Couplage dans les applications interactives de grande taille

Jean-Denis Lesage

▶ To cite this version:

Jean-Denis Lesage. Couplage dans les applications interactives de grande taille. Réseaux et télécommunications [cs.NI]. Institut National Polytechnique de Grenoble - INPG, 2009. Français. NNT : . tel-00459665

HAL Id: tel-00459665
https://theses.hal.science/tel-00459665

Submitted on 24 Feb 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT POLYTECHNIQUE DE GRENOBLE

N° attribué par la bibliothèque

|_|_|_|_|_|_|_|_|_|_|

THÈSE

pour obtenir le grade de **DOCTEUR DE L' Institut Polytechnique de Grenoble**
Spécialité : « Informatique »

préparée au laboratoire Laboratoire d'Informatique de Grenoble dans le cadre de **l'École Doctorale « Mathématiques, Sciences et Technologies de l'Information, Informatique »**

préparée et soutenue publiquement par

Jean-Denis Lesage

le 26/11/2009

Titre :

Couplage dans les applications interactives de grande taille

***sous la direction de M. Denis* Trystram**
***co-encadré par M. Bruno* Raffin**

JURY

M. Jocelyn Serot	, Président
Mme. Françoise Baude	, Rapporteur
M. Torsten Kuhlen	, Rapporteur
M. Denis Trystram	, Directeur de Thèse
M. Bruno Raffin	, Co-encadrant

A Chama et à mes parents...

Remerciements

Je souhaite remercier Bruno Raffin qui m'a encadré durant ma thèse et mon master. J'ai vraiment apprécié les 4 années de travail ensemble. Je le remercie de m'avoir fait confiance et de m'avoir impliqué dans des projets que j'ai trouvé passionnants. Un immense merci pour toute l'aide qu'il m'a apportée.

Je souhaite remercier les membres de mon jury. Merci à Françoise Baude et Torsten Kuhlen d'avoir accepté d'être rapporteurs. Je remercie aussi mon président Jocelyn Serot. Je remercie enfin Denis Trystram pour son aide et son soutien durant ma thèse.

Je remercie les personnes qui ont relu ma thèse à la recherche des dernières coquilles.

Je remercie mes collègues. Les membres des équipes FlowVR et Grimage : Thomas, Benjamin, Luciano, Antoine, Ingo, ... Merci à mes amis du laboratoire ex-ID ou de l'INRIA et particulièrement Pedro, Marin, Marc, Pof, PEPE, ... et tous ceux avec qui j'ai pu partagé quelques bières à Grenoble. Je remercie les amis avec qui j'ai écumé les bassins de la piscine de Montbonnot le midi.

Enfin, un immense merci à ma femme Chama, mes parents et ma famille pour leur soutien.

Table des matières

1 Introduction 1
- 1.1 Contexte . . . 1
- 1.2 Problématique . . . 2
- 1.3 Organisation du manuscrit . . . 3
- 1.4 Travail réalisé . . . 3

I Etat de l'art 7

2 Applications interactives 9
- 2.1 Introduction . . . 9
- 2.2 La boucle d'interaction . . . 9
- 2.3 Limites du modèle . . . 10
- 2.4 Modèle avec des boucles d'interactions multiples . . . 11
- 2.5 Exemples d'applications motivant le modèle . . . 12
- 2.6 Bilan . . . 15

3 Problématiques autour des applications interactives 17
- 3.1 Introduction . . . 17
- 3.2 Modèle de flots de données pour l'interactivité . . . 17
- 3.3 Synchronisation entre boucles . . . 20
- 3.4 Utilisation du parallélisme . . . 21
  - 3.4.1 Partage des ressources entre les différentes formes de parallélisme . . . 24
  - 3.4.2 Couplage de codes parallèles . . . 25
- 3.5 Bilan . . . 26

4 Optimisation des performances des applications interactives 27
- 4.1 Introduction . . . 27
- 4.2 Métriques pour les applications interactives . . . 27
- 4.3 Problème d'optimisation . . . 28

4.3.1 Satisfaction de contraintes 28
4.3.2 Problème d'optimisation 29
4.4 Bilan 30

II Modèle de composants hiérarchiques 33

5 Introduction 35
5.1 Présentation de FlowVR 35
5.2 Fonctionnement de FlowVR 36
5.2.1 Messages 37
5.2.2 Modules 38
5.2.3 Compilation et déploiement de l'application 38
5.2.4 Objets formant le réseau FlowVR 39
5.3 Bilan 41

6 Hiérarchie de composants 43
6.1 Motivation 43
6.2 Interface 44
6.3 Type de composants 44
6.4 Les liens 45
6.5 Application sur un exemple 45
6.6 Contrôleur 49
6.7 *Traverse* 51
6.8 Complexité 54
6.9 Caractérisation de $\overline{E_\infty}$ 55
6.10 Modification du chemin de compilation 56
6.11 Bilan 57

7 Modification de la politique d'envoi des messages 61
7.1 Bilan 63

8 Discussion 71
8.1 Introduction 71
8.2 Utilisation du modèle hiérarchique 71
8.3 Une librairie C++ comme ADL 73
8.4 Visualisation de graphes hiérarchiques 73
8.5 Interface graphique de conception de réseau 75
8.6 Librairie de schémas de communication 76
8.7 Utilisation du modèle hiérarchique dans l'application Grimage . . . 77
8.8 Ports Composites 78

8.9 Bilan 81

III Couplage de tâches itératives 83

9 Introduction 85
9.1 Motivations 85
9.2 Applications interactives 86
9.3 Echantillonnage 87
9.4 Sources d'échantillonnage 88
9.5 *Synchronization Lag* 90
9.6 Bilan 92

10 Etude de l'échantillonneur simple 93
10.1 Introduction 93
10.2 Etude de q_n 94
10.3 Comparaison avec les générateurs pseudo-aléatoires 96
10.4 Moyenne et variance du temps d'attente 98
10.5 Vérification expérimentale 99
10.5.1 Dispositif expérimental 99
10.5.2 Validation de la loi sur q_n 100
10.5.3 Validation de la loi sur la moyenne et la variance 100
10.6 Bilan 100

11 Mécanismes permettant de limiter l'impact de l'échantillonnage 105
11.1 Introduction 105
11.2 Un critère pour supprimer la gigue 105
11.2.1 Discussion sur le respect du critère 108
11.3 Echantillonneur à seuil 109
11.3.1 Description de l'échantillonneur 109
11.3.2 Moyenne et variance de l'attente d'un message 111
11.3.3 Inactivité de la `Destination` 112
11.4 Bilan 112

12 Régulation pour les applications interactives 117
12.1 Introduction 117
12.2 Parallélisme de pipeline 118
12.3 Comparaisons entre les méthodes UDP et TCP 119
12.3.1 VTK 119
12.3.2 Dispositif expérimental 121
12.3.3 Résultats 122

12.4 Efficacité des calculs . . . 123
12.5 Bilan . . . 127

13 Discussion **129**
13.1 Choix des fréquences . . . 129
13.2 Un énoncé du problème de couplage . . . 130
13.3 Démarches scientifiques pour résoudre le problème de couplage . . . 131

IV Conclusion **133**

14 Conclusion **135**
14.1 Bilan . . . 135
14.2 Perspectives . . . 136

Bibliographie **139**

Chapitre 1

Introduction

1.1 Contexte

D'après la définition du Larousse, l'interactivité est la "*faculté d'échange entre l'utilisateur d'un système informatique et la machine par l'intermédiaire d'un terminal doté d'un écran de visualisation.*". Plusieurs pistes permettent d'améliorer l'interactivité. Certains systèmes proposent des interactions dites naturelles. Dans ces systèmes, l'utilisateur réalise naturellement des actions, le système a la charge de les comprendre et de les interpréter. Il n'est plus nécessaire d'utiliser un langage ou du matériel spécifique pour dialoguer avec la machine. La console Nintendo Wii ou l'écran tactile de l'iPhone sont des exemples disponibles pour le grand public de ce type d'interaction.

Un autre moyen d'améliorer l'interactivité est d'améliorer l'immersion du système. L'immersion peut être donnée grâce à des simulations physiques précises, des rendus visuels réalistes,... Dans certains cas, ces aspects sont critiques. Un simulateur de vol pour l'entraînement de futurs pilotes doit être immersif. Il est nécessaire que l'utilisateur retrouve dans l'avion les sensations qu'il a connues lors de l'entraînement.

Dans tous les cas, pour proposer ces améliorations, les systèmes utilisent une capacité de calculs de plus en plus importante. Les algorithmes qui implémentent les interactions naturelles ou les simulations de phénomènes naturels sont coûteux. Ces algorithmes doivent être capables d'atteindre des temps d'exécution de la milliseconde à la seconde. Ces systèmes interactifs peuvent s'appuyer sur l'évolution des moyens de calculs pour proposer des interactions de plus en plus complexes.

Nous cherchons à garantir la puissance de calculs nécessaire pour l'exécution des algorithmes interactifs.

1.2 Problématique

Les applications interactives de grande taille couplent des centaines de tâches. Ces tâches sont des boucles itératives effectuant des calculs sur des flux de données. Ces systèmes sont complexes. Ils mêlent des codes parallèles utilisant des périphériques d'entrées-sorties spécifiques. Ils doivent respecter des contraintes de performances pour garantir l'interactivité. Les applications interactives haute-performance sont souvent exécutées sur des architectures distribuées et hétérogènes.

Dans les applications interactives, l'utilisation du parallélisme est souvent limitée à certains contextes précis. Les applications de visualisation scientifique travaillent sur des gros jeux de données et utilisent le parallélisme de données pour accélérer les traitements. En réalité virtuelle, le parallélisme est essentiellement utilisé pour contrôler les périphériques de rendus. Ces périphériques sont souvent composés de plusieurs surfaces qui doivent être alimentées par des images différentes. Un casque stéréo de réalité virtuelle stéréo est un exemple de périphérique comportant deux surfaces de rendus. Les images sont quelques fois produites par plusieurs cartes graphiques fonctionnant en parallèle. Depuis quelques années, la communauté utilise les cartes graphiques pour la parallélisation de leurs algorithmes. L'apparition de nouveaux langages de programmation comme les shaders, CUDA ou OpenCL favorise cette évolution.

