1. Or le temps d'itération d'un module précédent peut dépendre également de son temps d'exécution concurrente d'après l'équation 7.4. Donc, dans le cas où un module et un de ses modules précédents se retrouvent en concurrence sur un même noeud, nous avons une interdépendance entre les équations 7.4 et 8.1. Dans ce cas nous ne pouvons déterminer les T^{it}, les $T^{I/O}$ et donc les T^{conc} de ces modules. Le temps d'itération d'un module précédent dépend également du temps d'itération de ses propres modules précédents qui sont eux mêmes exprimés à partir de l'équation 7.4. Une interdépendance peut donc se produire de manière plus générale lorsque des modules appartenant à un même chemin synchrone, constitué d'arcs, sont en concurrence.

Modélisation et détection des interdépendances

Afin de modéliser les interdépendances nous ajoutons dans le graphe G_{sync} des arêtes entre les modules placés sur les mêmes noeuds et n'appartenant pas au même cycle synchrone. En effet, nous avons vu que dans le cas d'un cycle synchrone les modules appartenant à ce cycle ne s'exécutent jamais simultanément car ils s'attendent mutuellement. Le nouveau graphe ainsi obtenu est appelé G_{dep}. Un exemple de graphe G_{dep} pour une application constituée de deux modules placés sur un même noeud est présenté figure 8.1.

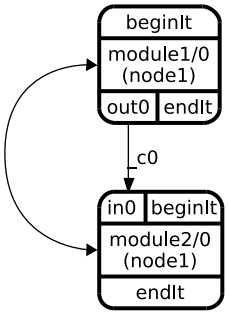

FIG. 8.1 – Exemple d'interdépendance entre deux modules placés sur le même noeud

Les interdépendances se produisent lorsque le temps d'itération d'un module est exprimé en fonction du temps d'exécution concurrente d'un autre module et réciproquement, c'est à dire lorsque des modules sont à la fois synchronisés et en concurrence. Une interdépendance est donc représentée par un cycle dans G_{dep} contenant à la fois des arêtes (synchronisations) et des arcs (concurrence).

En effet un arc entre deux modules signifie que le temps d'itération de l'un va s'exprimer en fonction du temps d'exécution concurrente de l'autre. Par exemple si nous considérons m_1 et m_2 appartenant à la même composante synchrone de G_{sync} tels qu'il existe un arc entre m_1 et m_2 alors $T^{it}_{m_2}$ va dépendre de $T^{it}_{m_1}$ d'après l'équation 7.3, $T^{it}_{m_2} = max(T^{conc}_{m_2}, T^{it}_{m_1})$. Si m_1 et m_2 sont placés sur le même noeud alors nous avons une arête entre eux. Le calcul de $T^{conc}_{m_2}$ va dépendre de l'ordre des valeurs de $T^{I/O}_{m_1}$ et de $T^{I/O}_{m_2}$ d'après notre algorithme. Or $T^{I/O}_{m_2}$ est exprimée en fonction de $T^{it}_{m_2}$ d'après l'équation 8.1.

Résolution des interdépendances

Nous pouvons distinguer différents cas d'interdépendances suivant la composition du cycle, et nous proposons différentes heuristiques pour essayer d'y remédier :

- Interdépendances entre des modules de la même composante synchrone : Dans ce cas on fait l'hypothèse que le temps d'itération est le même pour tous les modules. Cela semble souhaitable car si le temps d'itération n'est pas le même alors nous pouvons prédire un dépassement de tampon. Avec cette hypothèse, calculer le temps d'attente des modules revient à comparer leurs temps de calcul respectifs car nous avons $T^{I/O}_m + T^{exec}_m \times LD_m = T^{it}_m$. On peut donc les ordonner dans l'ordre inverse de leur temps de calcul ($= T^{exec}_m \times LD_m$) puis déterminer leurs temps d'exécution concurrente à partir de notre algorithme.
- Interdépendances entre modules non prédécesseurs de différentes composantes synchrones : Dans ce cas nous pouvons également utiliser l'hypothèse du temps d'itération identique pour les modules de la même composante. Les temps d'itération des différentes composantes sont alors fixés par leurs modules prédécesseurs respectifs.
- Autres cas : Dans les autres cas nous avons des cycles comportant au moins un module prédécesseur et des modules d'autres composantes. Une solution consiste à fixer les temps d'itération des modules prédécesseurs dans le cycle à la valeur de leur temps d'exécution : $T^{it}_{pm} = T^{exec}_{pm}$. On reprend ensuite notre hypothèse ce qui nous permet d'ordonner les modules. On applique alors l'algorithme 1 en observant l'évolution de l'ordre de priorités des modules. Si l'ordre est identique alors on obtient la même allocation d'après notre

algorithme, on a donc un ordonnancement stable. Dans le cas contraire, l'ordre obtenu est différent alors les performances vont varier dynamiquement à l'exécution de l'application. On peut alors recommander au développeur de déplacer certains modules pour supprimer l'interdépendance ou de fixer les priorités au niveau de l'ordonnanceur.

Notons que dans tous les cas les résultats obtenus à l'aide de ces approches ne sont pas garantis. De manière à obtenir des performances stables nous recommandons donc d'éviter les situations d'interdépendances en modifiant le placement des modules.

8.1.4 Remarques

Dans cette approche, les modules sont placés par l'utilisateur sur les noeuds de la grappe. Le placement des modules sur les processeurs est ensuite laissé à l'ordonnanceur du système d'exploitation. Cette solution est simple et fonctionne bien dans la plupart des cas car le nombre de modules placés sur une machine SMP reste proche du nombre de processeurs. En effet on souhaite généralement éviter la concurrence qui peut détériorer les performances des modules.

Néanmoins cette approche présente de nombreux inconvénients. Tout d'abord cet ordonnancement est dynamique et peut entraîner des variations dans les performances de l'application. Une variation locale de la performance des modules peut se propager aux modules synchronisés présents sur d'autres machines et affecter ainsi le comportement global de l'application. De plus l'ordonnancement ne tient pas compte de la performance globale de l'application mais seulement des modules placés sur une même machine. Les performances offertes ne sont donc généralement pas optimales.

Finalement cette approche suppose un système d'exploitation commun sur les noeuds de la grappe. Dans le cas contraire, il faut modéliser toutes les politiques des systèmes présents ce qui s'avère long et fastidieux. De plus, il est probable que les politiques futures de l'ordonnanceur du noyau Linux ne soient plus basées sur des files d'attente, comme par exemple l'ordonnanceur CFS intégré au noyau Linux 2.6.23. Afin de s'abstraire des détails de l'ordonnancement du système et de ne plus faire reposer la performance sur lui, une solution consiste à déterminer un ordonnancement tenant compte du comportement globale de l'application et de la performance attendue pour l'application de réalité virtuelle.

8.2 Allocation dirigée par le modèle de performance

Afin de résoudre les problèmes de l'approche précédente, nous proposons maintenant de définir une allocation statique des modules sur les processeurs dirigée par l'étude des synchronisations entre les modules de l'application. Cette approche permet de supprimer les problèmes de synchronisation (dépassements de tampon), les interdépendances, le comportement dynamique inhérent à l'ordonnanceur du système d'exploitation mais également d'abstraire notre modèle du système d'exploitation.

Le principe est de choisir le temps d'itération de chacune des composantes, et donc celui des prédécesseurs. Ce choix permet de déterminer la charge qui doit être attribuer aux prédécesseurs et fixe une valeur limite aux temps d'exécution concurrente des autres modules. A partir de cette valeur limite on détermine ensuite la charge minimale à allouer à chaque module afin de ne pas dépasser le temps d'itération de la composante correspondante. Afin d'améliorer

les performances, le développeur peut choisir d'allouer une charge plus importante comprise entre cette charge minimale et la charge LD du module. Une fois les charges choisies pour les différents modules, il reste à trouver une allocation des modules concurrents sur les différents processeurs du noeud.

8.2.1 Performance des composantes synchrones

Nous nous plaçons dans le cas ou nous souhaitons éviter les erreurs d'exécution dues à des problèmes de synchronisation. Nous devons donc assurer que le temps d'itération soit le même pour tous les modules d'une même composante. Pour une composante donnée, ce temps d'itération dépend de celui des modules prédécesseurs et ne peut être inférieur à leurs temps d'exécution. Le développeur a par contre la possibilité de choisir un temps d'itération supérieur afin de ralentir les modules de la composante. Cette opération peut être réalisée en modifiant la charge allouée aux prédécesseurs, ou plus simplement en ajoutant un synchroniseur de type $MaxFrequency$.

Considérons une composante synchrone C_{sync} de G_{sync} et choisissons un temps d'itération $T^{it}_{C_{sync}}$. Nous déterminons ensuite la charge LD^c_{pm} qui doit être allouée à chacun des prédécesseurs pm de C_{sync} pour atteindre ce temps d'itération. Cette charge dépend de la charge du module ainsi que du rapport entre le temps de calcul du module ($T^{exec}_{pm} \times LD_{pm}$) et le temps de calcul concurrent souhaité pour celui-ci ($T^{it}_{C_{sync}} - T^{I/O}_{pm}$). Nous obtenons donc la valeur de LD^c_{pm} à partir de l'équation suivante :

$$LD^c_{pm} = LD_{pm} \times \frac{T^{exec}_{pm} \times LD_{pm}}{T^{it}_{C_{sync}} - T^{I/O}_{pm}} \tag{8.4}$$

8.2.2 Calcul des charges minimales des modules

Nous déterminons maintenant les charges minimales à allouer aux autres modules de la composante C_{sync}. Lorsque cette charge minimale est allouée, le temps d'exécution concurrente du module est égal au temps d'itération de la composante et le module n'attend plus dans la méthode *wait()*. Nous pouvons donc également appliquer l'équation précédente pour déterminer leur charge minimale $LDcminm$:

$$LD^{cmin}_m = LD_m \times \frac{T^{exec}_m \times LD_m}{T^{it}_{C_{sync}} - T^{I/O}_m} \tag{8.5}$$

Notons que le développeur a ensuite la possibilité d'allouer une charge LD^c_m telle que $LD^{cmin}_m \leq LD^c_m \leq LD_m$ de manière à atteindre le temps d'exécution concurrente souhaité.

8.2.3 Modules non concurrents

Pour chaque module m nous disposons de son temps d'exécution concurrente T^{conc}_m ainsi que de la charge LD^c_m à allouer sur un processeur du noeud cible. Avant de placer les modules sur les processeurs, nous étudions les synchronisations entre modules afin de déterminer ceux qui ne sont jamais exécutés simultanément et d'optimiser ainsi l'allocation.

Nous avons déjà vu que les modules appartenant à un même cycle synchrone ne peuvent être exécutés simultanément. Si plusieurs modules du même cycle sont placés sur un même noeud alors nous pouvons les placer sur un même processeur sans qu'ils soient en concurrence entre eux et ainsi optimiser l'allocation. En effet ces modules occupent alors une charge qui équivaut au plus à celle du module possédant la charge la plus élevée.

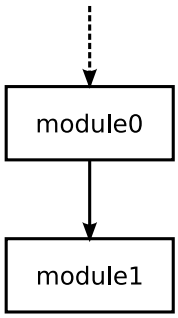

FIG. 8.2 – Chemin synchrone

Considérons maintenant deux modules organisés comme décrit figure 8.2 et étudions leur exécution détaillée figure 8.3. Soit $T^{it}_{C_{sync}}$ le temps d'itération de leur composante. A $t_0 = 0$, $module_0$ reçoit un message et commence son itération. A la fin de son itération, $t_1 = T^{conc}_{module_0}$, il émet un message à destination de $module_1$ par l'intermédiaire de la connexion c_0, puis il fait appel à la méthode *wait()* et se met en attente d'un nouveau message. Le message émis par $module_0$ nécessite un temps t_{c_0} pour être transmis :

$$t_{c_0} = \frac{Vol_{c_0}}{BW_{net_{c_0}}} + L_{net_{c_0}} \tag{8.6}$$

Il est reçu par $module_1$ au temps $t_2 = t_1 + t_{c_0}$. Le $module_1$ commence alors son exécution qui se terminera à l'instant $t_3 = t_2 + T^{conc}_{module_1}$. Si $t_3 \leq T^{it}_{C_{sync}}$ alors $module_0$ n'a pas encore commencé sa prochaine itération. Dans ce cas, $module_0$ et $module_1$ n'ont pas été exécutés simultanément. L'exécution se poursuit de la même manière à l'itération suivante. Les modules ne seront donc jamais exécutés simultanément. Nous en concluons que des modules appartenant à un même chemin synchrone $P_{sync}(V_P, E_P)$ ne seront jamais exécutés simultanément s'ils vérifient l'inégalité suivante :

$$\sum_{m \in V_P} T^{conc}_m + \sum_{c \in E_P} \left(\frac{Vol_c}{BW_{net_c}} + L_{net_c} \right) \leq T^{it}_{C_{sync}} \tag{8.7}$$

L'identification des chemins synchrones vérifiant cette inégalité se fait par un parcours en profondeur de chaque composante synchrone. Le placement des modules sur les noeuds étant donné, seuls les chemins synchrones entre modules de la même composante synchrone et placés sur les mêmes noeuds sont à considérer. Comme dans le cas des modules appartenant à des cycles synchrones, les modules d'un même chemin synchrone ne s'exécutant pas simultanément peuvent être placés sur le même processeur afin d'optimiser l'allocation. Notons qu'un même module peut appartenir à plusieurs chemins synchrones vérifiant l'inégalité 8.7. Il appartient alors au développeur de choisir le chemin qu'il souhaite optimiser.

8.2.4 Placement des modules

Nous considérons maintenant le placement des modules placés sur un même noeud sur les différents processeurs de ce noeud. Nous pouvons placer plusieurs modules sur le même

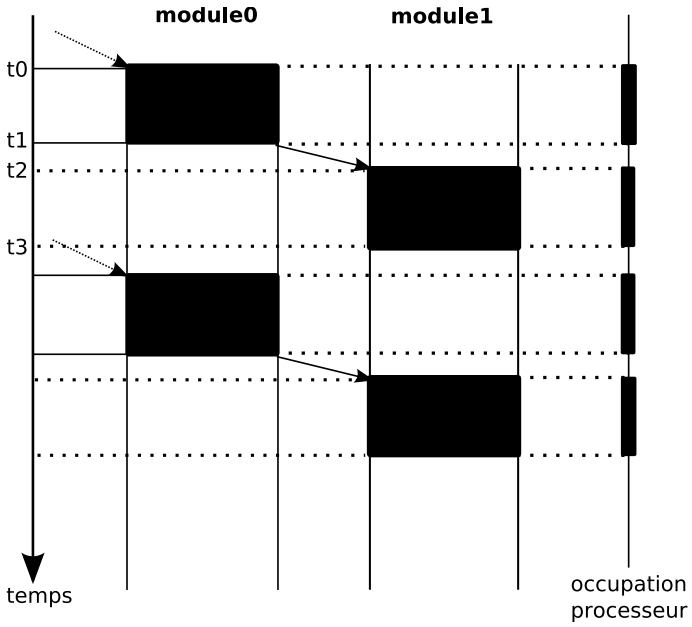

FIG. 8.3 – Exécution de 2 modules appartenant au même chemin synchrone

processeur tant que la somme de leurs charges LD^c ne dépasse pas 1. Pour chaque processeur d'un même noeud nous devons donc trouver une allocation qui vérifie l'inégalité suivante :

$$\sum_{\forall m \in Modules, cpu_m = cpu_n^i} LD_m^c \leq 1 \tag{8.8}$$

Il ne s'agit pas de trouver l'allocation qui minimise les charges des processeurs du noeud mais de décider si une allocation qui respecte la contrainte définit par l'inégalité 8.8 existe. Ce problème peut être ramené à un problème d'optimisation combinatoire de type Sac-à-dos (SAD) qui est reconnu comme NP-complet dans sa version décisionnelle. Dans le cadre de notre étude nous considérons des applications distribuées à gros grain. Le nombre de modules placés sur un même noeud est donc comparable au nombre de processeurs disponibles. Le nombre de placements à considérer et donc limité sur les architectures SMP ne disposant que de quatre processeurs. Cependant les architectures actuelles évoluent vers des noeuds SMP et des processeurs multi-cores. Nous pouvons alors envisager l'utilisation de méthodes métaheuristiques comme par exemple les algorithmes génétiques afin de résoudre le problème d'allocation.

Si une telle allocation n'est pas possible sur un noeud alors nous pouvons choisir un temps d'itération supérieur pour les composantes des modules placés sur ce noeud afin de diminuer leurs charges. Comme nous disposons des informations de charges sur les différents processeurs des noeuds, une autre solution consiste à modifier le placement des modules afin de mieux répartir les charges entre les noeuds.