Des équipes de chercheurs essaient de démontrer que le parallélisme, s'il était utilisé plus largement, permettrait d'implémenter de nouvelles formes d'interactions plus complexes. Elles construisent des applications de grande taille utilisant la puissance de calcul de clusters de PC ou de grilles de calculs. Ces systèmes utilisent des techniques avancées de programmation parallèle. Ces applications démontrent que de nouvelles interactions peuvent dès aujourd'hui être expérimentées. Le développement de ces applications est un processus complexe. Ce sont des projets longs initiés depuis plusieurs années où des équipes de chercheurs venant de communautés différentes coopèrent pour le développement des applications.

L'adoption des techniques avancées de programmation parallèle dans l'ensemble des applications interactives pose aujourd'hui des problèmes scientifiques importants. Les middlewares pour le développement d'applications interactives ne proposent pas un support évolué du parallélisme. C'est un frein à une utilisation plus large du parallélisme et des architectures nouvelles dans les applications interactives. Notre objectif est d'intégrer ce support dans les middlewares. Le développement de middleware performant permettra le développement d'applications interactives hautes-performances. Ces applications mettront en oeuvre de nouvelles formes d'interactions plus complexes.

L'éventail des questions à étudier est assez large. Nous nous sommes concentrés sur les méthodes améliorant les performances des applications. Cependant avant

d'attaquer ces questions, il est nécessaire de pouvoir maîtriser ces applications de grande taille qui motivent notre travail. Ces applications sont composées de centaines de briques logicielles. Elles s'exécutent souvent sur des architectures hétérogènes et distribuées. Les questions de génie logiciel et de méthodologie nous semblent indissociables de la construction d'algorithmes parallèles. Pour ces raisons, nous avons étudié en particulier deux questions :

- Comment décrire le couplage entre les tâches itératives dans le contexte des applications interactives ?
- Comment coupler des tâches itératives fonctionnant à des fréquences différentes ? Quel est l'impact de ce couplage sur la latence de l'application ?

1.3 Organisation du manuscrit

L'organisation du manuscrit reprend les questions précédentes. La partie I présente un état de l'art autour des applications interactives. Nous ferons une liste des questions ouvertes sur le développement d'applications interactives. La partie II présente notre contribution sur la définition d'un langage de composants hiérarchiques pour la description des applications interactives. Cette partie propose donc une réponse à la première question. Enfin la partie III rassemble des résultats de l'impact de l'échantillonnage sur la latence de l'application. Nous proposons notamment dans cette partie une caractérisation de la latence générée par un échantillonneur simple.

1.4 Travail réalisé

Ma thèse a été réalisée au sein de l'équipe MOAIS de l'INRIA à Grenoble de 2006 à 2009. Au sein de cette équipe, j'ai été encadré par Bruno Raffin. J'ai étudié l'application des techniques d'algorithmique parallèle pour les applications de réalité virtuelle. Cela m'a poussé à participer très activement au développement du logiciel libre FlowVR. L'objectif de FlowVR consiste à fournir les outils nécessaires pour le développement d'applications interactives sur les architectures parallèles. La première version publiée de FlowVR date de décembre 2003. Ce logiciel a été initialement le support du travail de thèse de Jérémie Allard jusqu'en 2005. Depuis 2003, les composants de la suite FlowVR sont maintenus par une dizaine de développeurs qui sont partagés entre l'équipe MOAIS de Grenoble et le laboratoire LIFO d'Orléans.

Aujourd'hui, FlowVR est un projet libre actif. Il propose une nouvelle version tous les 6 mois environ. En excluant les téléchargements depuis le serveur SVN du projet, la version 1.6 de FlowVR a été téléchargée plus de 100 fois en 4 mois environ.

D'après les messages de la mailing-list du projet, FlowVR est même utilisé comme support pour des travaux pratiques dans certaines universités. FlowVR compte plus de 130 000 lignes de codes C++. La majorité des développements liés à ma thèse y ont été insérés. Ces développements ont été faits avec l'objectif d'être à terme distribués à la communauté des développeurs FlowVR. Cela demande donc un effort supplémentaire de génie logiciel pour le test, le packaging ou l'écriture de la documentation.

J'ai aussi contribué au projet Grimage. Grimage est une salle de réalité virtuelle de l'INRIA Rhônes Alpes. Elle comporte un réseau de caméras et un cluster. Les deux projets Grimage et FlowVR sont très liés. Durant ma thèse j'ai contribué à Grimage soit en développant certaines parties de l'application soit en aidant Benjamin Petit et Thomas Dupeux qui sont les contributeurs principaux actuels de l'application. Ce projet m'a permis de mettre en pratique mon travail sur une véritable plateforme expérimentale.

Un des objectifs de Grimage est d'expérimenter les nouvelles architectures disponibles pour les utiliser dans des applications de réalité virtuelle. Cela m'a donc permis d'expérimenter des outils nouveaux. J'ai pu utiliser le calcul sur carte graphique, le réseau Infiniband ou bien les liens réseaux de Grid5000 pour utiliser les cluster de Grenoble, Bordeaux et Orléans au sein d'une même application. Ces technologies ont nécessité à chaque fois un temps d'apprentissage pour les maîtriser.

En dernière année de thèse, j'ai enfin eu l'opportunité de travailler au sein du projet FVNano. Un des partenaires du projet est un laboratoire de bio-chimie théorique. L'objectif est de permettre à un utilisateur d'interagir avec une simulation de dynamique moléculaire. Ce projet met en oeuvre la simulation de centaines de milliers d'atomes sur des dizaines de noeuds de calculs. FVNano est une plateforme expérimentale qui illustre les problématiques de ma thèse. FVNano a permis que FlowVR soit utilisé par des personnes extérieures à notre communauté. Cela a donné lieu à des échanges avec des chercheurs ayant un vocabulaire et une culture différente.

Une partie importante de mon travail a été consacrée à ces projets. Quelquefois ce travail couvrait des problématiques un peu lointaines du sujet original de ma thèse. Mais ces travaux m'ont permis d'acquérir une réelle maîtrise pratique de mon domaine de recherche. Ces projets m'ont aussi permis d'acquérir avant tous un peu de recul. La partie II par exemple a été initiée par le besoin d'outils logiciels pour développer l'application Grimage. Le temps que j'ai passé sur les différentes plateformes d'expérimentation m'a aussi aidé à appréhender et modéliser le problème d'optimisation posé dans mon sujet de thèse. La partie III sur le couplage a pour but d'expliquer et de comprendre certains phénomènes que j'observais durant mes expérimentations.

Ce travail a mené à la production de 3 publications dans des conférences internationales [1, 2, 3], 3 articles de journaux [4, 5, 6] et 2 démonstrateurs présentés lors de conférences internationales [7, 8]. Un article a été soumis en septembre 2009 pour une possible publication lors de la conférence IEEEVR 2010 [9].

Première partie

Etat de l'art

Chapitre 2

Applications interactives

2.1 Introduction

Dans ce chapitre, nous montrons comment sont définies les applications interactives. Nous discutons de la modélisation de ces applications. Nous verrons que ce modèle a évolué pour permettre de décrire des applications de plus en plus complexes. Pour illustrer cette évolution, nous présentons des exemples d'applications de grandes tailles. Ce sont ce type d'applications que nous visons dans cette thèse.

2.2 La boucle d'interaction

Une application interactive est une application qui interagit avec un utilisateur. Cette définition regroupe donc un très grand panel d'applications allant de la réalité virtuelle, jeux vidéos, visualisation scientifique à toutes applications contrôlées par une interface graphique ou textuelle. Une modélisation courante [10] consiste à représenter l'application par une boucle d'interaction (Fig. 2.1). Cette boucle comporte trois étapes :

Entrée : Récupération des évènements d'entrée venant de l'utilisateur.

Calculs : Mise à jour de l'application.

Sortie : Représentation des données à l'utilisateur.

Cette modélisation a influencé de nombreux outils comme `glut` par exemple. `glut` est une librairie qui permet d'interfacer les entrées-sorties du système d'exploitation avec l'exécution de code OpenGL. Dans `glut`, l'utilisateur implémente la boucle d'interaction par l'intermédiaire d'une méthode générique appelée *glutMainLoop* qui se charge de coupler les événements d'entrées (clavier, souris, etc.) avec les calculs et la production de l'image finale.

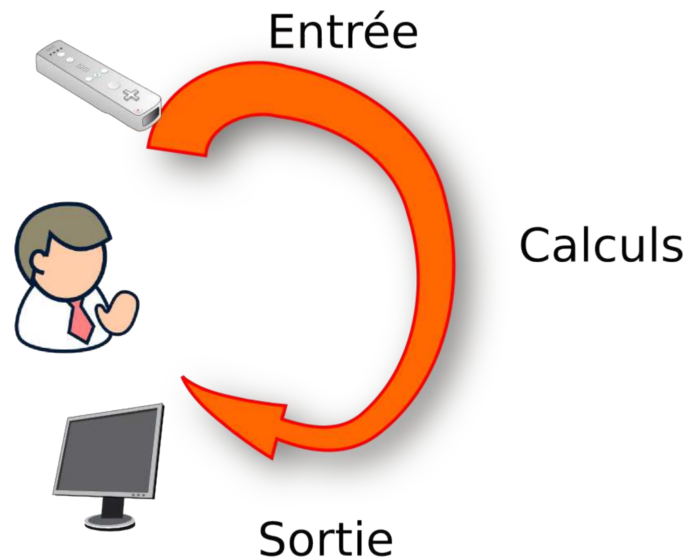


FIGURE 2.1 – La boucle d'itération représentant un pipeline Entrée→Calcul→Sortie

De nombreuses interactions avec la machine peuvent se modéliser facilement avec ce modèle. L'interpréteur de ligne de commandes en est un exemple. L'utilisateur entre une commande. Le système effectue alors un calcul et affiche un résultat à l'écran. La réactivité du système sera l'intervalle de temps entre le moment où l'utilisateur valide sa commande et l'affichage du résultat.