8.3 Conclusion

Nous avons présenté deux solutions pour modéliser les effets de la concurrence sur les modules d'une application à partir d'un placement donné. La première consiste à modéliser

la politique d'ordonnancement des différents systèmes d'exploitation des noeuds de la grappe. Cependant l'ordonnancement du système d'exploitation est effectué localement sur chaque noeud et ne tient pas compte des synchronisations globales de l'application. Les performances obtenues ne sont donc pas garanties comme optimales. De plus le comportement dynamique de l'ordonnanceur peut entraîner des performances variables. Cette solution semble donc plus adaptée dans le cas de grappes disposant d'un système d'exploitation homogène et lorsque peu de modules sont en concurrence.

Une autre approche consiste à se baser sur une étude des synchronisations entre les modules afin de déterminer les charges minimales à attribuer à chaque module. Le développeur a ensuite la possibilité de choisir une charge supérieure afin d'améliorer les temps d'exécution concurrente et de diminuer ainsi la latence. Si une allocation des charges sur les processeurs est possible, alors nous pouvons garantir les performances des différents modules. Cette approche permet de s'abstraire du système d'exploitation, elle requiert cependant un réglage fin de l'ordonnanceur afin d'assurer l'allocation des charges des modules. Cette opération est laissée à la charge du développeur. On peut également envisager de développer un ordonnanceur spécifique afin d'automatiser ce réglage.

Ces deux approches nous permettent de calculer les temps d'itération et d'exécution concurrente des différents modules de l'application. Nous disposons donc de la fréquence de chaque module. Nous sommes maintenant en mesure d'étudier les performances des communications.

Chapitre 9

Modèle pour les communications

Sommaire

Nous étudions maintenant les communications entre les différents composants de l'application. Le graphe de l'application modélise les communications FlowVR entre composants. Nous devons également considérer les communications qui peuvent intervenir en dehors du schéma FlowVR, par exemple lorsqu'un module parallèle utilise une librairie MPI.

L'objectif est dans un premier temps d'obtenir les volumes de données échangés sur chaque lien réseau afin de vérifier qu'ils ne dépassent pas les capacités physiques du réseau et détecter ainsi les dépassements des tampons d'envoi. Puis nous nous intéressons au calcul de la latence entre composants de l'application afin de vérifier que l'application répond bien au critère d'interactivité.

9.1 Modélisation des communications

9.1.1 Communications FlowVR

Les communications FlowVR sont représentées par des arêtes dans le graphe de l'application. Chaque arête est associée à un volume de données envoyé par itération. Afin de simplifier notre étude nous avons choisi de ne pas considérer les communications avec les synchroniseurs. En effet ces communications ne contiennent que des estampilles qui sont de petits messages utilisés par les synchroniseurs pour contrôler le couplage de l'application. Par exemple si on connecte un module à un synchroniseur $MaxFrequency$ afin de limiter sa fréquence, le synchroniseur échange uniquement un signal avec le module. De plus les synchroniseurs sont

généralement placés sur le noeud où se trouvent les modules qu'ils contrôlent. Dans ce cas ils n'engendrent pas de communications sur les réseaux.

9.1.2 Communications en dehors du modèle FlowVR

Les modules FlowVR peuvent faire appel à des communications en dehors du schéma FlowVR. Ceci est nécessaire lorsque des instances d'un module parallèle doivent communiquer au sein de la même itération. Par exemple, dans un module parallèle de simulation d'écoulement de fluide, le domaine est réparti entre les différentes instances du module. Au cours de chaque itération, les instances doivent s'échanger des informations afin de garder une cohérence globale de la simulation. Or le modèle FlowVR n'autorise pas les instances d'un même module à communiquer pendant la même itération. Il est alors nécessaire de recourir à d'autres solutions comme par exemple une librairie de type MPI. Ces communications ne sont pas visibles dans le graphe FlowVR. Afin de les modéliser nous avons besoin que le développeur du module fournisse le schéma de communication ainsi que le volume de données échangé entre les instances. Nous ajoutons ensuite des arêtes dans le graphe de l'application correspondant à ces communications entre les différentes instances du module parallèle. Chacune de ces arêtes est associée à un volume de données transmis par itération.

Maintenant que nous avons modélisé l'ensemble des communications entre composants nous allons déterminer les volumes de données échangés sur chaque lien réseau.

9.2 Performance des communications

9.2.1 Evaluation des volumes de données échangés

Notre objectif est de calculer les volumes de données émis et reçus par chaque noeud à travers les liens réseaux afin de déterminer si ces volumes n'excèdent pas les possibilités physiques des réseaux.

Dans un premier temps nous calculons le volume de données émis par seconde par chaque connexion. Pour chaque connexion, ce volume dépend du volume émis par itération et de la fréquence du composant émettant à travers cette connexion. Nous avons évalué les fréquences des modules, il reste à déterminer celles des filtres. La fréquence d'un filtre dépend de celle des modules en amont du flot de données sauf dans le cas du filtre *greedy* qui ne fournit un nouveau message qu'à la fin de l'itération du module récepteur auquel il est connecté. Un parcours du graphe à partir des modules permet donc de déterminer la fréquence des filtres. Pour chaque connexion c nous définissons src_c et $dest_c$ comme respectivement le composant à l'origine de la connexion c et le composant à destination de la connexion c. Nous définissons Vol_c^s comme le volume de données émis par seconde à travers une connexion c par :

$$Vol_c^s = Vol_c^{it} \times F_{src_c} \tag{9.1}$$

Nous additionnons ensuite les volumes des connexions empruntant le même lien réseau afin de calculer la bande passante engendrée. Nous obtenons ainsi pour chaque lien $netlink \in Netlinks$ la bande passante utilisée, notée $bw_s(netlink)$, d'après l'équation suivante :

$$bw_s(netlink) = \sum_{\substack{\forall c \in E, \\ Net(c)=net, \\ node(src(c))=n}} Vol_c^s \tag{9.2}$$

Si cette valeur dépasse la bande passante du réseau associé alors nous pouvons prédire que les messages vont s'accumuler dans le tampon d'envoi et produire un dépassement de celui-ci. Nous devons donc vérifier que $bw_s(netlink) \leq BW(net_{netlink})$ pour éviter ce cas.

Nous procédons de même pour déterminer la bande passante $bw_r(netlink)$ requise pour la réception des messages à travers chaque lien réseau $netlink \in Netlinks$. Nous utilisons pour cela l'équation suivante :

$$bw_r(netlink) = \sum_{\substack{\forall c \in E, \\ Net(c)=net, \\ node(dest(c))=n}} Vol_c^s \tag{9.3}$$

De la même manière que précédemment nous vérifions que la bande passante requise pour réceptionner les messages est bien inférieure à la bande passante du réseau c'est à dire que $bw_r(netlink) \leq BW(net_{netlink})$.

9.2.2 Mesure de la latence

De manière générale, la latence représente le temps entre l'émission et la réception d'une information. Dans cette section nous nous intéressons au temps nécessaire à une information pour être transmise d'un composant à un autre. Elle dépend donc du temps de traitement de cette information par les composants ainsi que du temps nécessaire pour transmettre leurs messages. Nous devons donc tenir compte des caractéristiques des réseaux utilisés.

Dans les applications interactives, la latence ne doit pas être perceptible par l'utilisateur afin d'assurer une bonne interactivité. Cependant la présence de filtres $greedy$ peut entraîner la perte ou la réplication d'une information. Dans notre définition de la latence nous considérons que l'information n'est pas perdue par le filtre $greedy$. Nous exprimons la latence entre deux composants suivant un chemin P les reliant dans le graphe de l'application. Dans le meilleur cas, la latence est minimale lorsque les messages sont émis directement par l'interface réseau à la fin de l'itération :

$$L_{min}(P) = \sum_{m \in P} T_m^{conc} + \sum_{\substack{\forall c \in P \\ Net_c \neq \emptyset}} \left(\frac{Vol_c^{it}}{BW_{net_c}} + L_{net_c}\right) \tag{9.4}$$

Lorsque plusieurs composants d'un même noeud émettent sur le même lien, les messages sont placés dans une file d'attente. L'étude des bandes passantes requises sur chaque lien réseau nous permet d'assurer que les liens sont capables d'émettre les volumes de données requis. Dans le pire cas le message sera placé dans la file d'attente mais sera émis avant que le message produit par l'itération suivante ne soit produit et mis en attente. Nous pouvons donc borner la latence :

$$L_{max}(P) = \sum_{m \in P} T_m^{conc} + T_m^{it} + \sum_{\substack{\forall c \in P \\ net_c \neq \emptyset}} \left(\frac{Vol_c^{it}}{BW_{net_c}} + L_{net_c}\right) \tag{9.5}$$

9.2.3 Echantillonnage des filtres $greedy$

Afin de prendre en compte la perte ou la réplication d'une information par un filtre f_g de type $greedy$, nous définissons son taux d'échantillonnage E_{f_g} comme le rapport des fréquences des modules connectés à ce filtre :

$$E_{f_g} = \frac{F_{src_{f_g}}}{F_{dest_{f_g}}} \tag{9.6}$$

Lorsque $E_g > 1$, des messages sont perdus par le filtre $greedy$. Si $E_g < 1$ alors le module destination peut récupérer plusieurs fois le même message. La connaissance du taux d'échantillonnage des filtres $greedy$ permet au développeur de régler les performances de son application. Par exemple si le taux d'échantillon est trop important cela signifie que beaucoup de calculs sont inutiles car leurs résultats sont perdus. Ce comportement n'est pas toujours souhaitable et dépend de la sémantique que le développeur souhaite donner à son application.

9.3 Conclusion

A partir des informations de fréquence, de placement, ainsi que de la modélisation du réseau, nous sommes capables d'évaluer les volumes de données qui transitent à travers les différents liens des réseaux. Ces informations nous permettent d'invalider des placements qui offrent le niveau de performance souhaité pour les composants mais qui génèrent trop de communications ou des latences trop importantes.

Ces informations peuvent être exploitées par le développeur afin d'optimiser ses schémas de communication, par exemple en utilisant de multiples réseaux, ou en évaluant les gains de performance qu'un réseau plus performant pourrait offrir.

Troisième partie

Programmation par contraintes pour le placement d'applications distribuées

Chapitre 10

Objectifs et présentation

Sommaire

Le modèle que nous proposons facilite l'évaluation des performances d'un placement donné. Néanmoins, la recherche d'un placement offrant les performances espérées par l'utilisateur est toujours à la charge du développeur. Cette recherche est basée sur son expérience et n'offre aucune garantie en terme d'optimalité. De plus le nombre de placements potentiels croît en fonction du nombre de noeuds et de composants de l'application et il devient rapidement impossible de les considérer tous. Afin de décharger le développeur de cette tâche, il serait donc souhaitable de créer un outil capable d'automatiser la recherche de placements. Cet outil permettrait ainsi à l'utilisateur final d'étudier directement l'influence d'un changement de couplage, ou de l'optimisation d'un module sur les performances et pourrait le guider dans ses développements.

Notre problème de placement s'exprime naturellement sous forme de contraintes : les communications ne doivent pas dépasser les capacités des réseaux, les modules ne doivent pas être plus lents que leurs prédécesseurs, la fréquence du module de visualisation doit être supérieure à 15Hz, etc.

Nous proposons donc d'étudier le principe de la programmation par contraintes ainsi que les possibilités offertes par cette approche afin de résoudre notre problème de placement. Dans un deuxième temps, nous étudions la complexité du problème de placement d'une application. Nous évaluons à cette fin le nombre de placements potentiels en fonction des différents paramètres de l'application et de l'architecture. Nous discutons finalement des possibilités offertes par la programmation par contraintes pour résoudre et optimiser notre problème de placement.

10.1 La programmation par contraintes

La programmation par contraintes [16] est un paradigme de la programmation informatique utilisé pour résoudre notamment des problèmes d'allocation, de planification ou d'ordonnancement. La programmation par contraintes est utilisée dans de nombreux domaines, citons par exemple le domaine des transports aériens lorsqu'il s'agit d'allouer les emplacements d'un aéroport à des avions [18], ou celui de la gestion de ressources humaines pour attribuer des personnels à des tâches [53]. L'objectif de cette approche est de répondre aux questions suivantes :

- Existe t-il une solution à mon problème ?
- Quelles sont les solutions de ce problème ?
- Quelle est la solution optimale à mon problème pour un critère donné ?

Nous présentons dans cette section le formalisme utilisé, ainsi que les méthodes de résolution et d'optimisation proposées par la programmation par contrainte.

10.1.1 Formalisme

Un problème de satisfaction de contraintes, ou CSP (*Constaint Satisfaction Problem*) est un problème de recherche qui consiste en un ensemble de variables définies sur des domaines finis et en un ensemble de relations logiques, les contraintes, définies entre ces variables. Formellement, un CSP est défini comme un 3-uple (V, D, C) tel que :

- V est l'ensemble des variables de notre problème ;
- D est l'ensemble des domaines finis pour les variables de V ;
- C est l'ensemble des contraintes définies sur les variables de V.

Le domaine d'une variable x est noté $D(x)$.

Les contraintes portent généralement sur un nombre limité de variables qui correspond à leur arité, par exemple la contrainte $x+y = z$ est d'arité 3. Il existe également des contraintes d'arité arbitraire appelées contraintes globales [19]. Un exemple de contrainte globale fréquemment utilisé est la contrainte *allDifferent(E)*, qui impose aux variables de l'ensemble E de prendre des valeurs différentes.

Une solution d'un CSP est une affectation des variables de V dans D vérifiant chacune des contraintes de C. Considérons par exemple le problème comportant deux variables x et y, dont les domaines respectifs sont $D(x) = \{0, 1, 2\}$ et $D(y) = \{1, 2, 3\}$, et un ensemble de contraintes $C = \{x \neq y, x + y = 3\}$. Alors l'affectation $A = \{(x, 1), (y, 2)\}$ est une solution de ce problème car elle vérifie les contraintes du problème.

10.1.2 Résolution d'un problème de contraintes

La recherche de solutions d'un CSP consiste à tester les affectations possibles et à vérifier les contraintes. Elle peut être réalisée par différents types d'algorithmes. Un algorithme naïf consiste à générer toutes les solutions et à tester les contraintes sur chacune. Cependant la taille de l'espace de recherche est égale au produit cartésien des tailles des domaines des variables. Cette solution n'est donc pas satisfaisante.

Une autre manière couramment employée et plus efficace consiste à utiliser un algorithme dit de *back-tracking* avec un algorithme de propagation de contraintes. Le *back-tracking* consiste

en une recherche arborescente sur l'ensemble des domaines des variables. Dès qu'une affectation partielle est inconsistante, il est inutile de continuer à explorer le domaine des autres variables, on remonte donc à la dernière affectation consistante obtenue et on recommence la recherche avec une nouvelle valeur pour la dernière variable affectée.

La propagation consiste à réduire les domaines des variables non encore affectées en supprimant des valeurs localement inconsistantes pour une contrainte. Ces suppressions s'enchaînent, d'où le nom de propagation.

Afin d'optimiser la recherche il est possible de définir des heuristiques, c'est à dire des règles de parcours de l'arbre de recherche non systématiques. Une heuristique sur l'ordre d'instanciation des variables permet généralement d'accélérer la recherche de solution, mais cet ordre dépend du problème considéré. Par exemple, une possibilité est de commencer par instancier les variables les plus contraignantes, c'est à dire apparaissant dans le plus grand nombre de contraintes.

10.1.3 Optimisation d'un problème de contraintes

L'optimisation dans un CSP a pour objectif de répondre à la question : quelle est la solution optimale à mon problème pour un critère donné ?

La technique principale d'optimisation dans un CSP est le *branch-and-bound*. Supposons que l'on veuille maximiser une variable X. La technique consiste, après avoir trouvé une solution dans laquelle $X = V$, à poursuivre la recherche en ajoutant la contrainte $X > V$. Les solutions de moins bonne qualité sont coupées dans la suite du problème et on retourne comme réponse la dernière solution trouvée.

10.2 Combinatoire des placements

Notre modèle de performance permet d'évaluer les performances de placements sans avoir à procéder à de longues et laborieuses étapes de déploiements d'applications distribuées sur une grappe. Cependant, le nombre de placements potentiels croit rapidement lorsqu'on augmente la taille de l'application ou le nombre de noeuds de l'architecture distribuée.

10.2.1 Evaluation du nombre de placements potentiels

L'évolution du nombre de placements dépend également de l'hétérogénéité de l'architecture considérée. Si on considère le cas simple d'une application comportant un seul composant alors nous avons la possibilité de le placer sur un des noeuds de l'architecture.

Si l'architecture est homogène, ces différents placements ne sont distingués que par le numéro du noeud sur lequel le composant se trouve. Nous ne pouvons donc les distinguer en terme de performance, nous considérons donc que nous avons un seul placement possible.

Si l'architecture est hétérogène le nombre de placements dépend du nombre de types différents de noeuds. Dans le cas de notre application à un composant, si nous avons deux types différents de noeuds alors nous avons deux possibilités de placements que l'on peut distinguer par leur performance.

Nous étudions maintenant plus généralement le nombre de placements en distinguant les cas des architectures homogènes et hétérogènes.

noeud1	noeud2	noeud3	noeud4
1,2	3	4	5,6

noeud1	noeud2	noeud3	noeud4
3	5,6	1,2	4

TAB. 10.1 – Deux placements équivalents sur une architecture homogène

Architecture homogène

Considérons une application composée d'un ensemble de n composants , et une grappe homogène constituée d'un ensemble de noeuds l tous connectés aux mêmes réseaux. Dans ce cas nous ne pouvons pas distinguer en terme de performance deux placements à des permutations de noeuds près. Par exemple, si nous avons une application de six composants et une grappe dotée de quatre noeuds, les placements décrits 10.1 offrent les mêmes performances.

Le nombre de placements possibles correspond alors à la manière de les regrouper, c'est à dire au nombre de partitions de n éléments en au plus l sous-ensembles. Le nombre de partitions de n éléments en k sous-ensembles est donné par le nombre de Stirling du second type $S(n,k)$:

$$S(n,k) = \sum_{j=1}^{k} (-1)^{k-j} \frac{j^{n-1}}{(j-1)!(k-j)!}$$

La table 10.2 donne les premières valeurs du nombre de Stirling.

$n\backslash k$	0	1	2	3	4	5	6	7	8	9
0	1									
1	0	1								
2	0	1	1							
3	0	1	3	1						
4	0	1	7	6	1					
5	0	1	15	25	10	1				
6	0	1	31	90	65	15	1			
7	0	1	63	301	350	140	21	1		
8	0	1	127	966	1701	1050	266	28	1	
9	0	1	255	3025	7770	6951	2646	462	36	1

TAB. 10.2 – Nombres de Stirling du second type

Pour obtenir le nombre de placements possibles de n composants sur l processeurs homogènes, que nous notons $N(n,l)$, nous devons donc faire la somme des nombres de Stirling pour des valeurs de k variant de 0 à l :

$$N(n,l) = \sum_{k=0}^{l} S(n,k)$$

Nous remarquons que pour $k = 2$, ce qui est le minimum pour constituer une grappe, nous avons $S(n,2) = 2^{n-1} - 1$. Dans le cadre de notre étude ($l \geq 2$) nous obtenons ainsi la minoration du nombre de placements suivante :

$$2^{n-1} - 1 \leq N(n,l),\ l \geq 2$$

Le nombre de placements sur une architecture distribuée donnée augmente donc exponentiellement en fonction du nombre de composants.

Architecture hétérogène

Dans le cas d'une grappe complètement hétérogène, chaque composant peut être placé sur un des l noeuds. Nous avons donc dans le pire cas l^n placements possibles car nous n'avons plus de placements symétriques.

Si la grappe est constituée de deux sous-ensembles de noeuds homogènes de tailles respectives l_0 et $l_1 = l - l_0$, alors pour chaque composant nous avons déjà le choix de les placer sur un des ces deux sous-ensemble, soit au minimum 2^n possibilités puisque ces deux sous-ensembles ne sont pas vides et que nous avons au minimum un noeud dans chaque. Nous pouvons donc préciser l'intervalle sans lequel se situe le nombre de placements potentiels, noté $N^{\neq}$:

$$2^n \leq N^{\neq} \leq l^n$$

Notons que, pour un même nombre de modules, le fait de passer d'une architecture homogène à une architecture composée de deux types de noeuds accroît le nombre de placements potentiels d'un facteur au moins égal à 2.

Plus précisément, si nous répartissons les n composants en deux sous-ensembles de n_0 et $n_1 = n - n_0$ composants et que nous les plaçons respectivement sur les deux ensembles de noeuds de tailles l_0 et l_1 alors le nombre de placements sur le sous-ensemble de taille l_0 est égal à $N(n_0, l_0)$, et à $N(n_1, l_1)$ sur celui de taille l_1. Le nombre de placements pour une partition de n composants sur ces deux sous-ensembles est donc égal à $N(n_0, l_0) \times N(n_1, l_1)$.

Nous avons ensuite à choisir n_0 composants parmi n, soit $C_n^{n_0}$ choix possibles, les composants restant sont placés dans n_1. On obtient ainsi $C_n^{n_0} \times N(n_0, l_0) \times N(n_1, l_1)$ possibilités de répartir les composants de n en deux sous-ensemble de tailles données. Nous devons ensuite considérer toutes les tailles possibles pour ces sous-ensembles et additionner les possibilités de placements associées pour chacune. On obtient finalement le nombre total de possibilités de placement des composants de n sur une architecture constituées de deux sous-ensembles homogènes de noeuds :

$$N^{\neq} = \sum_{i=0}^{n} C_n^i \times N(i, l_0) \times N(n-i, l_1).$$

Cette valeur constitue une borne inférieure plus précise.

En fonction de l'hétérogénéité de la grappe considérée, le nombre de placements que l'on peut obtenir est donc compris entre $2^{n-1} - 1$, dans le cas homogène, et l^n lorsque l'architecture est complètement hétérogène. Dans tous les cas ce nombre évolue de manière exponentielle. Considérons par exemple le placement de 8 composants sur 4 noeuds homogènes. Nous faisons la somme des valeurs de Stirling correspondantes dans le tableau 10.2 et nous obtenons $N(8,4) = 1 + 127 + 966 + 1701 = 2795$. Il n'est donc pas raisonnable de tester ces différents placements sur l'architecture cible. Notons que, si nous contraignons le placement d'un seul module à un noeud donné, le nombre de placement est de $N(7,4) = 1 + 63 + 301 + 350 = 715$ soit un gain d'un facteur 4.

10.2.2 NP-Complétude

Le problème d'optimisation du placement d'une application distribuée sur une grappe de PC est analogue au problème combinatoire appelé Bin Packing qui consiste à placer différents objets dans une quantité finie de conteneurs de capacité donnée. Dans notre cas un objet

représente un composant doté d'une charge et le conteneur un processeur disposant d'une charge finie. Le problème de Bin Packing est reconnu comme NP-difficile, il n'existe donc pas d'algorithme déterministe permettant de le résoudre en temps polynomial (tant qu'on a pas démontré que $P = NP \ldots$).

De nombreuses méthodes sont utilisées pour simplifier ce genre de problème et trouver de bonnes solutions sans garantie évidemment d'obtenir leur optimalité : les algorithmes génétiques, le recuit simulé, etc.

10.3 Conclusion

La programmation par contrainte est utilisée pour résoudre de nombreux problèmes d'allocations et d'ordonnancements comparables à notre problème de placement. Des contraintes peuvent être modifiées ou ajoutées à notre problème, ce qui apporte à cette approche plus de souplesse que l'utilisation d'algorithmes génétiques. Autre avantage, un solveur de contraintes peut être utilisé à double sens : soit pour rechercher une solution satisfaisant les contraintes, soit pour vérifier une solution proposée par le développeur en réduisant les domaines des variables à une valeur donnée.

L'utilisation de contraintes rend la recherche de solutions plus efficaces. Cela permet de repousser le problème de l'explosion combinatoire de l'espace de recherche mais ne le résout pas. Nous devons donc envisager l'utilisation d'heuristiques appropriées afin de pouvoir résoudre des problèmes de placements d'applications comportant plus de composants sur des architectures disposant de plus de noeuds.

Chapitre 11

Modélisation du problème de placement

Sommaire

Dans ce chapitre nous proposons de modéliser notre problème de placement sous forme d'un CSP. D'une part nous disposons d'une architecture donnée, d'autre part nous avons une application fournie par le développeur. Notre objectif est de trouver une affectation de chaque composant de l'application sur un processeur de l'architecture. Pour cela, nous utilisons dans cette approche le principe d'allocation décrit à la section 8.2 et non pas la modélisation de l'ordonnanceur du noyau Linux. Le développeur doit donc choisir les performances de son application en fixant les temps d'itération des composantes synchrones de son application, pour fixer leurs fréquences, ainsi que les temps d'exécution concurrente pour chacun des modules, de manière à respecter une certaine latence c'est à dire le temps que met une information pour être traitée par les composants et transmise entre eux. Pour chaque module il est ainsi possible de choisir un temps d'exécution concurrente compris entre le temps d'itération de la composante

et le temps d'exécution minimum du module sur les différents noeuds de l'architecture :

$$min_{\forall n_j \in Nodes}(T^{exec}_{m_i,n_j}) \leq T^{conc}_{m_i} \leq T^{it}_{C_{sync}},\ \forall m_i \in C_{sync} \tag{11.1}$$

Dans un premier temps nous présentons les données fournies en entrée à notre modèle. Nous définissons ensuite les variables de notre problème ainsi que leurs domaines associés. Puis nous exprimons les différentes contraintes définies entre ces variables. Nous présentons également quelques heuristiques destinées à limiter la recherche de solutions. Finalement nous nous attachons à l'étude de l'optimisation d'un placement suivant un critère donné.

Afin d'illustrer notre approche, nous déroulons dans ce chapitre un exemple de modélisation du problème de placement de l'application de la figure 6.2 sur l'architecture représentée figure 6.1. Pour l'expression des variables, domaines et contraintes, nous reprenons les notations précédemment utilisées pour la modélisation décrites dans la partie 6.

11.1 Données du problème

Notre modèle nécessite en entrée une architecture cible et une application donnée par le développeur.

11.1.1 Architecture

L'architecture est composée des ensembles suivants :

- $Nodes = \{node_0, \ldots, node_{|Nodes|-1}\}$ représente l'ensemble des noeuds de l'architecture ;
- $CPUs = \{cpu_0, \ldots, cpu_{|CPUs|-1}\}$ désigne l'ensemble des processeurs de l'architecture ;
- $Networks = \{net_0, \ldots, net_{|Networks|-1}\}$ est l'ensemble des réseaux ;
- $Netlinks = \{netlink_0, \ldots, netlink_{|Netlinks|-1}\}$ représente l'ensemble des liens appartenant aux réseaux.

Dans notre exemple, ces ensembles sont décrits à la section 6.1.

Chaque processeur doit ensuite être associé à un noeud, et chaque lien réseau à un réseau.

Processeurs et noeuds

La modélisation de l'architecture est statique, nous utilisons donc des constantes pour la définir. Pour associer chaque processeur $cpu_i \in CPUs$ à un noeud $n_j \in Nodes$ nous définissons la constante $node_{cpu_i}$ telle que :

$$node_{cpu_i} = n_j \tag{11.2}$$

Réseaux

Chaque réseau $net_i \in Networks$ est associé à une bande passante BW_{net} et une latence L_{net}.

De la même manière chaque lien réseau $netlink_i \in Netlinks$ est associé à un unique réseau $net_j \in Networks$. Nous définissons ainsi la constante $net_{netlink_i}$ par :

$$net_{netlink_i} = net_j \tag{11.3}$$

Les paramètres réseaux de notre exemple sont décrits par les tableaux 6.2 et 6.3 de la section 6.1.

11.1.2 Application

L'application est décrite comme dans notre modèle par les ensembles suivants :

- $Modules$, l'ensemble des modules de l'application ;
- $Filters$ représente l'ensemble des filtres de l'application ;
- $Synchronizers$, l'ensemble des synchroniseurs de l'application ;
- E représente l'ensemble des connexions.

Dans le cadre de notre exemple le lecteur trouvera ces ensembles décrits à la section 6.2.

Pour chaque module $m_i \in Modules$ nous disposons de son temps d'exécution $T^{exec}_{m_i,n_j}$ et sa charge LD_{m_i,n_j} sur chaque noeud $n_j \in Nodes$.

Pour chaque connexion c_i nous déterminons le volume de données Vol_{c_i} en suivant la démarche décrite à la section 6.2.3.

11.1.3 Performance des modules

Notre approche consiste à placer les modules de notre application sur les processeurs en fonction des performances que souhaite obtenir le développeur. Nous utilisons dans le processus d'allocation décrit à la section 8.2. Le développeur doit donc choisir les temps d'itération des composantes synchrones ainsi que les temps d'exécution concurrente des modules. Ces données sont représentées pour chaque module m_i par les constantes $T^{it}_{m_i}$ et $T^{conc}_{m_i}$.

Dans notre exemple, l'application comporte une seule composante connexe car la suppression du schéma de synchronisation entre $module_1$, $module_{3/0}$ et $module_{3/1}$ ne déconnecte pas le graphe, le temps d'itération est donc imposé pour tous les modules :

$$T^{it}_{module_1} = T^{it}_{module_{2/0}} = T^{it}_{module_{2/1}} = T^{it}_{module_{3/0}} = T^{it}_{module_{3/1}} \tag{11.4}$$

Pour chaque module m_i nous pouvons donc définir statiquement les charges $LD^c_{m_i,n_j}$ qu'il va potentiellement occuper sur les différents noeuds $n_j \in Nodes$ en fonction de leurs types.

Un module ne sera donc placé sur un noeud que s'il peut respecter le temps d'itération fixé. Nous pouvons donc également pré-calculer les volumes de données qui transitent sur chaque connexion FlowVR par seconde en utilisant l'équation 9.1. Nous définissons ainsi une constante $Vol^s_{c_i}$ pour chaque connexion $c_i \in E$.

11.2 Variables et domaines

Pour représenter les variables de notre problème nous reprenons une partie des notations utilisées dans notre modèle.

11.2.1 Variables de l'architecture

Charge des processeurs

Le placement d'un module sur un processeur entraîne l'augmentation de la charge de ce dernier. Cette charge est modélisée pour chaque processeur $cpu_i \in CPUs$ par une variable notée LD_{cpu_i} qui représente un pourcentage compris entre 0, si aucun module n'est placé sur lui, et 100%. Nous choisissons donc de définir la variable LD_{cpu_i} sur le domaine des entiers suivant :

$$D(LD_{cpu_i}) = [0..100] \tag{11.5}$$

Charge des réseaux

Nous considérons des liens réseaux fonctionnant en *full-duplex* c'est à dire que la bande passante est identique en émission et réception et qu'il est possible de recevoir et d'émettre des données simultanément. Dans chaque sens de communication, la charge d'un lien réseau ne peut pas dépasser la valeur de sa bande passante maximale. Les charges des liens réseaux sont donc définies sur les domaines suivants :

$$D(bw^s_{netlink_i}) = [0..BW_{net_{netlink_i}}] \tag{11.6}$$
$$D(bw^r_{netlink_i}) = [0..BW_{net_{netlink_i}}] \tag{11.7}$$

Notons que ces valeurs sont exprimées en octets pour obtenir un domaine fini.

11.2.2 Variables du placement

Nous disposons d'un ensemble de variables pour décrire le placement des composants et des connexions de notre application sur l'architecture cible.

Placement des composants

Un module $m_i \in Modules$ peut être placé sur un des noeuds de l'architecture, le domaine de la variable $node_{m_i}$ est donc à priori l'ensemble des noeuds $Nodes$:

$$D(node_{m_i}) = Nodes \tag{11.8}$$

Chaque module est également placé sur un des processeurs de l'architecture, nous définissons donc pour chaque module m_i la variable cpu_{m_i} dont le domaine est l'ensemble des processeurs $CPUs$:

$$D(cpu_{m_i}) = CPUs \tag{11.9}$$

Notons que le processeur sur lequel est placé un module doit appartenir au noeud sur lequel est placé le module. Nous verrons dans la section suivante comment assurer cette cohérence à l'aide d'une contrainte.

Nous devons placer de la même manière chaque filtre sur un noeud. Nous définissons donc le placement d'un filtre f_i par la variable $node_{f_i}$ dont le domaine est l'ensemble des noeuds :

$$D(node_{f_i}) = Nodes \tag{11.10}$$

Notons que, dans notre approche, nous avons choisi de ne pas modéliser la charge des filtres. Nous n'avons donc pas besoin de définir leur placement sur les différents processeurs.