2.3 Limites du modèle

Ce modèle est très simple. Plusieurs évolutions des applications interactives remettent en question la pertinence de cette modélisation. Une application multimodale se modélise difficilement avec ce modèle. La multimodalité est le rendu vers plusieurs sens humains au sein de la même application. Par exemple une application peut utiliser un rendu visuel (par le biais d'un écran), un rendu sonore (par le biais d'hauts-parleurs) ou même un rendu haptique (par le biais de moteurs et du retour de force). Chaque rendu doit être généré par une boucle indépendante. Selon le sens visé, la fréquence de rendu doit être adaptée. Par exemple, traditionnellement, une fréquence de 50Hz pour un rendu visuel est jugé comme acceptable. Pour un rendu haptique, il est nécessaire que la fréquence du rendu soit de plusieurs milliers d'itérations par seconde. Dans ce contexte, il est difficile de concevoir une boucle d'itération unique qui gère différent modes de rendus fonctionnant à des fréquences différentes.

L'exemple de la multimodalité montre que dans le cas d'une application multi-threadée, il est difficile de synchroniser les différents threads au sein d'une boucle unique d'interaction. Au delà de la multimodalité, l'utilisateur devra écrire des applications multithreadées pour profiter d'une architecture parallèle. La synchronisation au sein d'une boucle d'itération unique sera alors un problème critique.

Dans le cadre des applications interactives, les architectures utilisées sont souvent parallèles. Le rendu visuel peut-être fait sur des périphériques parallèles. Par exemple, le rendu stéréoscopique nécessite la production de deux représentations visuelles selon les points de vue des yeux de l'utilisateur. Un casque de réalité virtuelle (*Head Mounted Display*) est un exemple de matériel de rendu stéréo. Pour augmenter la résolution du rendu visuel, il est courant de juxtaposer les périphériques de rendus. La résolution du périphérique sera de l'ordre de la somme des résolutions de chacun des écrans. Il s'agit du principe des murs d'images comme le HIPerWall [11] du Calit à San Diego. Il permet d'afficher une résolution de 25 600 par 8 000 pixels en utilisant 28 cartes graphiques en parallèles. Les *CAVE* [12] sont des cubes dont les faces sont des surfaces de projection. Ils permettent d'obtenir une représentation de la scène virtuelle selon toutes les directions. Ces *CAVE* peuvent parfois être de haute-résolution et stéréo. Le C6 [13] est un *CAVE* à 6 faces dont les arrêtes font 3 mètres de longueur. Il propose une résolution totale de 100 millions de pixels. Il est composé de 24 vidéo-projecteurs et d'un cluster de 48 stations de travail dual-coeurs. Pour ces exemples de rendus parallèles, l'utilisation d'une seul boucle d'interaction rend plus difficile la conception de ces applications.

2.4 Modèle avec des boucles d'interactions multiples

Le modèle autour d'une boucle unique se prête mal à un cadre distribué ou parallèle. Pour étendre le modèle, il est possible de considérer qu'une application interactive est un ensemble de boucles fonctionnant à des fréquences indépendantes et s'échangeant des données (Fig. 2.2). Ce modèle a été appelé *Decoupled Simulation Model* par Shaw et Al. en 1993 [14]. La conception de ce modèle était déjà motivée par la multimodalité et la possibilité d'adapter l'application à une architecture distribuée. Une méthode naturelle pour représenter ce modèle est un graphe de flot de données où les noeuds sont des tâches itératives et les arêtes des flux de données. Cette représentation a été utilisée au départ par les middlewares de visualisation scientifique comme VTK [15], IrisExplorer [16], ou OpenDX [17]. Ces environnements sont appelés MVE (*Modular Visualization Environment*). Grâce à des interfaces graphiques, ils permettent de programmer le pipeline de traitement de données en juxtaposant des composants. Paraview (Fig. 2.3) est un MVE très

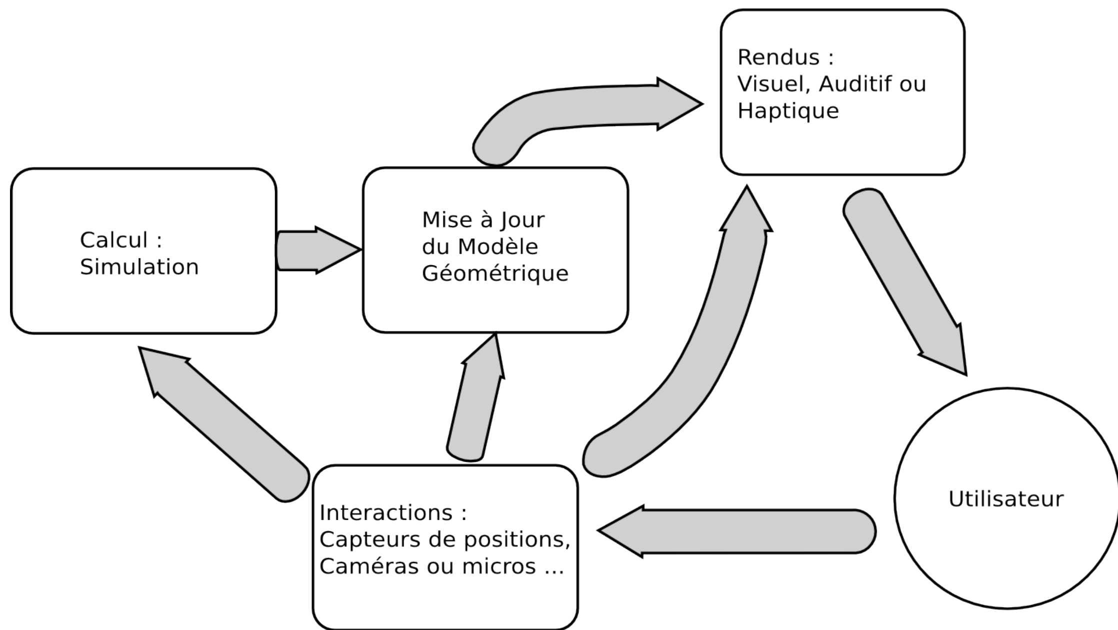


FIGURE 2.2 – Le *Decoupled Simulation Model* proposé par Shaw en 1993. Il montre une première évolution du modèle basée sur la boucle unique d'interaction. Ici, la simulation est séparée de la boucle principale d'interaction et peut fonctionner à une fréquence différente. Ce modèle justifie l'utilisation d'un graphe pour représenter l'architecture d'une application interactive.

populaire pour VTK.

2.5 Exemples d'applications motivant le modèle

Le modèle a évolué pour s'adapter aux nouvelles applications de grande taille. Nous listons ici quelques exemples d'applications interactives de grande taille. Certaines de ces applications comme les mondes virtuels ou la télé-présence sont considérées par les industriels comme des défis scientifiques actuels [18].

Les applications massivement multi-utilisateurs

Ces dernières années, nous avons pu assister au développement important des applications basées sur les interactions sociales. Ces applications rassemblent une large communauté d'utilisateurs et proposent des interactions sociales complexes. Ces applications mettent en interaction des millions d'utilisateurs simultanément et peuvent donc être appelées massivement multi-utilisateurs. *Facebook* (200 millions

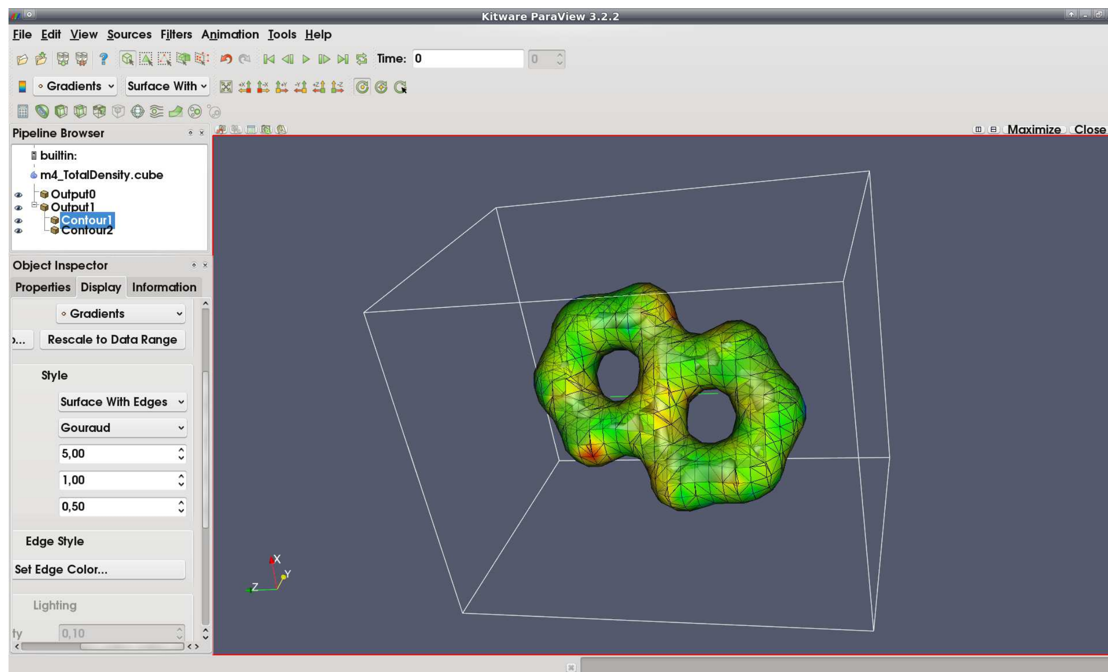


FIGURE 2.3 – Paraview est une interface graphique à VTK permettant de programmer un pipeline de traitement

d'utilisateurs), *Twitter* (55 millions d'utilisateurs) sont des exemples populaires. Le nombre de leurs visiteurs est de l'ordre de la population d'un pays.