Placement des connexions

Chaque connexion $c_i \in E$ est placée sur deux liens réseaux représentés par les variables $netlink^0_{c_i}$ et $netlink^1_{c_i}$ qui prennent une valeur dans l'ensemble $Netlinks$ et nous définissons donc leurs domaines respectifs de la manière suivante :

$$D(netlink^0_{c_i}) = Netlinks \tag{11.11}$$
$$D(netlink^1_{c_i}) = Netlinks \tag{11.12}$$

Une connexion est également placée sur un des réseaux de l'architecture. Nous définissons donc la variable net_{c_i} sur le domaine $Networks$:

$$D(net_{c_i}) = Networks \tag{11.13}$$

Evidemment, nous devons vérifier que les liens réseaux utilisés pour la connexion appartiennent au même réseau et que les noeuds où sont placés les composants source et destination ont bien accès à ce réseau. Cette tâche est effectuée par des contraintes qui sont définies dans la section suivante.

11.3 Contraintes du modèle

Le problème de placement peut se décomposer en quatre étapes :

1. placement des composants sur les processeurs de notre architecture ;
2. calcul et vérification des charges sur les processeurs ;
3. placement des connexions sur les liens d'un réseau disponible entre les modules source et destination ;
4. calcul et vérification des volumes de données échangés qui ne doivent pas dépasser les capacités des réseaux.

Le placement des modules conditionne leur performance en fonction du type du noeud hôte ainsi que de la concurrence avec les autres modules placés sur le même noeud. Nous utilisons dans cette étude le modèle d'allocation dirigée par notre modèle de performance afin de s'abstraire du système d'exploitation considéré. Nous considérons donc qu'un module est placé sur un processeur et non pas sur un noeud.

11.3.1 Placement des composants

Dans notre modèle de performance nous considérons que les synchroniseurs engendrent des charges et des communications négligeables par rapport aux modules et filtres. Le placement des synchroniseurs n'est donc pas contraint et ne génère pas de contraintes sur les autres composants. Nous choisissons donc de les ignorer dans cette étude.

Un module m_i est placé sur processeur cpu_{m_i} et un noeud $node_{m_i}$. Pour que le placement de m_i soit cohérent, le processeur cpu_{m_i} doit appartenir au noeud $node_{m_i}$. Nous posons donc l'ensemble de contraintes suivant :

$$node_{m_i} = node_{cpu_{m_i}}, \text{pour chaque } m_i \in Modules \tag{11.14}$$

11.3.2 Calcul des charges sur les processeurs

Chaque module ajoute sa charge sur le processeur où il est placé. Une première restriction est que la somme des charges des modules placés sur un même noeud ne peut dépasser 100%. Afin de calculer la charge totale des processeurs nous définissons l'ensemble de contraintes suivant :

$$LD_{cpu_i} = \sum_{cpu_{m_j}=cpu_i} LD^c_{m_j}, \text{pour chaque } cpu_i \in CPUs \tag{11.15}$$

Notons que nous n'avons pas besoin de poser la contrainte $LD_{cpu_i} \leq 100$ car la valeur maximale du domaine $D(LD_{cpu_i})$ est égale à 100, et le calcul d'une valeur supérieure provoque automatiquement la sortie de ce domaine.

11.3.3 Placement des connexions

Les contraintes suivantes doivent être posées afin d'assurer le placement des connexions.

Cohérence des liens

Lorsqu'un composant est placé sur un processeur, et donc sur un noeud, les connexions qui lui sont associées en émission et réception doivent être placées sur un des liens réseau du noeud. Nous devons donc vérifier pour chaque connexion que le lien réseau utilisé en émission (respectivement réception) appartient bien au noeud où est placé le module source (respectivement destination) de cette connexion. Nous pouvons donc définir les deux ensemble de contraintes suivantes :

$$node_{netlink^0_{c_i}} = node_{src_{c_i}}, \text{ pour chaque } c_i \in E \tag{11.16}$$

$$node_{netlink^1_{c_i}} = node_{dest_{c_i}} \tag{11.17}$$

Appartenance au même réseau

Si deux modules communiquent, alors la connexion qui les relie doit être placée sur des liens appartenant à un même réseau et nous avons donc l'ensemble de contraintes suivant :

$$network_{netlink^0_c} = network_{netlink^1_c}, \text{ pour chaque } c \in E \tag{11.18}$$

Communication locale

Dans le cas particulier de deux composants placés sur un même noeud les communications se font par la mémoire partagée. Le réseau est donc dans ce cas imposé. Nous devons donc poser l'ensemble de contraintes suivant :

$$node_{src_{c_i}} = node_{dest_{c_i}} \Rightarrow net_{c_i} = local, \text{ pour chaque } c_i \in E \tag{11.19}$$

11.3.4 Volumes de données échangés

Afin d'assurer que le débit sur un lien réseau ne dépasse pas le volume autorisé par le réseau nous devons additionner les débits des connexions empruntant les mêmes liens réseau

et vérifier que cette valeur est bien inférieure. Le calcul du volume de données engendré par seconde sur chaque lien réseau est effectué en transformant l'équation 9.2 en contrainte :

$$bw^s_{netlink_i} = \sum_{netlink_{c_j}=netlink^0_i} Vol^s_{c_j} \quad (11.20)$$

$$bw^r_{netlink_i} = \sum_{netlink_{c_j}=netlink^1_i} Vol^s_{c_j} \quad (11.21)$$

Comme dans le cas du calcul des charges processeurs, la définition des domaines de ces variables rend redondantes les contraintes $bw^s_{netlink_i} \leq BW_{netlink_i}$ et $bw^r_{netlink_i} \leq BW_{netlink_i}$ dont on peut ainsi se passer.

11.4 Autres contraintes

Afin d'améliorer la recherche de solutions, il est possible ajouter des contraintes supplémentaires issues des particularités de l'application et de l'architecture.

11.4.1 Contraintes de l'architecture

Le placement de certains modules est conditionné par la présence d'un périphérique connecté à un noeud donné. Ainsi le module d'interaction est nécessairement placé sur le noeud connecté au périphérique qu'il doit piloter. De même le placement de modules de visualisation se fait sur les noeuds connectés aux dispositifs d'affichage correspondants.

Différents systèmes d'exploitation peuvent également être présents sur les différents noeuds de l'architecture, certains codes ne peuvent donc être déployés que sur certains d'entre eux.

La programmation par contrainte simplifie l'ajout de contraintes et leurs modifications. Le développeur peut ainsi placer un module m_i sur un noeud n_j en ajoutant la contrainte $node_{m_i} = m_j$. Il est également possible de restreindre le placement d'un module m_i à un sous-domaine de l'architecture en modifiant son domaine.

11.4.2 Elimination des symétries

Certains placements sont identiques à une symétrie près. Dans notre exemple, figure 6.2, si les modules $module_{2/0}$ et $module_{2/1}$ ont les mêmes caractéristiques alors leur placement peut être interverti sans affecter les performances.

Afin de casser les symétries nous pouvons donc ordonner le placement des modules en utilisant une contrainte lexicographique. Dans notre exemple nous posons ainsi la contrainte suivante :

$$cpu_{module_{2/0}} \leq cpu_{module_{2/1}} \quad (11.22)$$

Sans cette contrainte le nombre de placements possibles pour ces deux modules est égal à $|CPUs| \times |CPUs|$. Avec cette contrainte le second module ne peut être placé que sur les noeuds d'ordre lexicographique supérieur. Si le premier module est placé sur le processeur cpu_i alors le second sera placé sur un des $|CPUs| - i$ processeurs restants. En effectuant la somme des combinaisons possibles on obtient un nombre de placements égal à $\frac{|CPUs| \times (|CPUs|-1)}{2}$. Cette contrainte permet donc de réduire par deux l'espace de recherche pour le placement de ces modules.

11.4.3 Placement des filtres

Considérons dans notre exemple les modules $module_1$, $module_{2/0}$ et $module_{2/1}$ placés sur trois noeuds distincts. Si nous plaçons le filtre $broadcast_0$ sur un quatrième noeud alors toutes les connexions reliant ces modules et ce filtre vont emprunter une connexion réseau. Par contre, si le filtre est placé sur le même noeud que l'un de ces modules nous économisons une connexion réseau et nous optimisons ainsi l'utilisation du réseau ainsi que la latence.

A partir de cette remarque nous pouvons restreindre le placement d'un filtre aux noeuds sur lesquels sont placés les modules qui y sont connectés en utilisant une contrainte disjonctive. Dans notre exemple nous posons ainsi la contrainte suivante :

$$node_{broadcast_0} = node_{module_1} \vee node_{broadcast_0} = node_{module_{2/0}} \vee node_{broadcast_0} = node_{module_{2/1}} \quad (11.23)$$

Le domaine d'un filtre se voit donc limité à deux valeurs. La taille du domaine de recherche pour un filtre est donc réduite d'un facteur $\frac{|Nodes|}{2}$. Notons que cette optimisation n'est pas obligatoire et que le développeur peut choisir de ne l'appliquer qu'à certains filtres.

11.5 Conclusion

Nous avons exprimé notre problème de placement sous forme d'un problème de contraintes. Cette approche est indépendante de la structure de l'application et de l'architecture ou la topologie de la grappe considérée. Notre modèle est capable de considérer des architectures hétérogènes, des réseaux multiples et également hétérogènes.

La programmation par contrainte permet d'envisager de multiples applications pour le développeur : la génération de placements est l'application immédiate de notre approche ; la validation de placements consiste à fixer le placement et appliquer les contrainte pour le vérifier ; la prévision de performance sur une architecture virtuelle, ce qui permet de déterminer les investissements matériels à effectuer pour être capable de faire fonctionner une application avec des critères de performance donnés ; l'optimisation car un problème de contrainte peut facilement être transformé en problème d'optimisation ce qui permet d'obtenir le placement optimal pour un critère donné.

Afin de tester ces différentes possibilités nous avons effectué une implantation de notre problème à l'aide d'un solveur de contraintes. Nous proposons dans le chapitre 12.4 une présentation de ce solveur ainsi que de son utilisation sur une application de réalité virtuelle pour résoudre notre problème de placement.

Quatrième partie

Expérimentations

Chapitre 12

Expérimentations

Sommaire

Afin de valider notre modèle et les placements produits par notre solveur nous avons développé différentes applications distribuées de test, ainsi qu'une application interactive de simulation d'écoulements de fluide. Nous avons ensuite testé ces applications sur la plateforme de réalité virtuelle du Laboratoire d'Informatique Fondamentale d'Orléans.

Nous décrivons dans ce chapitre les caractéristiques et les codes de ces différentes applications ainsi que la plateforme de réalité virtuelle utilisée.

12.1 Applications

12.1.1 Module générique

Pour pouvoir évaluer notre modèle sur des structures d'applications variées nous avons dans un premier temps développé un module FlowVR générique. Ce module est paramètrable afin de pouvoir modifier son comportement, c'est à dire son temps de calcul et sa charge. Il est ainsi possible de simuler différents types d'applications par simple instanciation et configuration de ce module sans avoir à en modifier son code.

Paramètres du module

Le module générique est un programme qui peut être configuré en utilisant différents arguments lors de son appel :

- *sleep* spécifie une durée incompressible d'attente du module ;
- *calc* spécifie le temps de calcul du module ;
- *in* définit le nombre de ports d'entrées ;
- *out* définit le nombre de ports de sorties ;
- *outports* contient la liste des tailles des messages envoyés sur les ports à chaque itération.

Ces paramètres sont fixés pour la durée de son exécution. Le module est associé au fichier de description présenté figure A.1. Ce fichier définit la commande à lancer pour exécuter le module (entre les balises <run></run>) ainsi que le script qui va permettre de générer la liste des ports d'entrées et de sorties en fonction des paramètres passés (entre les balises <script></script>).

Définition et configuration des modules de l'application

Les modules d'une application FlowVR sont définies dans un fichier de description de l'application au format XML. Un exemple de fichier de configuration d'une application comportant deux modules est décrit figure A.2.

Chaque module est associé à une section dans ce fichier décrivant son nom dans l'application, le fichier de description du module, dans notre exemple celui du module générique, ses paramètres entre les balises *template*, ainsi que son placement sur l'architecture cible et la commande pour l'exécuter. L'ajout de nouveaux modules dans l'application se fait simplement en ajoutant d'autres sections similaires.

Nous devons ensuite définir les connections entre les différents modules.

Communication et couplage de l'application

Le schéma de communication et de synchronisation est décrit dans un script basé sur le langage Perl. Dans le script donnée figure A.3 une connexion synchrone est ajoutée entre les deux modules de l'application décrit figure A.2.

Le changement de configuration des schémas de communication et de synchronisation ne nécessite donc pas de modifier le code du programme mais seulement de redéfinir ce script. Dans notre exemple, on peut ainsi remplacer la connexion synchrone par un filtre *greedy* en substituant la commande `&addSimpleConnection` par la commande `&addGreedy`.

Graphe de l'application

Le graphe de l'application obtenu à partir du fichier de description décrit figure A.2 et du fichier de couplage présenté figure A.3 est généré automatiquement par l'outil `flowvr-graph`. Le résultat obtenu est présenté figure 12.1.

12.1.2 Application de Réalité Virtuelle : FluidParticle

L'application FluidParticle est utilisée pour visualiser le déplacement de particules suivant un écoulement de fluide. L'utilisateur a d'une part la possibilité de déplacer le point de vue afin de se déplacer autour ou dans le champs de particules, et d'autre part il peut perturber l'écoulement de manière interactive.

L'application est constituée des modules suivants :

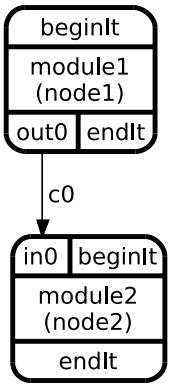

FIG. 12.1 – Résultat de la génération de graphe à partir de fichiers de configuration FlowVR

- *Interaction* est un module qui permet de capturer les mouvements de l'utilisateur ;
- *Simulation* effectue la simulation d'écoulement, il s'agit d'une version parallèle basée sur MPI d'une simulation proposée par Stam [47] qui est également utilisée dans l'exemple *fluid* fourni avec la bibliothèque FlowVR [1] ;
- *Particles* effectue l'advection d'un ensemble de particules qui peut être réparti sur différentes instances de ce module ;
- *Viewer* transforme les particules en primitives graphiques, plusieurs instances de ce module permettent de répartir cette opération ;
- *Renderer* est un module de l'extension FlowVRRender qui effectue le rendu des primitives graphiques, ce module peut être instancié plusieurs fois afin de distribuer le rendu sur plusieurs périphériques d'affichage.

Nous décrivons maintenant l'organisation de ces modules et la manière dont l'information est transmise entre eux.

Tout d'abord le module *Interaction* récupère un déplacement fournit par un dispositif de pointage et l'envoi vers *Simulation* de manière à perturber l'écoulement de fluide. Ces modules fonctionnent à des vitesses très différentes, un dispositif d'interaction fonctionne généralement à une fréquence comprise entre 200 et 1000Hz alors que notre simulation fonctionne à environ 25Hz. Afin d'éviter de saturer la simulation nous ajoutons définissons donc une connexion *greedy* entre ces modules.

Le module *Simulation* émet ensuite un champs de forces à destination du module *Particles* afin d'effectuer l'advection des particules. La connexion est de type FIFO car sémantiquement une itération de la simulation correspond à un déplacement des particules. La position du dispositif de pointage est également envoyée vers le module *Viewer* afin d'être transformée en primitive graphique et d'être ainsi représentée. Nous avons choisi de ne pas communiquer directement la position du pointeur issue du module *Interaction* mais de la récupérer en sortie de la simulation afin de pouvoir "poser" le pointeur sur le fluide est observer son déplacement suivant l'écoulement de fluide. Cela permet également d'assurer la cohérence entre la position du pointeur dans la simulation et à l'affichage.

Une fois les particules advectées par le module *Particles*, leurs positions sont envoyées à destination du module *Viewer*.

Finalement les primitives graphiques issues du *Viewer* sont transmises au module *Renderer* pour les afficher. Le module *Renderer* gère le point de vue utilisé pour effectuer le rendu de la scène. Une connexion *greedy* est donc définie entre le module *Viewer* et *Renderer* afin de pouvoir modifier ce point de vue sans devoir se synchroniser avec le module *Viewer*. Nous

pouvons ainsi avoir une fréquence d'affichage indépendante de la fréquence de la simulation.

Le graphe de l'application obtenue avec une instance de chaque module est présenté figure 12.2.

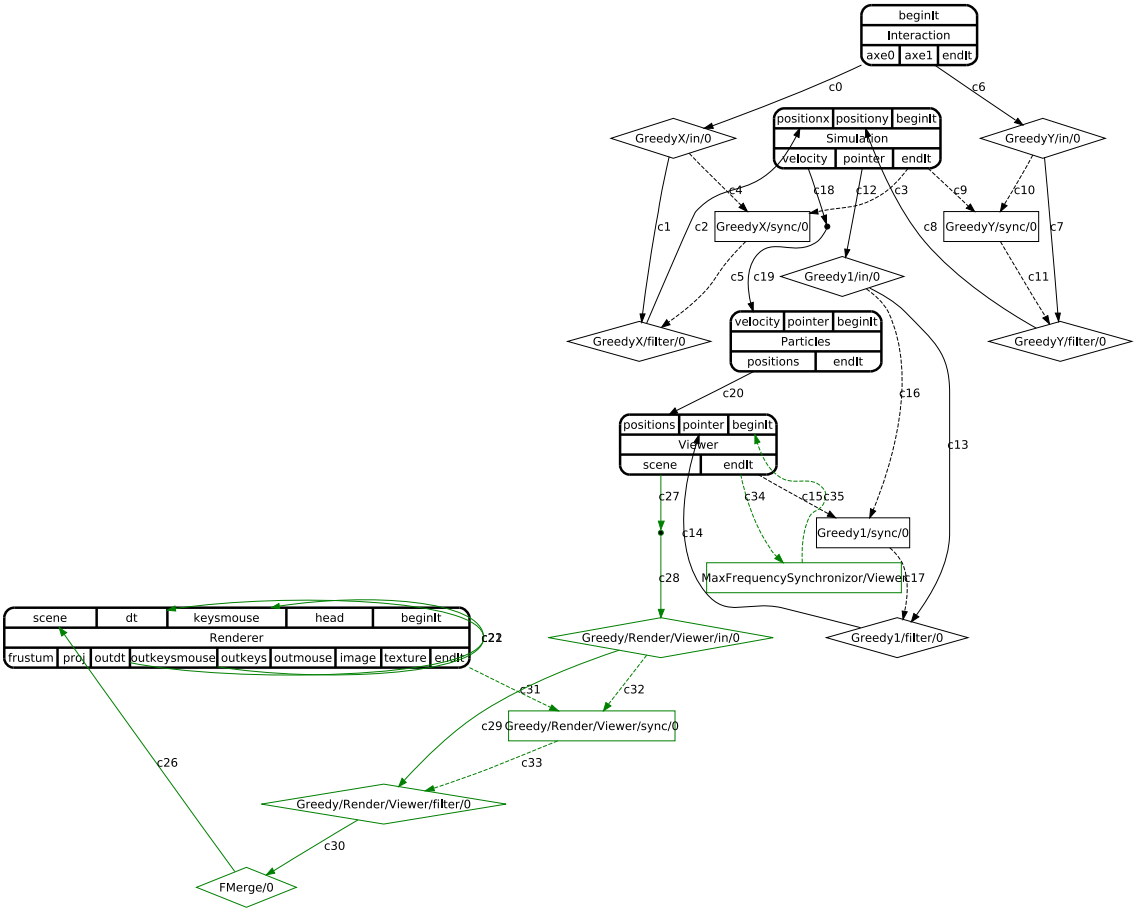

FIG. 12.2 – Organisation de l'application FluidParticle

12.2 La plateforme MiReV

Nos expérimentations sont effectuées sur MiReV (figure 12.3), la plateforme de réalité virtuelle du LIFO. Cette plateforme est constituée d'une grappe de PC à laquelle est connecté un mur d'image. Le système d'exploitation utilisé est basé sur un noyau Linux 2.6.18.

La grappe est constituée de 16 noeuds répartis en deux sous-ensembles :

- 8 noeuds SMP comportant 2 processeurs Opteron doubles-coeurs (64 bits) et 4Go de mémoire vive ;
- 8 noeuds SMP disposant de 2 processeurs Xeon (32 bits) et de 2Go de mémoire vive.

Nous nommons $node_1, node_2, \ldots, node_8$ les noeuds du premier sous-ensemble et $node_{11}, \ldots, node_{18}$ ceux du second sous-ensemble.

Ces noeuds sont interconnectés par un réseau gigaEthernet commun. Chaque sous-ensemble possède également un réseau dédié : un second réseau gigaEthernet pour le premier et un réseau

FIG. 12.3 – MiReV : la plateforme de réalité virtuelle du LIFO

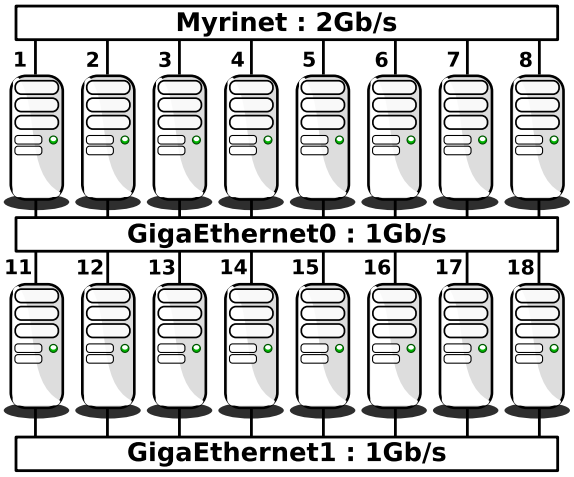

FIG. 12.4 – Schéma d'interconnexion de la grappe de PC de la plateforme MiReV

Myrinet pour le second. Le schéma d'interconnexion de la grappe est présenté figure 12.4

Le mur d'image est constitué d'un écran sur lequel sont projetés les images issues de quatre vidéoprojecteurs connectés à quatre noeuds de la grappe ce qui permet de distribuer l'affichage et d'obtenir une résolution supérieure.

12.3 Validation du modèle

Dans cette section nous réalisons différentes applications basées sur notre module générique décrit précédemment. En fonction de leur stucture et de leur placement, ces applications nous permettent de mesurer les effets des synchronisations, de la concurrence et des communications sur les performances. Dans chaque cas, nous comparons ces résultats aux valeurs déterminées à l'aide de notre modèle pour vérifier son adéquation et valider nos différentes hypothèses sur le modèle FlowVR.

Nous effectuons ensuite plusieurs déploiements de notre application FluidParticle et vérifions de la même manière les performances obtenues avec celles déterminées à partir de notre modèle.

12.3.1 Protocole de test

Pour garantir les résultats obtenus lors de nos tests, la plateforme MiReV est isolée et dédiée à l'exécution de nos applications. Nous évitons ainsi que les exécutions soient perturbés par d'autres processus.

Afin de déterminer le plus précisément possible les valeurs mesurées, nous réalisons lors de chaque test plusieurs exécutions de nos applications. Les différents temps retenus sont obtenus en effectuant la moyenne des valeurs mesurées de ces temps sur plusieurs milliers d'itérations. Les mesures sont effectuées à l'aide de la commande `gettimeofday` qui fournit une précision de l'ordre de la μs. Nos tests montrent que la mesure du temps d'exécution d'un code varie de l'ordre de 1 à $2ms$ entre chaque exécution. Nos résultats sont donc exprimés en utilisant cette unité avec une précision de l'ordre de $2ms$.

12.3.2 Synchronisation

Nous testons dans un premier temps les différentes configurations de synchronisation entre les modules. Afin d'étudier uniquement l'effet des synchronisations sur les performances et d'éviter les phénomènes de concurrence nous plaçons chaque module sur un processeur différent. Les temps d'itération dépendent donc uniquement des temps d'exécution des modules et des temps d'itération de leurs prédécesseurs.

Connexions Greedy

Notre première application est constituée de deux modules $module_1$ et $module_2$ placés sur des noeuds différents, connectés par un schéma de synchronisation *greedy* et dont nous fixons le temps d'exécution. Le graphe de cette application est décrit figure 12.5. Les résultats sont

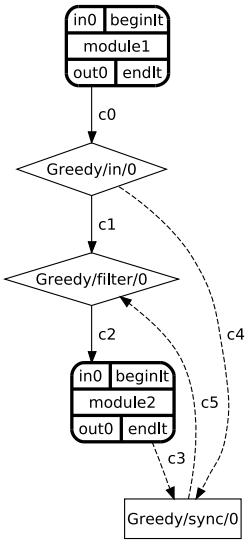

FIG. 12.5 – Schéma *greedy* entre 2 modules

présentés dans le tableau 12.1 et confirme que le schéma *greedy* n'affecte pas les performances des modules qui y sont connectés.

Module	*Placement*	T_{exec}	T_{it} modèle	T_{it} réel
$module_1$	$node_1$	37	37	37
$module_2$	$node_2$	18	18	18

TAB. 12.1 – Performance de 2 modules connectés par un schéma *greedy* (temps en ms)

Connexions FIFO

Nous remplaçons le schéma *greedy* par une connexion FIFO comme décrit figure 12.6. Nous configurons les modules de manière à ce que le temps d'exécution de $module_1$ soit plus lent que celui de $module_2$. Notre modèle prévoit que, dans ce cas, le temps d'itération de $module_2$ doit être égale au temps d'exécution de $module_1$. Les résultats relevés, tableau 12.2, confirment cette prévision.

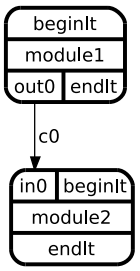

FIG. 12.6 – Modules connectés en FIFO

Nous inversons ensuite la connexion entre les deux modules en gardant les mêmes temps d'exécution respectifs. Dans ce cas nous prévoyons une erreur de dépassement de tampon à l'exécution de l'application, ce qui est confirmé par nos tests. Notons que l'erreur intervient après un certain nombre d'itérations dépendant de la taille de la mémoire partagée allouée.

Si les modules ont des temps d'exécution proches et élevés, s'ils échangent des messages de petite taille et disposent d'une mémoire partagée conséquente, le dépassement de tampon n'intervient pas immédiatement à l'exécution de l'application. Un test basé sur l'exécution d'une application ne suffit donc pas à garantir l'absence de ces erreurs. Notre modèle apporte donc un moyen de détecter efficacement ces erreurs.

Module	*Placement*	T_{exec}	T_{it} modèle	T_{it} réel
$module_1$	$node_1$	37	37	37
$module_2$	$node_2$	18	37	37

TAB. 12.2 – Performance de 2 modules connectés en FIFO (temps en ms)

Cycles synchrones

Nous étudions maintenant les performances de modules organisés en cycles synchrones. Nous considérons l'application décrite par la figure 12.7 qui comporte trois modules. Chaque module émet par itération un message dont la taille est fixée à 5Mo.

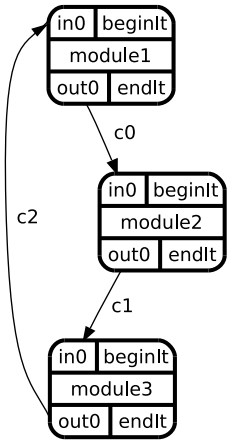

FIG. 12.7 – Modules connectés en cycle

Dans un premier temps nous plaçons les modules de l'application sur un même noeud. La formule 7.5 nous indique que le temps d'itération de ces modules est égale dans ce cas à la somme des temps d'exécution des modules, ce que nous confirme les résultats obtenus, tableau 12.3. Notre hypothèse sur le coût négligeable des transferts par la mémoire partagée entre modules placés sur un même noeud est donc vérifiée.

Module	*Placement*	T_{exec}	T_{it} modèle	T_{it} réel
$module_1$	$node_1$	37	84	84
$module_2$	$node_1$	26	84	84
$module_3$	$node_1$	21	84	84

TAB. 12.3 – Performance d'un cycle de 3 modules placés sur un même noeud (temps en ms)

Nous disposons maintenant les modules du cycle sur des noeuds différents. Nous devons donc tenir compte des temps de communication nécessaires à l'envoi des messages entre les noeuds qui dépendent de la taille des messages, de la bande passante du réseau et de sa latence. Nous utilisons pour ce test le réseau gigaEthernet qui dispose d'une bande passante de 1000Mb/s. Le temps de transfert de chaque message de 5Mo est donc estimé à 50ms. Nous considérons que la latence, de l'ordre de quelques dizaine de ms, est donc négligeable par rapport au temps de communication des messages. D'après la formule 7.5 nous devons additionner les temps de communication nécessaires à l'envoi des trois messages aux temps d'exécution des modules. Les valeurs prévues et mesurées sont présentées dans le tableau 12.4.

Module	*Placement*	T_{exec}	T_{it} modèle	T_{it} réel
$module_1$	$node_1$	37	234	240
$module_2$	$node_2$	26	234	240
$module_3$	$node_3$	21	234	240

TAB. 12.4 – Performance d'un cycle de 3 modules placés sur des noeuds distincts (temps en ms)

On note que, même avec une évaluation simple des capacités du réseau, la valeur prévue par notre modèle est très proche de la valeur mesurée.

12.3.3 Concurrence

Les performances sont également affectées par la concurrence entre les modules. Nous testons dans un premier temps notre modèle basé sur la modélisation de l'ordonnanceur Linux. Nous plaçons 4 modules sur une machine SMP dotée de 2 processeurs. Ces modules sont indépendants et ne sont pas synchronisés. Nous réalisons deux tests en faisant varier les temps de calcul et les charges de modules. Ces données sont décrites dans les tableaux 12.5 et 12.6. Nous commençons par calculer les temps d'attente des modules afin de les ordonner. Puis nous appliquons notre algorithme. Les prévisions et résultats obtenus pour ces deux tests sont présentées dans les tableaux 12.5 et 12.6. Notre algorithme donne une bonne approximation des valeurs effectivement observées dans le premier test.

				Modèle			*Réel*	
Module	*Placement*	T_{exec}	LD	$T_{I/O}$	T_{cexec}	LD_c	T_{cexec}	LD_c
$module_1$	$node_{11}$	50	1.00	0	114	0.44	112	0.45
$module_2$	$node_{11}$	48	0.66	16	80	0.40	68	0.47
$module_3$	$node_{11}$	160	0.50	80	160	0.50	180	0.45
$module_4$	$node_{11}$	90	0.56	50	90	0.56	97	0.52

TAB. 12.5 – Performance de 4 modules concurrents non synchronisés sur un noeud bi-processeurs : test 1

Dans le second test, tableau 12.6 , nous notons que la charge attribuée au module $module_1$ est sous-évaluée par notre approche. Il semble que dans ce cas les modules ne sont pas placés comme décrit dans notre algorithme. Les modules $module_2$ et $module_4$ sont les plus prioritaires d'après leurs temps d'attente respectifs et devraient être placés sur des processeurs différents. Cependant, l'ordonnanceur les place sur un même processeur alors que les modules $module_1$ et $module_3$ sont placés sur l'autre processeur. Cette allocation peut s'expliquer par le fait que les modules $module_2$ et $module_3$ ont des temps d'attente proches et qu'ils sont placés par le scheduler dans la même file d'attente avec la même priorité. Notre approche est donc plus adaptée aux cas où des modules concurrents ont des différences de temps d'attente significatives.

					Modèle		*Réel*	
Module	*Placement*	T_{exec}	LD	$T_{I/O}$	T^{conc}	LD_c	T^{conc}	LD_c
$module_1$	$node_{11}$	20	1.00	0	48	0.42	36	0.55
$module_2$	$node_{11}$	16	0.30	11	16	0.30	19	0.26
$module_3$	$node_{11}$	10	0.50	5	14	0.37	17	0.44
$module_4$	$node_{11}$	51	0.58	20	51	0.58	56	0.55

TAB. 12.6 – Performance de 4 modules concurrents non synchronisés sur un noeud bi-processeurs : test 2

Un exemple de détection d'interdépendances et d'allocation dirigée par les performances est présenté sur notre application FluidParticle à la section 12.3.4

12.3.4 Tests sur l'application FluidParticle

Nous appliquons maintenant notre modèle à notre application FluidParticle décrite à la section 12.1.2. Dans cette étude nous choisissons de ne pas tenir compte du module *Interaction*

car la charge relevée lors de son exécution est inférieure à 1%. De plus sa fréquence dépend de l'interaction de l'utilisateur, dans le cas d'une interaction continue, le périphérique fonctionne à sa fréquence maximale soit 200Hz. A chaque itération deux positions, représentées par deux entiers, sont envoyées sur ces ports de sorties, ce qui représente un volume total de 1,6ko/s, ce qui est négligeable en comparaison des volumes de données échangés par les autres modules.

Les autres modules effectuent uniquement des opérations de calcul, les charges mesurées sont proches de 100%. Afin d'améliorer les performances nous utilisons plusieurs instances pour chaque module :

- le module *Simulation* est instancié 8 fois de manière à atteindre un temps d'exécution de 40ms qui permet d'obtenir une fréquence de 25Hz et donc une simulation interactive. La simulation de fluide est effectuée sur une grille discrétisée de dimension 400×400. Chaque instance dispose donc d'un sous-domaine de 200×100 éléments, chaque élément représentant un vecteur vitesse codé par deux nombres réels (2×32bits).
- les modules *Particles*, *Viewer* sont instanciés 4 fois de manière à optimiser le temps de traitement et donc la latence de l'application. Nous considérons un système de 400×400 particules, soit 200×200 particules par instance du module *Particles*, la position de chaque particule étant représentée par deux nombres flottants (2×32bits).
- le module *Renderer* possède également 4 instances afin de produire un affichage distribué sur le mur d'image de la plateforme MiReV.