Les applications précédentes ont une communauté importante et proposent des interactions sociales. Pour cela, on peut les considérer comme des applications interactives de grande taille. Mais en raison de leurs formats (site internet), elles ne sont pas immersives.

A mi-chemin entre ces applications et la réalité virtuelle, on retrouve les applications proposant des mondes virtuels. Ces applications proposent à un utilisateur de contrôler un *avatar* au sein d'un monde virtuel. L'avatar est une représentation de l'utilisateur. Dans le monde virtuel, l'utilisateur pourra se déplacer et interagir avec des millions d'avatars pilotés par d'autres utilisateurs. Ces mondes virtuels sont parfois appelés *Metavers* en référence au roman *Snow Crash* de Neal Stephenson. *Second Life* (5 millions d'utilisateurs) et *World of Warcraft* (11.5 millions d'utilisateurs) sont les metavers les plus populaires aujourd'hui.

L'immersion dans ces jeux reste cependant limitée si nous la comparons à celle proposée par les applications de réalité virtuelle. Pour permettre à des millions d'utilisateurs d'être réunis au sein du même monde, ces applications utilisent des outils permettant le passage à l'échelle. La difficulté consiste à proposer des mécanisme d'immersion dans ce contexte distribué. Si le développeur utilise des rendus

ou des médias complexes, il sera difficile de garder la cohérence entre tous les utilisateurs. Ces mondes rassemblent une masse importante de données souvent proportionnelle avec le nombre de joueurs. Pour augmenter le réalisme, il est souvent nécessaire d'enrichir d'informations le monde virtuel. Il faut donc avoir la capacité de calcul nécessaire pour traiter l'ensemble de ces données.

Interactions avec des simulations complexes

Certaines applications de simulation scientifiques sont aujourd'hui des codes massivement parallèles fonctionnant sur les machines du TOP 500 [19]. Une nouvelle classe d'application propose d'utiliser cette puissance de calcul pour pouvoir interagir avec des simulations complexes. On peut citer l'exemple d'Hercules. Ce système couple une simulation de séisme avec une visualisation en temps réel. L'application utilise 2000 processeurs et atteint une fréquence de 2Hz sur 12 milliards d'éléments de simulation [20].

Un autre exemple caractéristique est la simulation moléculaire. Pouvoir interagir avec une simulation moléculaire aurait de nombreux avantages. Cela permettrait par exemple de comprendre plus facilement la structure de certaines molécules [21]. Ces simulations doivent s'effectuer en temps réel sur des millions d'atomes. Elles devront appliquer plusieurs fois par seconde, les lois de la physiques sur des millions d'éléments et gérer les interactions avec l'utilisateur. Il est donc nécessaire de proposer des algorithmes qui utilisent efficacement les ressources disponibles pour fournir la puissance de calcul nécessaire.

La télé-présence et la visualisation à distance

La télé-présence propose de nouvelles interactions sociales à distance. La télé-présence propose comme pour les mondes virtuels l'interaction par le biais d'avatars. Contrairement aux exemples précédents, l'avatar doit être ressemblant à son utilisateur pour donner l'impression de présence. L'avatar doit être capable de reproduire le physique, la voix ou les émotions de son utilisateur. Ce type d'application, donnera la possibilité à des personnes éloignées géographiquement de se retrouver dans un monde virtuel pour communiquer entre elles.

Les difficultés sont multiples. Comme pour les applications multi-utilisateurs, il est important de garder la cohérence entre les différents utilisateurs. Cela devient plus difficile si nous multiplions le type de contenu à partager entre les utilisateurs (voix, images, objets 3D) ou le nombre d'utilisateurs.

L'autre difficulté réside dans la construction d'avatars en temps réel. Il existe actuellement de nombreuses méthodes basées sur des algorithmes de vision par ordinateur pour produire ces avatars [22, 23, 24, 25, 26]. Ces méthodes essaient en temps réel de produire un modèle le plus riche possible de l'utilisateur. La

limitation actuelle est la puissance de calcul disponible pour être capable d'enrichir le modèle avec le maximum d'information.

La visualisation à distance est un autre type d'application qui consiste à représenter un jeu de données complexes à distance. Cela peut-être le contenu d'une base de données ou le résultat de simulation complexe. L'OptIPuter [27] est une architecture distribuée entre plusieurs campus américain. L'OptIPuter doit transférer des quantités importantes de données sur des distances importantes. Pour que l'application soit interactive, il est nécessaire de garantir des performances minimum sur les temps de transfert. Là encore, le facteur limitant peut venir aussi des ressources disponibles. Pour développer ces applications, il est nécessaire de proposer des outils utilisant efficacement les ressources disponibles.

2.6 Bilan

Les applications interactives regroupent un large éventail d'applications. Cela part des applications de réalité virtuelle jusqu'à la visualisation scientifique en passant par les jeux vidéos.

Une application interactive peut être modélisée comme un ensemble de tâches itératives. Ces tâches s'échangent des données le long de connexions. Les tâches peuvent fonctionner à des fréquences différentes.

Nous avons présenté des exemples d'applications de grande taille. Le développement de ces applications pose des problèmes scientifiques que nous allons étudier dans la suite du manuscrit.

Chapitre 3

Problématiques autour des applications interactives

3.1 Introduction

Les applications interactives sont modélisées par des boucles s'échangeant des données. Nous proposons maintenant de faire un tour d'horizon de certaines problématiques autour de la conception d'applications interactives.

Nous avons choisi de décrire dans ce chapitre en particulier trois grandes problématiques de recherche qui sont en lien avec le passage à l'échelle et le parallélisme.

3.2 Modèle de flots de données pour l'interactivité

Les tâches itératives formant une application interactive peuvent être représentées sous forme d'un graphe. Les noeuds sont les tâches itératives. Les arêtes représentent les flux de données. Ce graphe de flots de données représente l'architecture de l'application. Il identifie les composants logiciels et les communications entre ces composants.

Pour définir des applications interactives, les middlewares de visualisation scientifiques ont été les premiers à utiliser le modèle de flots de données. Pour la description de ce graphe, ils proposent une interface graphique (appelée MVE) permettant de décrire à la souris le graphe de l'application. Avec ce type d'approche, il est difficile d'automatiser une tâche. Les applications de visualisation scientifique contiennent quelques dizaines de composants au maximum. Dans ce cas, le surcoût lié à l'interface graphique est acceptable. Cependant, avec la multiplication des composants disponibles, des outils ont été développés pour assister l'utilisateur. Par exemple, VisComplete [28] est un outil qui suggère à partir d'une base de données, des schémas classiques de graphe VTK à l'utilisateur.

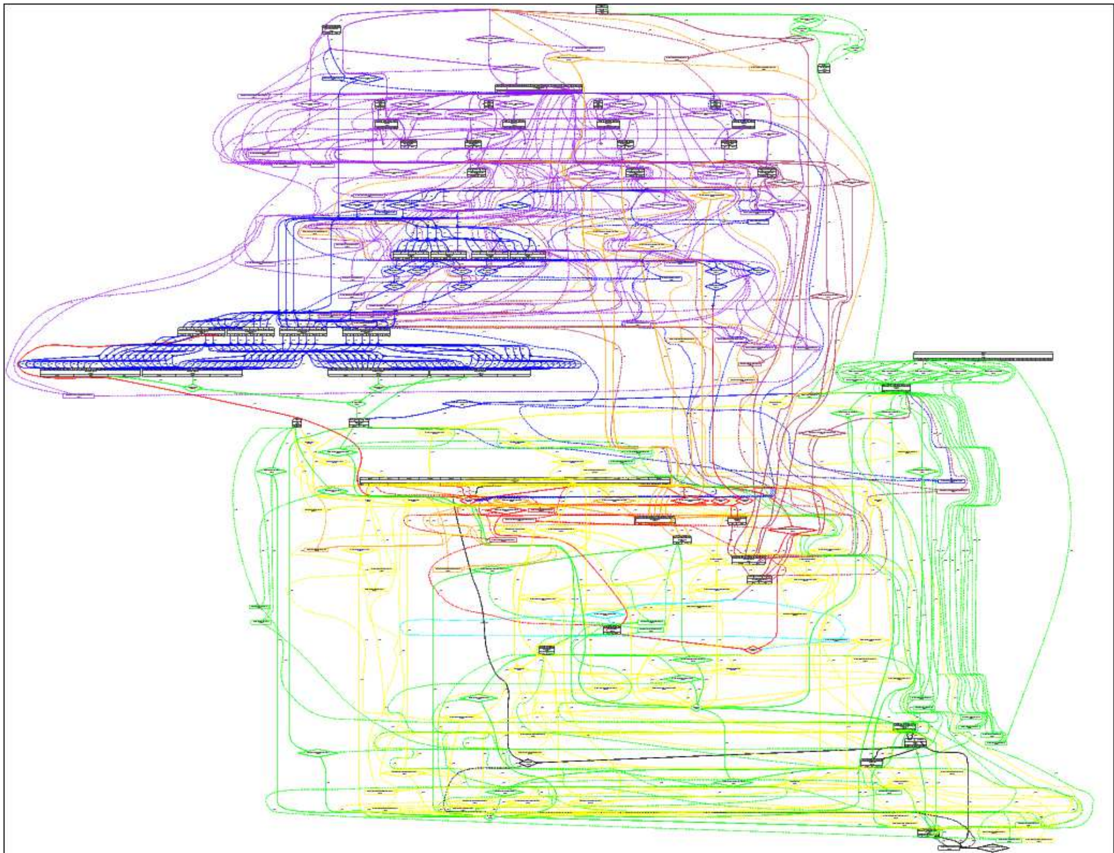


FIGURE 3.1 – Un graphe FlowVR comportant des centaines de tâches et des centaines de communications.