Nous obtenons donc un total de 20 modules que nous répartissons dans un premier temps sur 4 noeuds disposant de 4 processeurs chacun ($node_1$ à $node_4$). Etant donné la taille du graphe généré, nous présentons uniquement une version simplifiée du graphe de l'application obtenu (4 instances pour *Simulation*, 4 pour *Particles* et *Viewer*) est présentée à la figure A.4. Notons que notre plateforme nous permettrait de placer chaque module sur des processeurs distincts, mais un de nos objectifs est de réduire le nombre de noeuds occupés par nos applications car la plateforme est partagée par plusieurs utilisateurs.

Etude de la concurrence

Dans un premier temps nous laissons l'ordonnanceur gérer l'allocation des processeurs aux modules. La construction du graphe G_{dep}, représenté figure A.5, de cette application met en évidence des cycles d'interdépendances. Dans ce cas nous pouvons prévoir des performances variables pour les différents modules. Pour remédier à ce problème et obtenir des performances optimales, nous proposons d'isoler les modules de simulation et le module de rendu sur des processeurs dédiés. Les modules *Particles* et *Viewer* sont donc placés sur le dernier processeur disponible. Nous remarquons que ces deux modules font partie d'un chemin synchrone et que la somme de leurs temps d'exécution est inférieure au temps d'itération du module *Simulation*. Ces deux modules ne seront donc pas exécutés simultanément et leur temps d'exécution sera optimal. Nous choisissons donc de leur attribuer une charge maximale correspondant à leur charge LD. Pour chaque module nous décrivons dans le tableau 12.8 son placement sur les différents noeuds, son temps d'exécution T^{exec}, sa charge LD. Nous présentons également l'allocation choisie sur les processeurs de chaque noeud (numérotés de 1 à 4), la charge minimale déterminée à partir des équations 8.5 ains que la charge LD^c choisie pour atteindre le temps d'exécution concurrente T^{conc} attendu.

Nous avons déterminé une allocation qui répond à nos exigences en terme de performance des modules. Nous étudions maintenant les communications entre les différents modules pour vérifier que cette allocation est possible.

Module	Placement		Ordonnanceur	
		T^{exec}	T^{conc}	T^{it}
$Simulation$	$node_{1...4}$	40	45-50	45-50
$Particles$	$node_{1...4}$	9	9	45-50
$Viewer$	$node_{1...4}$	10	10	45-50
$Renderer$	$node_{1...4}$	10	10-20	10-20

TAB. 12.7 – Résultat de l'exécution lors de l'utilisation de l'ordonnanceur du noyau Linux (temps en ms)

Module	Placement			Allocation			Prévision		Mesure	
		T^{exec}	LD	cpu	LD^{cmin}	LD^{c}	T^{conc}	T^{it}	T^{conc}	T^{it}
$Simulation$	$node_{1...4}$	40	1	1, 2	1	1	40	40	42	42
$Particles$	$node_{1...4}$	9	1	4	0.225	1	9	40	10	42
$Viewer$	$node_{1...4}$	10	1	4	0.25	1	10	40	10	42
$Renderer$	$node_{1...4}$	10	1	3	1	1	10	10	11	11

TAB. 12.8 – Résultat de l'exécution à partir d'une allocation fixée (temps en ms)

Etude des communications

Afin d'envoyer l'ensemble du domaine aux instances du module *Particles* nous effectuons un assemblage des sous-domaines à l'aide d'un schéma de communication en arbre effectuant la fusion des sous-domaines à l'aide de filtres *gather*. Chaque instance du module *Particles* transmet ensuite la position de ses particules à l'instance du module *Viewer* correspondante. Enfin un échange global de données est effectué entre les instances du module *Viewer* et celles du module *Renderer*. La figure 12.8 représente le schéma de communication de l'application ainsi que l'estimation des volumes de données échangés par seconde sur chaque lien FlowVR ainsi qu'entre les instances des modules *Simulation* (communications MPI). Notons que pour simplifier la représentation, les instances du module *Simulation* placées sur un même noeud sont représentées comme un seul module.

Les volumes de données reçus et émis par chaque noeud sont obtenus respectivement en additionnant les volumes de chaque lien à destination et en provenance du noeud considéré. Les valeurs obtenues sont présentées dans le tableau 12.9. Nous notons que le volume émis par le noeud $node_1$ est supérieur à la capacité du réseau gigaEthernet. Nous pouvons donc prévoir une saturation du réseau lorsqu'un seul réseau de ce type est utilisé. Une solution pour résoudre ce problème consiste à utiliser le second réseau afin de déplacer une connexion sur celui-ci comme décrit par la figure 12.9.

Noeud	$node_1$	$node_2$	$node_3$	$node_4$
Emission	88	24	56	24
Réception	48	56	64	56

TAB. 12.9 – Volumes de données (Mo/s) émis et reçus par chaque noeud

Nous comparons maintenant nos prédictions aux valeurs mesurées lors de nos expérimentations.

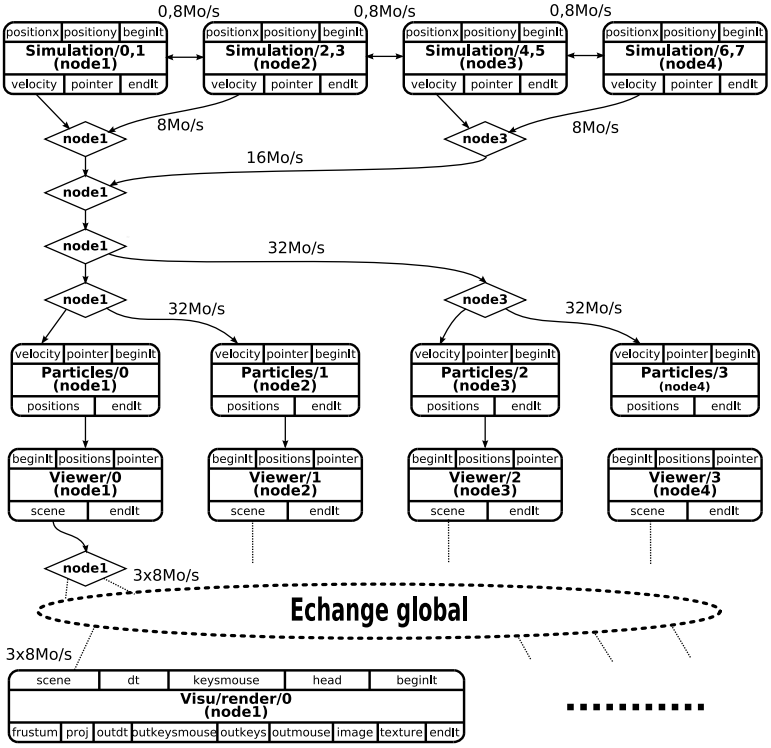

FIG. 12.8 – Schéma de communication et volumes de données échangés

Résultats

Dans un premier temps nous avons laissé l'ordonnanceur du noyau Linux prendre en charge l'exécution de l'application. Les résultats obtenus, tableau 12.7, montrent que les instances des modules de l'application ont des performances variables ce qui coïncide avec la détection de l'interdépendance par notre modèle.

Pour remédier à ce comportement, nous choisissons d'imposer l'allocation décrite par le tableau 12.8. Le placement des modules sur les processeurs est effectué par l'utilisation de la commande `taskset` qui fait partie de l'ensemble d'utilitaires `schedutils` [6]. Les résultats, tableau 12.8, montrent que les temps d'exécution concurrente observés pour les différents modules sont proches des prévisions fournies par notre modèle.

Module	*Placement*			*Allocation*			*Prévision*		*Mesure*	
		T^{exec}	LD	*cpu*	LD^{cmin}	LD^{c}	T^{conc}	T^{it}	T^{conc}	T^{it}
$Simulation$	$node_{1,2}$	40	1	1-4	1	1	40	40	40	40
$Particles$	$node_{11...14}$	13	1	1	0.325	1	13	40	15	40
$Viewer$	$node_{11...14}$	15	1	1	0.375	1	15	40	15	40
$Renderer$	$node_{11...14}$	20	1	2	1	1	20	20	23	23

TAB. 12.10 – Placement de l'application sur des noeuds hétérogènes (temps en ms)

Partant de ce placement nous pouvons également proposer un autre placement de l'application qui utilise 4 noeuds bi-processeurs et 2 noeuds dotés de 4 processeurs. Les noeuds

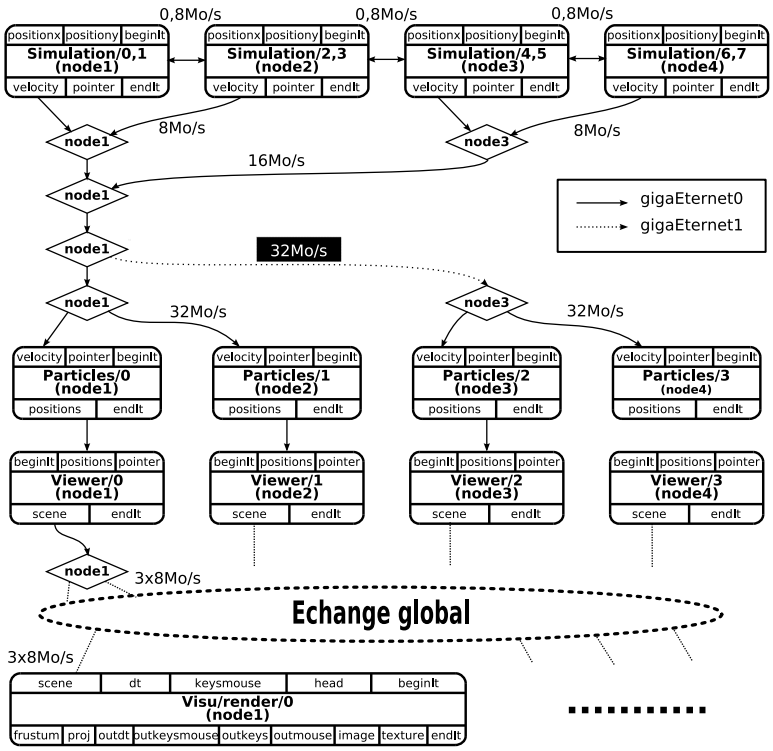

FIG. 12.9 – Schéma de communication utilisant deux réseaux

bi-processeurs étant moins performants que les noeuds quadri-processeurs, nous libérons ainsi plus de puissance pour les autres utilisateurs de la plateforme. Les prévisions de cette allocation sont présentées dans le tableau 12.10. Notons que les temps d'exécution des modules placés sur les noeuds bi-processeurs sont augmentés du fait de leur moindre puissance. Les résultats obtenus pour ce nouveau placement sont dans ce cas également conformes aux prévisions effectuées.

12.4 Utilisation des contraintes

Nous avons implanté notre problème de placement en utilisant la bibliothèque de programmation Gecode [44] (*Generic Constraint Development Environment*). Ce choix est motivé par sa diffusion sous licence libre, par son interface de programmation accessible en C++ et en Java, ainsi que par ses performances qui sont de l'ordre de celles des solveurs propriétaires les plus connus tels qu'ILOG ou SICStus Prolog.

Notre solveur prend en entrée les modélisations, sous forme de graphes, de l'architecture et de l'application. A partir de ces données nous définissons, comme décrit au chapitre 11, les ensembles de variables, leurs domaines, ainsi que les contraintes associées.

Nous présentons dans ce chapitre les différentes possibilités offertes par notre approche ainsi que les résultats obtenus. Nos exemples sont basés sur notre architecture MiReV et notre application FluidParticle.

12.4.1 Génération de placements

Le premier objectif de notre approche est de générer automatiquement des placements d'applications. Pour cela nous devons fournir les modélisations de notre architecture MiReV et de notre application FluidParticle, ainsi que les charges choisies pour chaque module.

Configuration de l'architecture et de l'application

Les caractéristiques de l'application FluidParticle choisies pour nos tests sont décrites dans le tableau 12.11.

Modules	*# modules*	T_{exec}	LD	T_{it}	LD_{cmin}	LD_c
$Simulation$	8	40	100	40	1	1
$Particles$	4	9	100	40	0.225	1
$Viewer$	4	10	100	40	0.25	1
$Renderer$	4	10	100	10	1	1

TAB. 12.11 – Modules de l'application et performances choisies (temps en ms)

Afin de tester notre approche et d'observer ses limites nous considérons des déploiements uniquement sur le premier sous-ensemble de la grappe constitué de 8 noeuds quadri-processeurs interconnectés par deux réseaux gigaEthernet.

Contraintes définies par l'utilisateur

Une première exécution du solveur produit immédiatement un très grand nombre de placements (près d'un millier en quelques secondes). Beaucoup ne sont cependant pas satisfaisants, par exemple, les modules *Renderer* ne sont pas fixés aux noeuds connectés au mur d'images. Nous devons donc ajouter des contraintes pour obtenir des placements correspondants à nos attentes.

Certains modules liés aux matériels peuvent être placés statiquement sur les noeuds. C'est le cas du module *Interaction* placé sur le noeud $node_5$, ainsi que des instances du module *Renderer* placées sur les quatre premiers noeuds. Notons que lorsqu'un noeud à un seul module fixé sur lui, il est possible de supprimer des symétries en fixant ce module sur un processeur. Nous ajoutons donc les contraintes suivantes :

$$\begin{aligned} cpu_{Interaction} &= cpu_{17} \\ cpu_{Renderer/0} &= cpu_{1} \\ cpu_{Renderer/1} &= cpu_{5} \\ cpu_{Renderer/2} &= cpu_{9} \\ cpu_{Renderer/3} &= cpu_{13} \end{aligned}$$

Il ne reste donc plus qu'à placer les modules *Simulation*, *Particles* et *Viewer*.

Nous ajoutons également des contraintes lexicographiques sur l'ordre des instances de chaque type de module afin de supprimer les symétries. Les instances du module *Renderer* étant déjà

placées il suffit de définir les contraintes suivantes sur les autres modules :

$$\begin{aligned} cpu_{Simulation/0} &\leq cpu_{Simulation/1} \leq \ldots \leq cpu_{Simulation/7} \\ cpu_{Particles/0} &\leq cpu_{Particles/1} \leq \ldots \leq cpu_{Particles/3} \\ cpu_{Viewer/0} &\leq cpu_{Viewer/1} \leq \ldots \leq cpu_{Viewer/3} \end{aligned}$$

Résultats

Le nombre de placements, correspondant aux critères de performance que nous avons défini, est très élevé malgré le nombre de contraintes ajoutées pour réduire l'expace de recherche ce qui nous laisse envisager la possibilité de pouvoir augmenter la taille du problème considéré. Notons que parmi les placements obtenus nous retrouvons le placement que nous avons étudié précédemment et décrit par le tableau 12.8.

12.4.2 Tests des limites de l'application et de l'architecture

Nous pouvons également utiliser le solveur pour tester les limites de notre application. Par exemple, nous souhaitons déterminer s'il est possible d'augmenter la taille de notre système de particules.

Nous modifions à cette fin les volume de données générés par les instances du module *Particles* ainsi que le temps de calcul nécessaire pour effectuer l'advection d'un plus grand nombre de particules. Nous fixons la taille du système de particule à 600×600. Les temps d'exécution et charges des modules *Particles* et *Viewer* sont indiqués dans le tableau 12.12. Le solveur produit à nouveau une multitude de placements. Augmentons encore la taille du système de particules pour atteindre 700×700 éléments. Cette fois-ci le solveur répond immédiatement qu'il n'y a pas de solution. Nous avons donc dépassé la taille critique pour notre système de particule.

Modules	*# modules*	T_{exec}	LD	T_{it}	LD_{cmin}	LD_c
$Simulation$	8	40	100	40	1	1
$Particles$	4	14	100	40	0.3	1
$Viewer$	4	18	100	40	0.48	1
$Renderer$	4	10	100	10	1	1

TAB. 12.12 – Modules de l'application et performances choisies : test 2 (temps en ms, charges en %)

12.4.3 Validation de placements

Le domaine de chaque variable de placement des composants et des connexions peut être restreint à une valeur de manière à imposer un placement. De cette manière il est possible de valider un placement donné par le développeur. Soit le placement est valide et le solveur fournit ce même placement en sortie, soit au moins une contrainte n'est pas vérifiée et le solveur ne fournit pas de solution.