Blue-C [29] ou Grimage [3] sont des exemples d'applications interactives hautes-performances de grande taille. Ces applications comportent plusieurs centaines de composants et connexions (Fig. 3.1). L'approche par MVE n'est plus envisageable avec ce type d'application. Il est nécessaire de trouver un langage de description qui permet de programmer le graphe représentant le couplage des composants.

Les composants sont des codes compilés qui peuvent facilement s'interfacer. Les fonctionnalités et l'interface de chaque composant sont clairement spécifiées. Grâce à cette spécification, il est possible de coupler les composants au sein d'une même application. Le langage qui décrit le couplage de ces composants est appelé *Architecture Description Language* (ADL). CCA (*Common Component Architecture*) ou CCM (*Corba Component Model*) sont des exemples de langages de description d'architecture (ADL) pour des applications distribuées. Les modèles à composants sont des solutions qui permettent d'appréhender la complexité logicielle. Ils permettent de développer des fonctionnalités indépendamment du contexte d'utilisation. Le

modèle repose sur une spécification claire et précise pour chaque composant de la fonctionnalité à développer. Ce principe rend le couplage et la réutilisation de code plus simple. Il est à noter qu'il existe des intergiciels dédiés à la visualisation scientifique ou à la réalité virtuelle déjà basés sur ces modèles. Par exemple SCIRun est un environnement de visualisation scientifique développé à partir de CCA [30]. VRJuggler [31] propose un support des composant CORBA.

La séparation entre la description du couplage des composants (l'ADL) et l'implémentation des tâches permet de favoriser la modularité de l'application. Cela facilite la réutilisation du code et le travail en équipe. Le problème du choix de l'ADL est un problème important de génie logiciel. On peut par exemple citer l'exemple de Fractal [32]. Fractal est une norme proposant un langage de description basé sur des composants hiérarchiques. Cette hiérarchie permet de séparer les niveaux d'implémentation. Ces normes de conception sont ensuite implémentées pour être adaptées à un contexte d'utilisation précis. ProActive [33] est une implémentation de Fractal dédiée aux applications s'exécutant sur les grilles de calcul.

Dans le cadre de la réalité virtuelle, de nombreuses recherches étudient spécifiquement le problème de la description des applications interactives. InTML [34] est un ADL en XML dédié aux applications de réalité virtuelle. [35] ou [36] sont d'autres exemples d'ADL pour la réalité virtuelle.

Les composants proposent une méthodologie qui simplifie le couplage d'objets logiciels. Certains problèmes d'ingénierie sont partagées par de nombreuses applications. Les développeurs se posent souvent les mêmes questions lors de l'élaboration de l'architecture de leurs applications. Pour guider les développeurs, l'expérience des développeurs sur ces questions récurrentes a été compilée. Cette compilation propose de définir des patrons de conception [37] (*design pattern*) répondant à des problèmes d'ingénierie classiques. La connaissance et l'étude de ces patrons donnent une méthode systématique pour résoudre ces questions. Ces patrons donnent aux développeurs une boite à outils robuste pour développer leurs applications.

Les patrons de conception sont donc plus une méthodologie qu'un outil logiciel. Avec le même objectif de structurer les applications, la programmation par squelette [38, 39] se propose d'encapsuler les schémas récurrents dans des structures logicielles. Les librairies de squelettes fournissent un langage haut niveau permettant de décrire une application. L'implémentation des squelettes est cachée au sein de la libraire. L'implémentation des schémas peut-être adaptée au contexte d'exécution par une phase de compilation. Un modèle de coût peut être aussi associé aux squelettes. Les squelettes poussent à structurer l'application. Ils ne contiennent pas les détails de l'implémentation. Ces avantages rendent la maintenance de ces applications plus simple. Les applications programmées avec les squelettes sont

performantes car adaptées au contexte d'exécution. La programmation par squelette peut-être utilisée pour réaliser du calcul haute-performance. Par exemple, ASSIST [40] est une libraire de squelettes pour le calcul sur grille.

Il est difficile de construire une librairie exhaustive de squelettes. Pour cette raison, certaines librairies de squelettes se spécialisent volontairement sur un domaine particulier. Cela permet de proposer un ensemble restreint mais robuste de squelettes. Par exemple, Skipper [41] est dédiée aux applications de vision par ordinateur. Une étude précise des applications visées par la librairie permet d'identifier les squelettes pertinents. Le travail consiste ensuite à produire des implémentations performantes de ces schémas.

3.3 Synchronisation entre boucles

Au sein d'une application interactive des boucles ayant des fréquences différentes peuvent cohabiter. Les applications multimodales font communiquer des boucles dont les écarts de fréquences sont importants. Le problème réside dans l'échange de données entre ces boucles. On appelle échantillonage le mécanisme qui permet d'adapter la fréquence d'un flux de message à la fréquence d'une boucle.

Les middlewares de réalité virtuelle comme FlowVR [42], OpenMask [43], COVISE [44] ou VISTA [45] proposent des échantillonneurs dans leurs librairies de composants. OpenMask insiste particulièrement dans son modèle sur la notion d'échantillonneur en introduisant les *polateurs*. Les polateurs effectuent l'échantillonnage. Un polateur est associé à chaque port de sortie d'un composant OpenMask. Une méthode d'échantillonnage simple mais pouvant être utilisée sur tous types de flux est proposée par défaut. L'utilisateur peut redéfinir la méthode d'échantillonage pour l'adapter à son propre flux de données.

La méthode d'échantillonage qui consiste à donner toujours dans un flux le message le plus récent peut être utilisée de façon générale. Par contre, cette méthode peut provoquer des incohérences dans le rendu final. Par exemple, le couplage d'un rendu volumique lent avec un rendu d'objets rapides [46] provoque des incohérences (ou *artefacts*). Par exemple, si l'utilisateur modifie la position de la source lumineuse de la scène virtuelle, les ombrages du rendu rapide seront mis à jour en premier. Avant la mise à jour des ombrages sur le rendu lent, les deux objets seront éclairés selon des sources lumineuses différentes. L'utilisateur détectera cette incohérence entre les ombrages.

Ces incohérences peuvent dans certains cas rendre désagréable l'utilisation d'un système de réalité virtuelle. Une application utilisant un casque de réalité virtuelle, si des incohérences apparaissent entre le capteur de position de la tête et la méthode de rendu dans le casque, l'utilisateur peut ressentir des vertiges et le "mal de mer". Pour réduire cet effet, l'échantillonneur va tenter d'extrapoler la trajectoire de la

tête de l'utilisateur donnée par le capteur. Ces méthode d'extrapolation peuvent utiliser des mécanismes de prédiction [47].

Les méthodes d'extrapolation dépendent souvent du type de donnée à échantillonner. La recherche en imagerie propose de nombreuses solutions pour coupler des rendus de vitesses différentes [48, 49, 50]. Une application avec un rendu haptique doit envoyer des commandes aux périphériques de rendu (un phantom par exemple) à plusieurs kHz. Le système a alors moins d'1 milliseconde pour calculer les forces à appliquer sur le périphérique haptique. Il n'est donc pas possible de calculer dans ce laps de temps la mécanique de la scène entière. On utilise plutôt une simulation physique basée sur un modèle très simple que l'on applique localement autour du pointeur haptique. Cette simulation extrapole alors à haute fréquence les forces autour du pointeur haptique à partir des résultats générés par une simulation globale de la scène. La simulation globale utilise un modèle mécanique plus réaliste et complexe et fonctionne à fréquence plus basse.

Les incohérences ont un impact sur l'utilisation des systèmes interactifs [51]. Cependant il est difficile de mesurer cet impact. Si on reprend les exemples précédents, l'artefact sur un rendu visuel pourra gêner l'utilisateur mais ne sera pas critique. Par contre, l'incohérence au niveau des résultats du capteur de positions rendra l'utilisateur d'un casque de réalité virtuelle impossible. La méthode actuelle pour mesurer cet impact est de réaliser des études utilisateurs. La nécessité d'avoir un observateur humain pour mesurer l'impact rend difficile l'étude formelle des mécanismes d'échantillonnage et de cohérence dans l'application. En effet, il est difficile de formaliser une métrique représentant le ressentis d'un utilisateur.

Le problème de l'échantillonnage est central dans les applications interactives. La plupart des middlewares ont intégré les outils nécessaires pour proposer des méthodes d'échantillonnage. Cependant, les solutions proposées sont souvent dépendantes de l'application. Il est difficile de proposer des méthodes génériques permettant de résoudre le problème d'échantillonnage.

3.4 Utilisation du parallélisme

Pour les applications de visualisation scientifique, Ahrens [52] a listé les différentes formes possibles de parallélisme. Cette liste peut facilement se généraliser à l'ensemble des applications interactives.

Parallélisme de pipeline

Une application est une liste de filtres à appliquer sur les données. Elle consiste à partager les étapes du traitement sur plusieurs unités de calcul. Cette forme

de parallélisme permet d'augmenter le débit des résultats. Par contre, la latence globale de l'application ne diminue pas.

La modélisation en DAG permet d'exprimer ce parallélisme. En effet, il est possible d'associer un thread de calcul à chaque tâche du DAG.

Il existe de nombreuses politiques pour diffuser les données d'un étage à l'autre. On peut diviser en deux catégories de politique : poussé par la demande (*demand-driven*) ou poussé par les évènements (*event-driven*). Avec la politique *demand-driven*, les calculs sont lancés par la demande d'une valeur à la sortie du pipeline. La politique *event-driven* permet au composant en amont du pipeline de contrôler le débit.

La programmation par flux (ou *streaming programming*) est basée sur ce type de parallélisme. Les langages comme StreamIT [53] (Fig. 3.2) ou Brook [54] permettent de déclarer une liste de filtres à utiliser en pipeline. Sur des architectures assez contraintes comme les cartes graphiques, de nouveaux langages [55] permettent d'exploiter le parallélisme de pipeline.