Nous avons ainsi testé le placement que nous avons décrit précédemment dans le tableau 12.10. Le solveur nous confirme la validité de ce placement ainsi que les volumes de données transmis sur les liens réseaux.

La validation de placements est utile car elle permet de valider un placement et de calculer automatiquement les volumes de données échangés.

12.5 Conclusion

La programmation par contrainte représente une solution puissante pour la résolution de notre problème de placement d'applications distribuées sur des architectures de type grappe de PC. En effet notre problème ce traduit naturellement sous forme de contraintes. De plus celles-ci peuvent être modifiées ou ajoutées simplement ce qui permet d'apporter beaucoup de souplesse par rapport à des approches telles que les algorithmes génétiques.

Nous avons exploré avec succès les différentes possibilités offertes par la programmation par contraintes. L'implantation de notre problème à l'aide du solveur Gecode nous permet d'éviter les nombreux déploiements infructueux nécessaires jusqu'alors à l'obtention de placements optimisés. Nous pouvons désormais apporter rapidement des réponses aux questions suivantes : Existe t'il un placement pour mon application sur l'architecture considérée ? Quels sont les placements possibles pour mon application ? Peut-on optimiser le placement pour diminuer le nombre de noeuds requis ? L'ajout d'un réseau supplémentaire est il suffisant pour améliorer les performances de mon application ?

Nous avons cependant remarqué que le temps nécessaire au solveur pour trouver une solution est très variable. Un de nos objectifs est d'étudier plus précisément les paramètres qui peuvent contribuer à une amélioration des performances du solveur.

Cinquième partie

Conclusions et perspectives

Conclusion

La réalisation d'applications de réalité virtuelle toujours plus complexes nous conduit à envisager leur distribution sur des grappes de PC pour fournir la puissance nécessaire à leur exécution. Ces architectures sont généralement hétérogènes de par les spécificités de certains codes qui composent ces applications qui conduisent à l'utilisation de différents matériels, par exemple pour effectuer les calculs de rendu, mais également de par la possibilité d'augmenter la puissance d'une grappe par ajout de noeuds qui conduit à l'intégration de machines de générations différentes. Cette hétérogénéité est également présente au niveau des systèmes d'exploitation utilisées sur une même architecture, certains codes, souvent les pilotes de périphériques, nécessitant des systèmes spécifiques.

Si l'utilisation d'intergiciels tels que FlowVR facilite la conception et le couplage des codes sur plateformes hétérogènes, la performance offerte par l'application dépend du choix de placement des codes et des connexions respectivement sur les noeuds et les réseaux. Cette tâche reste cependant à la charge du concepteur qui doit prendre en compte les paramètres de l'architecture et de l'application pour déterminer un placement offrant les performances attendues.

Nous avons donc présenté tout l'intérêt d'associer un modèle de performance à FlowVR pour évaluer les choix de placement du concepteur. L'étude des modèles de performance classiques, tels que BSP ou LogP, nous a montré que ceux-ci considèrent des modèles de programmation ou des architectures homogènes qui ne correspondent pas au cadre de notre étude.

Nous proposons donc un nouveau modèle de performance capable de s'appliquer aux applications hétérogènes déployées sur des architectures également hétérogènes. Ce modèle nécessite en entrée les modélisations de l'application, de l'architecture, et du placement. La modélisation de l'application est basée sur sa représentation FlowVR enrichie d'informations de performance sur ses différents modules que le concepteur doit fournir, et qui sont généralement déjà disponible car FlowVR facilite la réutilisation de modules. Nous fournissons également une modélisation des grappes de PC capable de représenter des noeuds SMP dotés de multiples adapteurs réseau, interconnectés suivant des topologies complexes. Finalement nous modélisons le placement comme une association des composants sur les noeuds de l'architecture et des connexions sur les liens des différents réseaux.

Ces informations nous permettent d'étudier les effets de la synchronisation, de la concurrence et des communications entre les différents codes de l'application sur leurs performances. Notre objectif étant de calculer les fréquences de ces codes ainsi que les latences entre eux afin de vérifier que l'application se comporte de manière interactive.

Les synchronisations sont données par le graphe de l'application soit explicitement par la présence de *synchroniseurs* soit implicitement par les communications. Nous avons montré que l'étude de ce graphe nous permet de détecter des problèmes de configuration de l'application indépendamment du placement de l'application. Pour les résoudre nous pouvons proposer, suivant les cas, soit l'ajout d'éléments de synchronisation ou au contraire le relâchement de certaines synchronisations.

Afin de déterminer l'effet de la concurrence sur les performances des modules nous avons choisi de modéliser le fonctionnement de l'ordonnanceur du noyau Linux. Cette solution ne s'est pas révélée satisfaisante car la politique de cet ordonnanceur est dynamique et basée sur des informations locales, elle n'offre donc aucune garantie d'obtention des performances escomptées. Nous avons donc proposé d'effectuer une allocation statique des modules sur les

processeurs guidée par une étude globale de l'application et par les performances attendues par l'utilisateur. Cette approche permet ainsi de garantir les performances et de remédier aux problèmes engendrés par un ordonnancement dynamique et local. Pour effectuer cette allocation nous déterminons le processeur sur lequel placer chaque module ainsi que la charge à lui attribuer sur celui-ci pour atteindre la performance attendue. Cependant une telle allocation n'est pas toujours possible. Dans ce cas nous pouvons localiser les noeuds surchargés et proposer de déplacer certains modules sur d'autres noeuds. Ce processus d'allocation nous permet donc de calculer la charge totale de chaque processeur et de déterminer le taux d'utilisation de l'architecture.

L'étude des synchronisations et de la concurrence entre les composants nous permet ensuite de déterminer les fréquences de ceux-ci et de vérifier qu'elles correspondent effectivement aux performances attendues. A partir des fréquences, et connaissant le volume de données envoyé par chaque composant, nous pouvons ainsi calculer les volumes de données transitant sur chaque lien réseau et détecter leur éventuelle saturation. Cela nous permet également d'obtenir les temps de communication des messages. Nous pouvons finalement calculer la latence entre les composants, c'est à dire le temps que met une information pour être traitée par les composants et transmise par les réseaux, pour vérifier que sa valeur permet une exécution interactive de l'application.

Notre modèle permet d'évaluer les performances d'un placement donné. Si ce placement ne convient pas, nous pouvons en identifier les raisons et conseiller le concepteur sur les modifications à effectuer pour améliorer son placement. Cependant, la recherche d'un placement est un processus long car de nombreux paramètres sont à prendre en considération et nos recommendations ne garantissent pas l'obtention d'un meilleur placement. Par exemple, déplacer un module sur un noeud différent pour lui allouer plus de ressources permet d'améliorer ses performances, par contre cela change le volume des communications sur les réseaux et peut engendrer une saturation du réseau.

Afin de décharger le concepteur de la recherche d'un placement convenable, nous proposons d'automatiser cette tâche par l'utilisation d'outils basés sur la programmation par contraintes. Nous avons montré que notre problème de placement s'exprime naturellement sous forme de contraintes. De plus, cette approche apporte de nombreuses possibilités : il est possible de l'utiliser pour évaluer, générer, optimiser des placements, mais également pour prévoir la configuration matérielle nécessaire à l'exécution d'une application. Le concepteur d'applications de réalité virtuelle a donc à sa disposition un modèle ainsi que des outils pour l'assister dans le déploiement de ses applications sur des architectures de type grappes de PC.

Perspectives

Le modèle que nous proposons n'est pas intrèsequement limité aux applications de réalité virtuelles et aux architectures de type grappes de PC. Nous pouvons donc envisager de l'appliquer à des application distribuées déployées sur des architectures de type grilles de calcul. En effet notre modèle permet de modéliser des noeuds hétérogènes ainsi que des réseaux interconnectés suivant des topologies complexes. Notre modèle est également suffisamment souple pour permettre son utilisation sur d'autres intergiciels que FlowVR. On pourrait par exemple étudier son application à CORBA ou à OpenMASK.

Concernant le modèle d'allocation dirigée par les performances que nous proposons, celui-ci permet de s'abstraire du système d'exploitation, cependant pour assurer cette allocation le

paramètrage de l'ordonnanceur du système d'exploitation reste à la charge du développeur. Afin d'abstraire complètement cette tâche, une solution serait de proposer un ordonnanceur FlowVR effectuant ce paramètrage de manière transparente pour l'utilisateur. Des contraintes permettraient ainsi de définir automatiquement une allocation minimale et qui constituerait une base d'optimisation pour le développeur.

La résolution de notre problème est effectuée par un solveur de contraintes. Nous devons maintenant étudier plus précisément le comportement du solveur lors de la résolution de problèmes de plus grandes tailles, c'est à dire des applications composées d'un plus grand nombre de composants et des architectures plus vastes et hétérogènes. De nombreuses améliorations sont envisagées pour optimiser ce passage à l'échelle, citons par exemple : la recherche d'heuristiques sur l'ordre de branchement des variables ; l'optimisation de la propagation ou encore la détection automatique de symétries à partir du graphe de l'application. Ces optimisations nous permettraient d'envisager le déploiement d'applications à partir d'une description de l'application de plus haut niveau, comme celle proposée par Lesage [32] pour FlowVR, c'est à dire demander au solveur de placer les différents modules de l'application, mais également de trouver le nombre d'instances ainsi que les schémas de communication optimisés nécessaire à l'obtention de performances interactives.

Afin de faciliter le déploiement d'applications distribuées nous envisageons également la création d'outils graphiques pour assister l'utilisateur sans que celui-ci ait besoin de connaître les détails de l'architecture ni de l'application. Ces outils permettraient ainsi de favoriser la mise en place d'applications de réalité virtuelle sans le recours à des spécialistes.

Bibliographie

[1] The FlowVR library. http://flowvr.sourceforge.net.

[2] The GNOME ORBit2 project. http://www.gnome.org/projects/ORBit2.

[3] OMG : The Object Management Group. http://www.omg.org.

[4] OmniORB. http://omniorb.sourceforge.net.

[5] OpenMASK. http://www.irisa.fr/siames/OpenMASK/.

[6] The scheduler utilities. http://rlove.org/schedutils/.

[7] Top500 Supercomputing Sites. http://www.top500.org.

[8] J. Aas : Understanding the Linux 2.6.8.1 CPU Scheduler. http ://citeseer.ist.psu.edu/aas05understanding.html.

[9] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser et Chris Scheiman : LogGP : Incorporating long messages into the LogP model for parallel computation. *Journal of Parallel and Distributed Computing*, 44(1):71–79, 1997.

[10] J. Allard : *FlowVR : calculs interactifs et visualisation sur grappe*. Thèse de doctorat, Institut National Polytechnique de Grenoble, 2005.

[11] J. Allard, V. Gouranton, L. Lecointre, S. Limet, E. Melin, B. Raffin et S. Robert : FlowVR : a Middleware for Large Scale Virtual Reality Applications. *In Proceedings of Euro-par 2004*, Pisa, Italia, August 2004.

[12] J. Allard, V. Gouranton, E. Melin et B. Raffin : Parallelizing pre-rendering computations on a Net Juggler PC cluster. *In IPTS 2002*, 2002.

[13] J. Allard, C. Ménier, E. Boyer et B. Raffin : Running large VR applications on a PC cluster : the FlowVR experience. *In Proceedings of EGVE/IPT 05*, Denmark, October 2005.

[14] M. Ammi et A. Ferreira : Virtualized reality interface for tele-micromanipulation. *In IEEE Int. Conf. on Robotics and Automation*, New-Orleans, LA, USA, 2004.

[15] A. L. Ananda, B. H. Tay et E. K. Koh : A survey of asynchronous remote procedure calls. *SIGOPS Oper. Syst. Rev.*, 26(2):92–109, 1992.

[16] K. Apt : *Principles of Constraint Programming*. Cambridge University Press, 2003.

[17] B. Arnaldi, P. Fuchs et J. Tisseau : *Le traité de la réalité virtuelle*, volume 1, chapitre 1. Les Presses de l'Ecole des Mines de Paris, 2003.

[18] N. Barnier, P. Brisset et T. Rivière : Slot allocation with constraint programming : Models and results. *In the Fourth International Air Traffic Management R&D Seminar ATM*, Sante Fe, New Mexico, 2001.

[19] N. Beldiceanu, M. Carlsson et Jean-Xavier Rampon : Global constraint catalog, 2005. http ://www.lina.sciences.univ-nantes.fr/Publications/2005/BCR05.

[20] Allen BIERBAUM, Christopher JUST, Patrick HARTLING, Kevin MEINERT, Albert BAKER et Carolina CRUZ-NEIRA : VR Juggler : A virtual platform for virtual reality application development. *In VR '01 : Proceedings of the Virtual Reality 2001 Conference (VR'01)*, page 89, Yokohama, Japan, 2001. IEEE Computer Society.

[21] D. P. BOVET et M. CESATI : *Understanding the Linux Kernel, Third Edition*, chapitre 7. Oreilly, 2005.

[22] G. CAVALHEIRO, F. GALILEE et J.-L. ROCH : Athapascan-1 : Parallel programming with asynchronous tasks. *In Proceedings of the Yale Multithreaded Programming Workshop*, Yale, June 1998.

[23] C. CRUZ-NEIRA, D.J. SANDIN, T.A. DEFANTI, R.V. KENYON et J.C. HART : *The CAVE : Audio Visual Experience Automatic Virtual Environment*, volume 35. Communications of the ACM, june 1992.

[24] David E. CULLER, Richard M. KARP, David A. PATTERSON, Abhijit SAHAY, Klaus E. SCHAUSER, Eunice SANTOS, Ramesh SUBRAMONIAN et Thorsten von EICKEN : LogP : Towards a realistic model of parallel computation. pages 1–12, 1993.

[25] F. DEVILLERS, S. DONIKIAN, F. LAMARCHE et J. TAILLE : A programming environment for behavioural animation. *Journal of Visualization and Computer Animation*, 13(5):263–274, 2002.

[26] S. DONIKIAN, A. CHAUFFAUT, T. DUVAL et R. KULPA. : Gasp : from modular programming to distributed execution. *In Computer Animation'98, IEEE*, Philadelphia, USA, june 1998.

[27] J.M.Vincent F. ZARA, F. Faure : Physical cloth simulation on a pc cluster. *In Eurographics Workshop on Parallel Graphics and Visualization*, 2002.

[28] P. FUCHS, G. MOREAU et J. TISSEAU : *Le traité de la réalité virtuelle*, volume 3, chapitre 12, pages 335–368. Les Presses de l'Ecole des Mines de Paris, 3 édition, 2007.

[29] R. GAUGNE, S. JUBERTIE et S. ROBERT : Distributed multigrid algorithms for interactive scientific simulations on clusters. *In Online Proceeding of the 13th International Conference on Artificial Reality and Telexistence, ICAT*, december 2003.

[30] Amy HENDERSON : *The ParaView Guide*. Kitware, Inc. publishers, 2004.

[31] T. C. HUDSON, A. SEEGER, H. WEBER, J. JULIANO et A. T. HELSER : VRPN : a device-independent, network-transparent VR peripheral system. *In ACM Symposium on Virtual Reality Software & Technology*, 2001.

[32] B. Raffin J.-D. LESAGE : A hierarchical programming model for large parallel interactive applications. *In Proceedings of IFIP International Conference on Network and Parallel Computing, NPC,*, Dalian, China, september 2007.

[33] S. JUBERTIE et E. MELIN : Multiple networks for heterogeneous distributed applications. *In* Hamid R. ARABNIA, éditeur : *Proceedings of PDPTA'07*, pages 415–424, Las Vegas, june 2007. CSREA Press.

[34] S. JUBERTIE et E. MELIN : Performance prediction for mappings of distributed applications on pc clusters. *In Proceedings of IFIP International Conference on Network and Parallel Computing, NPC,*, Dalian, China, september 2007.

[35] Gabriel LOH : A critical assessment of logp : Towards a realistic model of parallel computation. 2000.

[36] D. MARGERY : *Environnement logiciel temps-réel distribué pour la simulation sur réseau de PC*. Thèse de doctorat, Université de Rennes 1, 2001.

[37] D. MARGERY, B. ARNALDI, A. CHAUFFAUT, S. DONIKIAN et T. DUVAL : OpenMASK : Multi-Threaded or Modular Animation and Simulation Kernel or Kit : a General Introduction. *In* B. Taravel (editors) S. RICHIR, P. Richard, éditeur : *VRIC 2002*, june 2002.