Parallélisme de données

De nombreux algorithmes appliquent des traitements indépendants sur un ensemble de données. Par exemple, en traitement d'image, il est courant d'itérer sur les pixels pour appliquer une fonction sur le pixel et ses voisins. Une méthode de parallélisme consiste à partager l'espace de l'image entre plusieurs threads qui appliqueront séparément la fonction.

Ce parallélisme s'adapte bien aux architectures SIMD (*Single Instruction Multiple Data*). Ce genre d'unité de calcul applique simultanément la même instruction sur un tableau de données. Les cartes graphiques sont des exemples d'architecture SIMD. Dans OpenGL, les *shaders* sont des programmes qui représentent les instructions à appliquer à l'ensemble des données. Des langages comme Cg [56] ou GLSL [57] permettent de les programmer. Le *vertex shader* sont les instructions à appliquer à tous les sommets d'un objet d'une scène virtuelle. Le *pixel shader* pour tout les pixels de l'image. Pour utiliser les cartes graphiques pour des calculs plus généraux, des nouveaux langages comme CUDA [58] ou OpenCL [59] sont apparus. Le principe est toujours de définir un noyau (*kernel* en anglais) qui représente les instructions à appliquer sur les objets en entrée.

En visualisation scientifique, l'enjeu est de visualiser de très gros jeux de données. Le rapport entre le nombre de coeurs disponibles pour le traitement et la taille des données est petit. Dans ce cadre, le parallélisme de donnée est suffisant pour partager efficacement le travail sur les architectures parallèles.

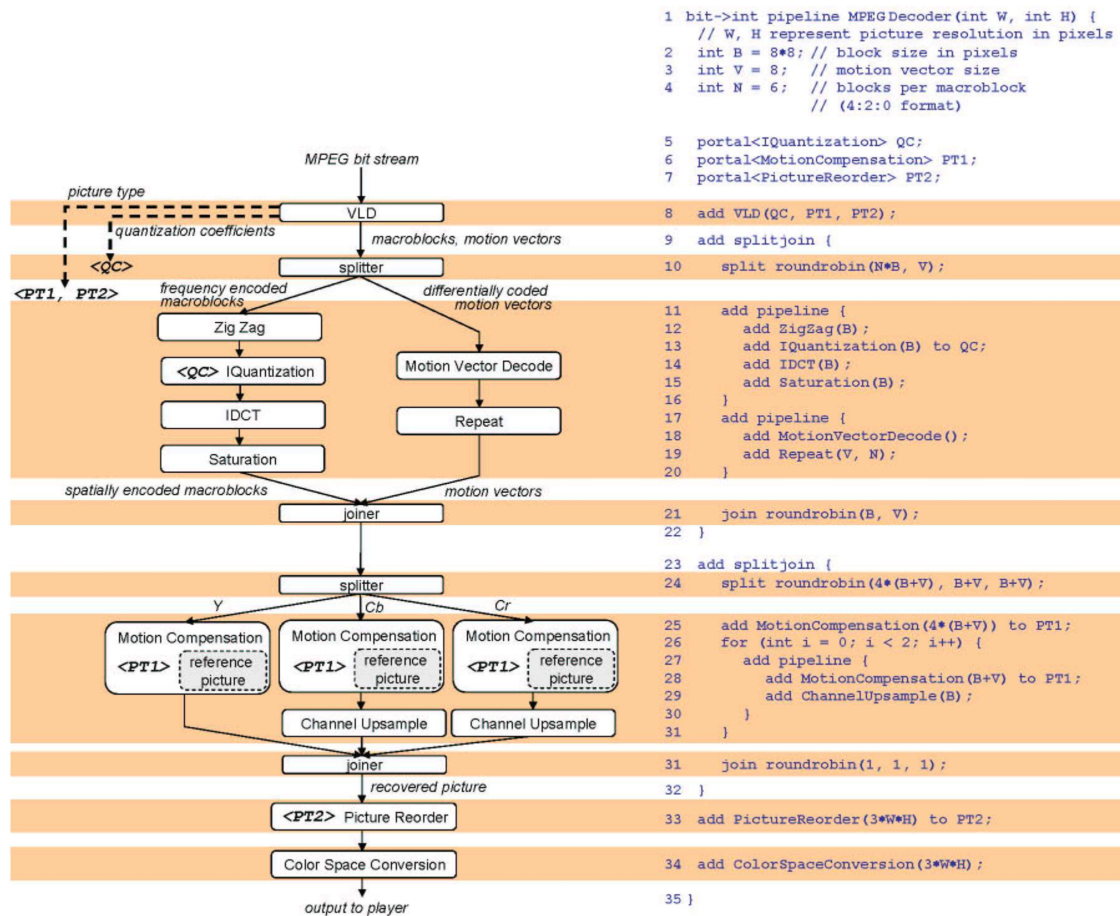


FIGURE 3.2 – Description avec *streamIt* d'un décodeur de flux MPEG.

Parallélisme de tâches

Le parallélisme de tâches est l'exécution simultanée de tâches qui ne sont pas en précédences dans le DAG représentant l'application.

Pour certains algorithmes comme le rendu, la communauté de visualisation scientifique a développé des algorithmes parallèles spécifiques. En effet, pour certaines applications, il est possible d'utiliser des hypothèses supplémentaires pour accélérer le rendu. Certains algorithmes permettent d'afficher de façon interactive des milliers d'atomes ou des rendus volumiques complexes [60]. Pour développer ces algorithmes, le support du parallélisme des middlewares est souvent basique. Les développeurs se tournent vers les langages parallèles traditionnels [61] qui permettent d'utiliser toutes les formes de parallélisme.

Remarque

Dans les applications interactives, les tâches itératives peuvent être parallèles. Par conséquent ces trois formes de parallélisme peuvent se retrouver soit au niveau de l'application entière soit au niveau des tâches.

3.4.1 Partage des ressources entre les différentes formes de parallélisme

Il existe peu d'outils proposant une gestion forte de toutes les formes de parallélisme. En pratique, l'utilisateur doit gérer manuellement le partage des ressources. Si nous prennons l'exemple de Grimage [62], FlowVR gère le parallélisme de données et de pipeline dans l'application. Grimage utilise l'algorithme de reconstruction géométrique EPVH [63]. Cet algorithme permet à partir du flux de plusieurs caméras synchronisées entre elles, de construire le maillage de l'utilisateur. Cet algorithme a été parallélisé séparément [64]. Il n'y a pas de mécanisme automatique dans Grimage pour le choix du nombre de processeurs affectés à la reconstruction. C'est le développeur qui choisit le nombre adéquat selon le contexte d'utilisation. Dans le cas de VTK, certains composants de traitements d'image sont multithreadés, là aussi, le contrôle du nombre de threads à affecter à chaque filtre doit se faire à la main. En général, le problème du placement des tâches est un problème qui n'est pas traité dans la plupart des middlewares.

En pratique, dans le cadre de la réalité virtuelle, le placement est assez contraint. Les plateformes sont hétérogènes, ce qui impose des contraintes de placement sur de nombreuses tâches. Par exemple, les tâches traitant les résultats des capteurs doivent être placées sur les machines où les capteurs sont connectés. Le rendu graphique doit se réaliser sur une machine ayant les cartes graphiques nécessaires.

Mais à terme, avec le développement des nouvelles architectures comme les machines SMP ou les hiérarchies de mémoire, une grande expertise sera nécessaire pour proposer un placement efficace des tâches. L'utilisateur aura besoin d'outils qui l'aideront à réaliser ce placement.

Le placement des tâches est actuellement réalisé manuellement. Cela rend aussi les applications moins portables. En effet, pour réaliser un placement, l'utilisateur doit adapter l'application à l'architecture cible. Si nous reprennons le cas de Grimage, le nombre de caméras et de pipelines de traitements d'images dépendra de l'architecture. Ce problème se retrouve dans la programmation parallèle. Il est souvent nécessaire de *tuner* (adapter) les algorithmes pour obtenir de bonnes performances. Certains middlewares pour le calcul haute performance implémentent une stratégie *best-effort*. *Best-effort* peut se traduire par "fait au mieux". L'utilisateur grâce à des mots clés définit les tâches pouvant être exécutées en parallèle. Le système calcule la granularité (le nombre de tâches à créer) et l'ordonnancement des tâches selon les ressources disponibles pour l'exécution de l'application. Ce mécanisme peut être adaptatif et dynamique en utilisant par exemple des techniques comme le vol de tâches [65]. Cilk [66], KAAPI [67] ou TBB d'Intel [68] sont des exemples de middlewares fournissant des mots clés au langage C++ permettant de déclarer des tâches et leurs dépendances. Plus largement, il existe une communauté qui étudie en particulier les algorithmes écrits indépendamment du nombre de processeurs disponibles (*processor-oblivious*). Grâce à ces algorithmes, le développeur écrit des applications pour une architecture parallèle mais laisse au système le soin de choisir la granularité de l'application.

Certains algorithmes classiques ont été implémentés en utilisant le vol de tâches. On peut citer des implémentations de la STL [69] ou des parcours d'octree [70]. Des méthodes adaptatives se rapprochant du vol de tâches sont utilisées pour gérer le problème de l'allocation de ressources [71]. Par contre, dans le cadre des applications interactives, il n'existe pas encore de résultats utilisant des techniques adaptatives permettant de résoudre le problème du partage de ressources.

3.4.2 Couplage de codes parallèles

Le couplage des tâches parallèles est un problème important. Un schéma de communication naïf entre les tâches va entrainer des goulots d'étranglement qui pénaliseront les performances de l'application. Prenons deux tâches parallèles s'échangeant un flux de données. Le producteur de données utilise N processeurs. Le consommateur fonctionne sur M processeurs. Le problème est d'envoyer efficacement les données des N producteurs aux M consommateurs. La solution naïve qui consisterait à réunir sur un noeud les données des N producteurs pour les diffuser aux M consommateurs ne passe pas à l'échelle. Le schéma reliant N producteurs à M consommateurs s'appelle une communication NxM.

Dans le cadre de la visualisation scientifique, il existe des composants traitant ce problème [72, 73, 74]. Ces composants ne prennent pas en compte l'échantillonnage. Pour cette raison, sur une application de réalité virtuelle, il est difficile d'utiliser directement ces composants. Pour coupler efficacement des codes parallèles dans les applications interactives, il serait nécessaire d'étudier en détail les communications NxM avec échantillonnage.

3.5 Bilan

Nous avons énuméré différentes problématiques autour des applications interactives. Pour permettre le passage à l'échelle, nous avons besoin d'outils permettant la description d'applications. Cette problématique est partagée par de nombreuses communautés. La communauté autour de la réalité virtuelle cherche à adapter les contributions existantes sur cette problématique. Par exemple, le groupe de travail SEARIS [1] cherche sur l'ensemble de la communauté de la réalité virtuelle à rassembler ces initiatives. Pour notre part, les contributions qui seront présentées dans la partie II essaient de répondre à cette problématique.

Les autres problématiques que nous avons identifiées sont directement liées au problème de performance. L'utilisation du parallélisme est un enjeu important pour obtenir la puissance de calcul nécessaire à l'élaboration de nouvelles formes d'interaction. Là aussi, l'enjeu est d'adapter les problématiques traditionnelles de la communauté du parallélisme aux applications interactives car le contexte est particulier. Il est possible de détruire ou de reproduire des données, les architectures sont hétérogènes et distribuées. Nous avons énuméré au fil du chapitre des exemples d'applications parallèles [62, 29] démontrant que le parallélisme permet d'expérimenter de nouvelles formes d'interactions. La communauté cherche maintenant à développer les outils génériques et robustes qui permettront d'utiliser le parallélisme efficacement.

1. http://www.searis.net

Chapitre 4

Optimisation des performances des applications interactives

4.1 Introduction

Ce chapitre présente un état de l'art sur le problème d'ordonnancement pour les applications intéractives. Nous avons vu dans les chapitres précédents que les applications sont définies par un graphe où les noeuds représentent les tâches et les arêtes les communications entre tâches. Cette modélisation permet de définir des métriques et des problèmes d'optimisations. Ce sont ces problèmes que nous définissons dans ce chapitre.

Les algorithmes d'ordonnancement sont au centre de la programmation parallèle. Ce sont des problèmes très largement étudiés. Nous verrons dans ce chapitre, qu'il est nécessaire d'adapter ces résultats à notre contexte.

4.2 Métriques pour les applications interactives

Dans le cadre de la boucle unique, les métriques à optimiser seront la fréquence et la latence. La fréquence est le nombre d'itérations par seconde de la boucle. La latence serait le temps d'une itération de la boucle. Dans le cadre des boucles multiples, ces métriques sont plus complexes. Tout d'abord, chaque boucle a sa propre fréquence. Il n'est plus possible de définir une fréquence globale de l'application. La latence n'est plus unique. Il est possible d'associer une latence à chaque couple source-puit dans le DAG.

En prenant en compte l'échantillonnage (cf paragraphe 3.3), nous introduisons de nouvelles métriques. Tout d'abord, il est possible de prendre en compte le taux d'incohérence. Cependant comme nous l'avons présenté, cette métrique est complexe à formaliser. Si nous prennons l'exemple du rendu multi-fréquence [46],

l'incohérence peut être la différence de latence entre le chemin passant par la tâche rendu rapide et la tâche rendu lent. Dans le cas, du casque de réalité virtuelle, l'incohérence sera la différence de la latence entre le capteur de position et le rendu dans le casque comparée avec le temps de réaction de l'oreille interne de l'utilisateur qui donnera aussi la position de la tête. Dans ce cas, il est nécessaire d'intégrer l'humain au modèle et de comprendre le fonctionnement de son corps pour quantifier l'incohérence.

En pratique, mesurer certaines métriques demande quelques fois beaucoup d'efforts. Steed propose une mesure de la latence d'une caméra reliée à un système de rendu, en utilisant un oscillateur mécanique [75]. Intégrer ces méthodes dans un middleware pour obtenir certaines constantes du système demanderait une phase d'étalonnage. Dans le cadre d'un système hétérogène ou distribuée, il est difficilement envisageable de mesurer ces valeurs.

Il est possible de considérer des métriques liées à l'utilisation du système. Cela peut-être par exemple le taux d'utilisation des machines. Dans ce cas, on cherchera à utiliser au maximum et le plus efficacement la machine cible. L'échantillonnage apporte une métrique sur l'efficacité de l'utilisation d'une machine. En effet, comme les boucles fonctionnent à des fréquences différentes, il est possible que les boucles proches des sources dans le DAG aillent beaucoup plus vite que les boucles en fin de graphe. Dans ce cas, les échantillonneurs jetteront la plupart des messages produits. On peut donc mesurer le taux de perte de message par les échantillonneurs. Si ce taux de perte est élevée, cela impliquera que des tâches ont réalisé des itérations inutiles. Ces tâches ont donc consommé des ressources pour calculer des messages qui seront à terme jetés. Ce taux de perte est souvent mesuré par les protocoles de régulation ou de qualité de service (QoS). RTCP [76] est par exemple un protocole de régulation qui réduit le taux de perte lors de la diffusion de vidéo.

4.3 Problème d'optimisation

4.3.1 Satisfaction de contraintes

A partir des métriques, nous pouvons lister des problèmes d'optimisation. Une première possibilité est de considérer qu'il existe des bornes à respecter pour que l'application soit réactive. Cela peut être par exemple qu'un rendu haptique doit fonctionner à plusieurs kHz, que la latence soit inférieure à une borne ou que la gigue (la variation de la réactivité) soit inférieur à un taux. Les questions seront alors :

- Existe t'il un ordonnancement respectant les contraintes d'interactivité ?
- Si oui, lequel ?
- Si non, que dois-je modifier dans mon application ou l'architecture que j'uti-

lise pour atteindre les contraintes ?
Un problème similaire est étudié dans la communauté temps réel. Il existe une modélisation des tâches itérative qui définit les problèmes de *Cyclic Scheduling* (Ordonnancement cyclique) [77].

Le temps-réel garantit pour des systèmes critiques que toutes les contraintes soient respectées. Dans le cadre des applications interactives, les utilisateurs peuvent se satisfaire d'une garantie plus faible. En effet, si occasionnellement, des contraintes ne sont pas respectées, l'utilisateur pourra ne pas s'en rendre compte. Les algorithmes temps-réel sont basées sur des mesures de performance des tâches. Les ordonnancements sont calculés avant l'exécution de l'application à partir de ces mesures. Les applications interactives utilisent des algorithmes dont la complexité varie lors de l'exécution de l'application. En rendu graphique, la complexité de nombreux algorithmes dépend du nombre d'objets dans la scène par exemple. Les performances de ces tâches dépendent du contexte d'exécution et ne peuvent pas être prévues à l'avance. Ces approches s'adaptent difficilement à notre contexte.

Comme nous l'avons rappelé précédemment sur les métriques, la définition de l'interactivité est complexe. Si certaines métriques sont difficiles à mesurer ou à formaliser, les bornes à atteindre sont toutes aussi complexes à définir. Contrairement à un système mécanique comme un véhicule, où il est possible grâce aux lois de la physique de mesurer le temps de réactivité maximum acceptable d'un système de freinage par exemple. Il est beaucoup plus difficile d'appliquer la même méthodologie pour des comportements humains. Pour cette raison, la communauté de la réalité virtuelle définit statistiquement l'interactivité d'une application par des études utilisateurs. Sur un groupe d'utilisateurs, on cherche à savoir si une tâche donnée est réalisable en utilisant un système interactif. Le résultat de l'étude statistique définit le niveaux d'interactivité du système. [78] est un exemple d'étude statistique analysant l'interêt d'utiliser un retour haptique dans une application.

4.3.2 Problème d'optimisation

Une autre vision du problème consiste à faire au mieux avec l'architecture disponible. Le problème devient alors un problème d'optimisation. Nous nous plaçons dans un contexte plus simple sans échantillonnage. Dans ce cas, toutes les tâches fonctionnent à la même fréquence. Certaines métriques peuvent être étudiées. Cela peut être la latence maximale (qui correspond au C_{max}, le chemin le plus long dans le graphe), la fréquence des composants ou d'autres métriques comme la fiabilité du système. Sur des applications représentées uniquement par une chaîne de tâches, le problème de la recherche de l'ordonnancement optimal sur une plateforme hétérogène est dur [79]. Si on considère plusieurs métriques, le problème devient souvent multi-critère [80]. Cela nécessite de définir un compromis entre les métriques pour décider de l'ordonnancement optimale.

Pour illustrer ces résultats, prenons un exemple simple. Considérons l'application représentée par la figure 4.1. Nous nous plaçons dans un cadre simple où les machines sont identiques. Nous ne prenons pas en compte les communications. μ^i représente le coût de la tâche i et $v = 1$ est la vitesse des machines P1 et P2. L'application a trois tâches itératives (M1,M2,M3). Il y a 3 placements possibles. Nous pouvons pour chaque placement calculer la latence et la fréquence. En énumérant toutes les solutions possibles, il n'est pas possible de trouver un placement qui optimise la fréquence et la latence en même temps.

Sur des instances non-triviales, ces problèmes d'ordonnancement sont au minimum NP-dur [81]. Nous ne connaissons pas d'algorithme en temps polynomial donnant la solution optimale. Pour une application interactive comportant des centaines de tâches dont les périodes d'itération sont de l'ordre de la dizaine de millisecondes, les choix de l'ordonnanceur doivent se faire rapidement. Il n'est donc pas possible de calculer l'ordonnancement exact optimal. Une approche réalisable consiste à adopter un algorithme qui donnera en temps polynomial une solution approchée dont nous pourrons idéalement estimer l'écart avec la solution optimale. L'exemple le plus connu est l'ordonnanceur par liste (*list-scheduling*) où un algorithme glouton permet d'ordonnancer des tâches avec dépendances à un facteur 2 de la solution optimale [82]. Ces approches peuvent mener à des algorithmes distribués efficaces.