[38] E. MELIN : *Traitement de l'Irrégularité dans la Parallélisation de Code Séquentiel par Distribution des Données*. Thèse de Doctorat d'Université, Université d'Orléans, Février 1998.

[39] Eric Charles OLSON : Cluster juggler – pc cluster virtual reality. Mémoire de D.E.A., Iowa State University, 2002.

[40] X. REBEUF : *Un modèle de coût symbolique pour les programmes paralléles asynchrones à dépendances structurées*. Thèse de Doctorat d'Université, Université d'Orléans, décembre 2000.

[41] José L. RODA, Casiano RODRÍGUEZ, Daniel GONZÁLEZ-MORALES et Francisco ALMEIDA : Predicting the execution time of message passing models. *Concurrency - Practice and Experience*, 11(9):461–477, 1999.

[42] B. SCHAEFFER et C. GOUDESEUNE : Syzygy : Native pc cluster vr. *In IEEE VR Conference*, 2003.

[43] Will SCHROEDER, Ken MARTIN et Bill LORENSEN : *The Visualization Toolkit*. Kitware, Inc. publishers, 3 édition, 2004.

[44] Christian SCHULTE et Guido TACK : Views and iterators for generic constraint implementations. *In Recent Advances in Constraints*, volume 3978 de *Lecture Notes in Artificial Intelligence*, pages 118–132. Springer-Verlag, 2006.

[45] Chris SHAW, Mark GREEN, Jiandong LIANG et Yunqi SUN : Decoupled simulation in virtual reality with the MR toolkit. *Information Systems*, 11(3):287–317, 1993.

[46] Chris SHAW, Jiandong LIANG, Mark GREEN et Yunqi SUN : The decoupled simulation model for virtual reality systems. *In CHI '92 : Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 321–328, New York, NY, USA, 1992. ACM Press.

[47] J. STAM : Real-time fluid dynamics for games. *In Proceedings of the Game Developer Conference*, March 2003.

[48] Andrew S. TANENBAUM : *Modern Operating Systems*. Prentice Hall, 2001.

[49] Vildan TANRIVERDI et Robert J.K. JACOB : Vrid : a design model and methodology for developing virtual reality interfaces. *In VRST '01 : Proceedings of the ACM symposium on Virtual reality software and technology*, pages 175–182, New York, NY, USA, 2001. ACM Press.

[50] A. R. TRIPATHI et T. NOONAN : Design of a Remote Procedure Call system for object-oriented distributed programming. *Software-Practice and Experience*, 28(1):23–48, 1998.

[51] Leslie G. VALIANT : A bridging model for parallel computation. *Commun. ACM*, 33(8): 103–111, 1990.

[52] VRCO. *Trackd User Guide*. http://www.vrco.com/trackd/trackd_Documentation.html.

[53] Rong YANG : Solving a workforce management problem with constraint programming. *In The 2nd International Conference on the Practical Application of Constraint Technology*, pages 373–387. The Practical Application Company Ltd, 1996.

[54] C. El ZAMMAR : *Interactions coopératives 3D distantes en environnements virtuels : gestion des problèmes réseau*. Thèse de doctorat, Institut National des Sciences Appliquées de Rennes, 2005.

Sixième partie

Annexes

Annexe A

Expérimentations

```
<?xml version="1.0" encoding="LATIN1"?>
<!DOCTYPE metamoduledesc
    SYSTEM "http://flowvr.sf.net/flowvr-parse/dtd/metamoduledesc.dtd" >

<metamoduledesc>
<metamoduledescription>
    <run id="ssh" shell="/bin/bash">
      flowvr-run-ssh -L -p '<hostlist/>'
        <template id="cmd"> </template> ./../bin/generic
          <template id="sleep"/>
          <template id="calc"/>
          <template id="in"/>
          <template id="out"/>
          <template id="outports"> </template>
    </run>
<modulelist>
<script shell="/bin/bash">
let module_rank=0;
for host_name in ''<hostlist/>
do
    echo '&lt;module id="<metamoduleid/>/'$module_rank'" host="'$host_name'"&gt;';

    if [[ <template id="in"/> -gt 0 ]]
    then
      echo '&lt;input&gt;'
      for (( i=0 ; $i&lt;<template id="in"/> ; i++ ));
      do
        echo '&lt;port id="in'$i'"&gt;&lt;datatype/&gt;&lt;/port&gt;';
      done;
      echo '&lt;/input&gt;'
    fi

    if [[ <template id="out"/> -gt 0 ]]
    then
      echo '&lt;output&gt;'
      for (( i=0 ; $i&lt;<template id="out"/> ; i++ ));
      do
        echo '&lt;port id="out'$i'"&gt;&lt;datatype/&gt;&lt;/port&gt;';
      done;
      echo '&lt;/output&gt;'
    fi

    echo '&lt;/module&gt;'

    let module_rank+=1;

done
</script>
</modulelist>
</metamoduledescription>
</metamoduledesc>
```

FIG. A.1 – Fichier de description du module générique

```
<?xml version="1.0" encoding="LATIN1" ?>
<!DOCTYPE metamodulelist
    SYSTEM "http://flowvr.sf.net/flowvr-parse/dtd/metamodulelist.dtd">

<metamodulelist>

    <metamodule id="module1">
        <metamoduledescription>generic.desc.xml</metamoduledescription>
        <template id="sleep">2000</template>
        <template id="calc">10000</template>
        <template id="in">0</template>
        <template id="out">1</template>
        <template id="outports">100000</template>
        <host>node1</host>
        <run id="ssh" />
        <modulelist/>
    </metamodule>

    <metamodule id="module2">
        <metamoduledescription>generic.desc.xml</metamoduledescription>
        <template id="sleep">1000</template>
        <template id="calc">8000</template>
        <template id="in">1</template>
        <template id="out">0</template>
        <host>node2</host>
        <run id="ssh" />
        <modulelist/>
    </metamodule>

</metamodulelist>
```

FIG. A.2 – Exemple de fichier de configuration d'une application basée sur le module générique

```
#!/usr/bin/perl -w
use strict;
use FlowVR::XML ':all';

&parseInput;

&addSimpleConnection('module1/0', 'out0', 'module2/0', 'in0');

&printResult;
```

FIG. A.3 – Exemple de définition du schéma de communication et de couplage d'une application

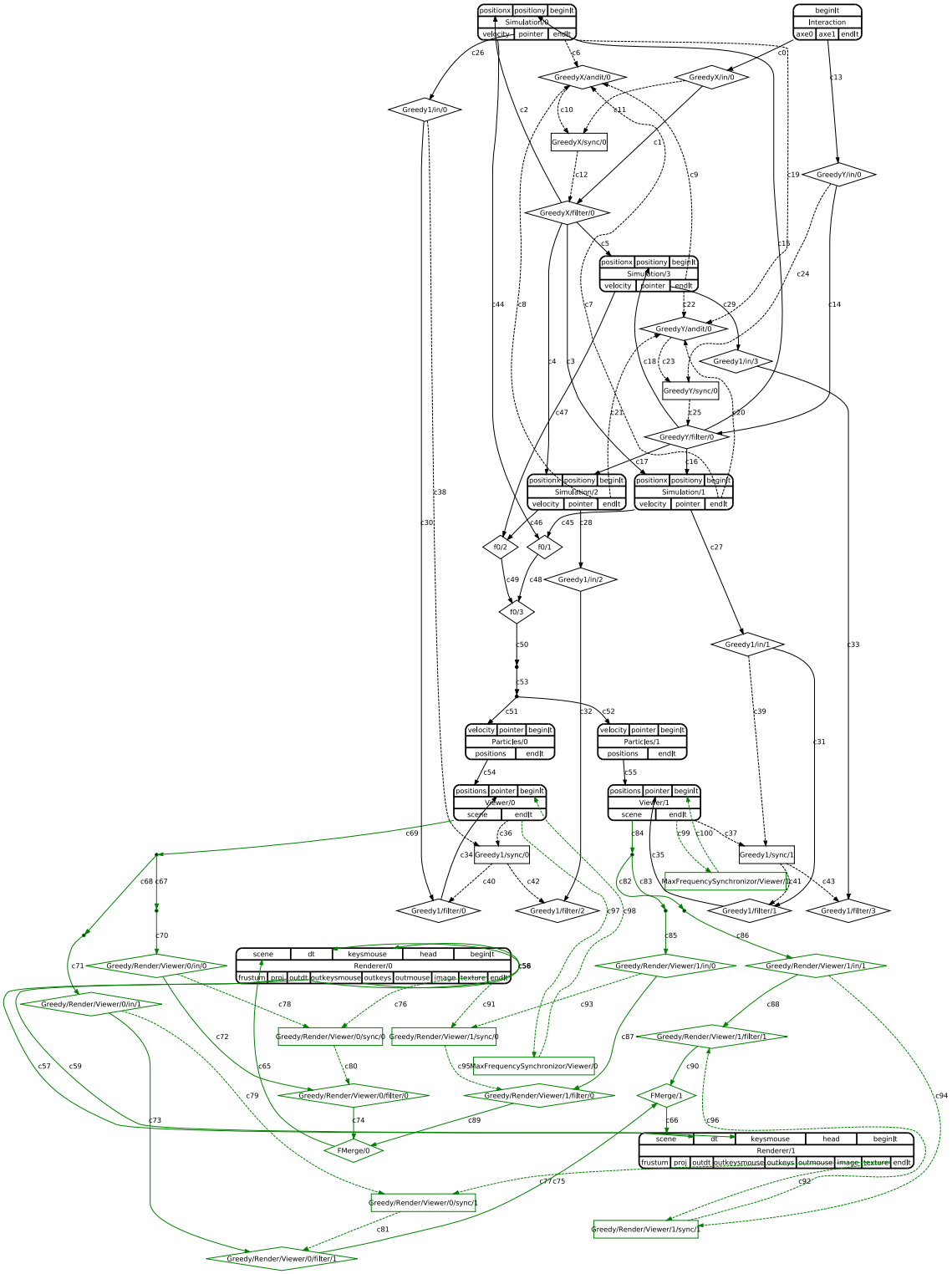

FIG. A.4 – Graphe de l'application FluidParticle (Simulation : 4 instances, Particles : 2, Viewer : 2, Renderer : 2)

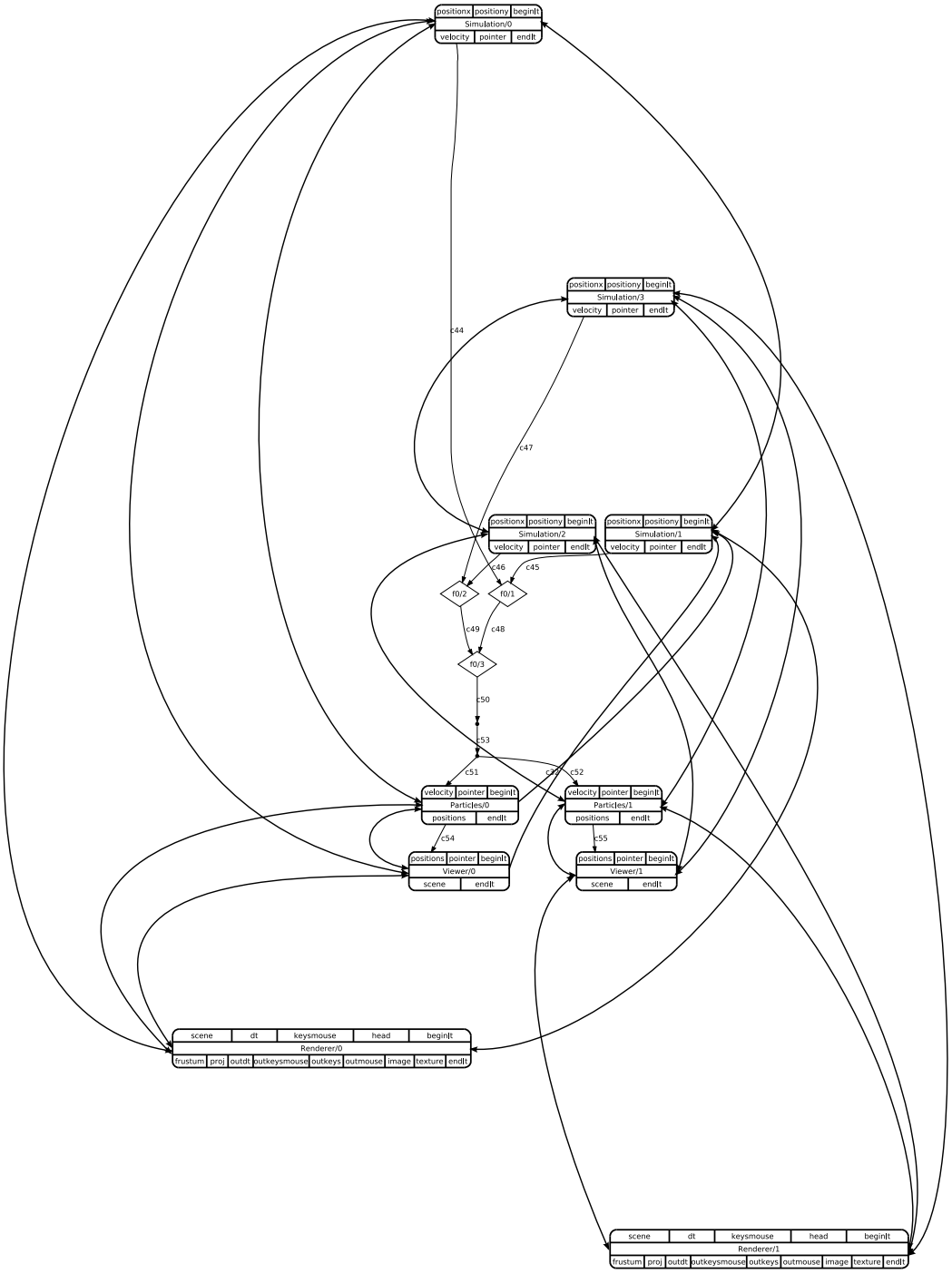

FIG. A.5 – Graphe Gdep de l'application FluidParticle (Simulation : 4 instances, Particles : 2, Viewer : 2, Renderer : 2)

Modèles et outils pour le déploiement d'applications de Réalité Virtuelle sur des architectures distribuées hétérogènes

Les applications de Réalité Virtuelle requièrent une puissance de calcul importante qui peut être apportée par les grappes de PC, des ensembles d'ordinateurs connectés par des réseaux performants. Afin d'exploiter la puissance de ces architectures, une approche consiste à décomposer les applications en plusieurs composants qui sont ensuite déployés sur les différentes machines. Les performances de telles applications dépendent alors du matériel ainsi que des synchronisations entre les différents composants. Evaluer les performances d'une application de RV suivant un déploiement donné consiste à observer si son exécution permet l'interactivité. Cependant, cette phase de test rend la recherche d'un déploiement répondant à ce critère longue et fastidieuse et monopolise l'architecture. Nous proposons donc de définir un modèle permettant l'évaluation des performances à partir de la modélisation de l'architecture, de l'application et de son déploiement. Nous proposons ensuite d'utiliser la programmation par contraintes pour résoudre les contraintes extraites de notre modèle et permettre ainsi d'automatiser la génération de déploiements capables de fournir le niveau d'interactivité souhaité. Cette approche permet ainsi de répondre aux nombreuses questions que peut se poser un développeur : Existe t'il un ou plusieurs déploiements de mon application permettant l'interactivité sur mon architecture ? Si oui, quels sont ils ? L'ajout de machines supplémentaires permet il un gain de performances ?

Models and tools for the mapping of Virtual Reality applications on distributed heterogeneous architectures

Virtual Reality applications require a huge amount of computational power that clusters, sets of computers connected with networks, are able to provide. To take advantage of these architectures, it is possible to split applications into several parts, called components, and to map them on different cluster nodes. Performance of such applications depends on hardware performance, on their mapping and also on the synchronization and communication schemes between components. To determine if a VR application can run in an interactive way, we can map and run it on the architecture. If the application does not perform as expected we have to try another mapping. However, it is often a long and tedious process before finding a mapping with expected performance. To speed up this process we define a performance model which enables to evaluate performance of a given mapping for a distributed application on a cluster from architecture, application, and mapping descriptions. Then, we propose an approach based on constraint programming to automatically generate mappings. Constraints are defined from our model, from performance of the architecture and also from performance expected by the user. This approach enables to answer the following answers : Does at least one mapping exists with the expected performance on the given architecture ? If it does then what are these mappings ? Does the application performs better if we increase the number of nodes of the architecture ?

Laboratoire d'Informatique Fondamentale d'Orléans - Université d'Orléans
6, rue Léonard de Vinci - BP 6759 - 45067 ORLEANS CEDEX 2