Il existe pour les applications interactives des approches tentant de résoudre le problème d'optimisation [83]. Sur des applications de grande taille, la modélisation du système pose des problèmes techniques. Mais, il est possible d'appliquer ces résultats sur des applications plus petites comme un encodeur MPEG [84]. Certains langages dédiés aux systèmes embarqués comme StreamIT [53] utilisent ces résultats pour implémenter des ordonnanceurs efficaces.

4.4 Bilan

Le problème d'ordonnancement pour les applications interactives est complexe. Nous avons identifié plusieurs points difficiles à résoudre. Tout d'abord, l'interactivité est définie par rapport aux ressentis de l'utilisateur. La traduction de la notion d'interactivité en métrique demande de comprendre comment l'homme perçoit le monde. Cette question est difficile. Une première mesure expérimentale de l'interactivité consiste à réaliser des études statistiques de l'utilisation des systèmes. La compilation de ces études pour formaliser des métriques est à elle seule un domaine de recherche.

Nous pouvons étudier d'autres métriques qui permettent de décrire l'usage des ressources. Ces métriques sont objectives et ne dépendent pas d'un utilisateur humain. Nous pouvons donc les formaliser à partir d'une modélisation du sys-

tème. Nous pouvons les mesurer de façon automatique. Ces métriques permettent donc de poser, de réfléchir et d'expérimenter des solutions aux problèmes d'ordonnancement des applications interactives. En utilisant efficacement les ressources, l'application aura de meilleures performances. Ces optimisations auront un effet bénéfique sur l'interactivité du système.

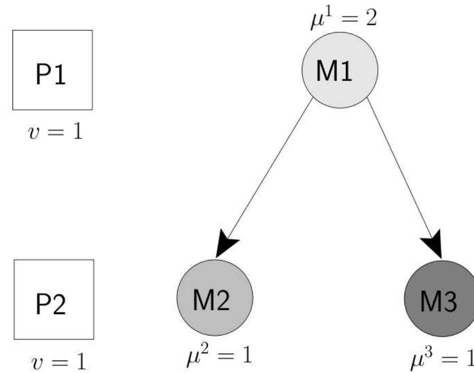
Les algorithmes d'optimisation hors-ligne s'adaptent mal au contexte des applications interactives. Le coût de calcul des tâches ou la quantité d'informations transitant entre les tâches varient lors de l'exécution de l'application. Nous devons donc priviligier des algorithmes en-ligne qui sauront s'adapter à ces modifications.

Les résultats connus de complexité sur l'ordonnancement cyclique ou l'ordonnancement de pipeline montrent que ces problèmes sont complexes. Nous pouvons penser qu'en prennant en compte l'échantillonnage, c'est à dire la perte ou la duplication de données, le problème de l'ordonnancement d'application interactive sera au moins aussi complexe. L'échelle de temps dans les applications intéractives est de l'ordre de la milliseconde. La complexité d'un algorithme d'ordonnancement devra au maximum être polynomiale par rapport à la taille de l'application. Ces résultats de complexité nous indiquent qu'il n'est pas envisageable de trouver l'ordonnancement optimal. Nous devons donc chercher des algorithmes approchés.

La démarche que nous avons choisie pour l'étude de la partie III a été choisie à partir de ces observations. Nous avons priviligié des métriques objectives comme des mesures de temps ou l'utilisation des ressources. Dans cette partie, nous étudierons essentiellement les algorithmes distribuées et en ligne.

Tâches	Transitions	Processeurs
M1 : $\mu^1 = 2$	**M1** vers **M2**	**P1** : $v_1 = v = 1$
M2 : $\mu^2 = 1$	**M2** vers **M3**	**P2** : $v_2 = v = 1$
M3 : $\mu^3 = 1$		

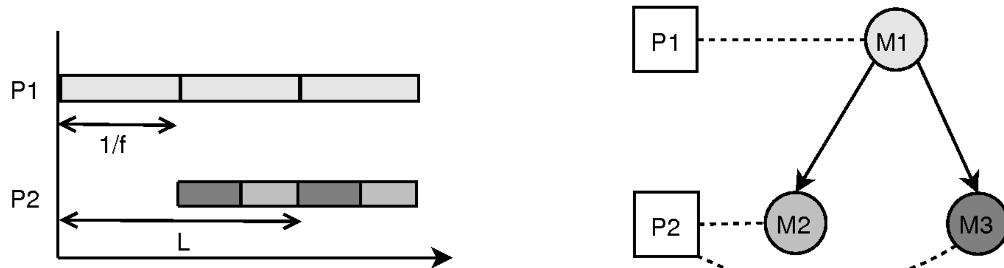
(a) Description de l'application



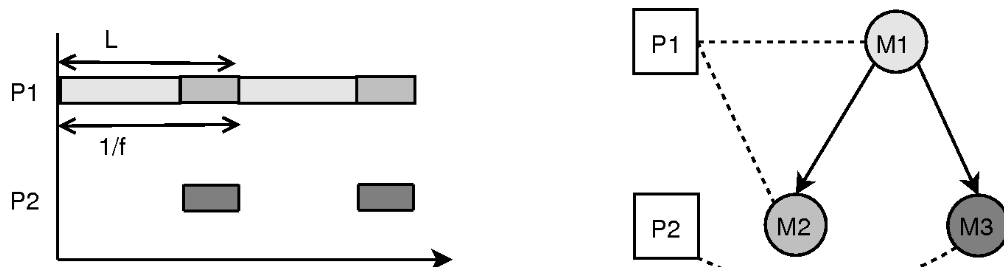
(b) DAG représentant l'application

Placement des tâches	{M1, M2, M3}{}	{M1, M2}{M3} (figure 4.1e)	{M1}{M2, M3} (figure 4.1d)
1/f (unité de temps)	4	3	2
L (unité de temps)	4	3	4

(c) Enumération de tous les placements possibles



(d) Chronogramme et placement optimisant la fréquence



(e) Chronogramme et placement optimisant la latence

FIGURE 4.1 – Application ne pouvant maximiser la latence et la fréquence en même temps

Deuxième partie

Modèle de composants hiérarchiques

Ce chapitre décrit le travail réalisé autour de la définition d'un modèle à composants hiérarchiques pour la description d'applications interactives de grande taille. Les résultats présentés dans cette partie ont été publiés dans deux articles. L'article [1] a reçu un prix d'*Excellent Student Paper Award* à la conférence NPC 07 en Chine. Le second [4] est une version étendue de l'article et a été publié dans le journal *Supercomputing* en juillet 2008. L'implémentation de ces résultats a été intégrée à l'intergiciel libre FlowVR [1]. La version 1.6 de FlowVR comporte une version stable de cette implémentation. Ce modèle est actuellement utilisé dans plusieurs applications FlowVR. Enfin, un article de synthèse sur FlowVR reprenant entre autres cette partie a été accepté cette année pour une édition spéciale du journal Presence [6].

---

1. http://flowvr.sourceforge.net/

Chapitre 5

Introduction

Dans cette partie, nous étudions en particulier le problème de description de l'application. Le prochain chapitre présente en détail nos motivations et la contribution. Ce chapitre introductif est une présentation de FlowVR. FlowVR est un middleware pour le développement d'applications interactives. Nous nous sommes servis de FlowVR pour valider nos contributions.

FlowVR est un bon candidat pour représenter des middlewares pour le développement des applications interactives. Des applications de visualisation scientifique ou de réalité virtuelle ont déjà été développées grâce à cette suite logicielle. Nous pensons qu'avant de présenter en détail nos contributions, il est important de faire la présentation de ce logiciel. Cette présentation donne les éléments nécessaires à la compréhension de l'architecture et les spécificités d'un middleware pour les applications interactives.

5.1 Présentation de FlowVR

FlowVR [42] est un middleware dont le développement a débuté en 2002 en collaboration avec l'Université d'Orléans. Ce travail est le coeur de la thèse de Jérémie Allard [85]. FlowVR a, entre autre, été intensivement utilisé pour le développement d'une application de reconstruction 3D interactive, thème central de la thèse de Clément Menier [86]. Mon travail s'inscrit dans la continuité du développement de FlowVR en se focalisant sur les problèmes de la description de l'application, thème qui n'avait pas été abordé de manière approfondie. Pour ces raisons, nous ne détaillons pas ici l'ensemble des caractéristiques et fonctionnalités de FlowVR, mais introduisons l'essentiel nécessaire pour la présentation des extensions développées dans le cadre de cette thèse.

L'objectif de FlowVR est de fournir les outils nécessaires au développement d'applications interactives sur les architectures parallèles. La suite FlowVR com-

porte plusieurs éléments [2] :

FlowVR : Les outils permettant de développer les applications interactives. C'est essentiellement cette partie que nous présentons dans la suite

FlowVR-Render [87] : Primitives de haut-niveau basées sur les shaders graphiques qui permettent de rendre une scène 3D de façon distribuée sur un cluster graphique.

VTK-FlowVR : Composant VTK qui permet de produire des primitives graphiques FlowVR-Render. Cela donne la possibilité d'utiliser VTK sur un mur d'images par exemple.

FlowVR-VRPN [88] : Encapsulation du serveur VRPN dans un composant FlowVR. Cela donne la possibilité d'utiliser simplement les périphériques de réalité virtuelle supportés par VRPN dans une application FlowVR.

Des outils dédiés à FlowVR pour le développement (visualisateurs de graphes, de traces d'exécution, etc ...)

Comme nous l'avons rappelé dans l'introduction, FlowVR est un projet actif. Une dizaine de doctorants, ingénieurs ou chercheurs ont travaillé sur FlowVR depuis 2002. FlowVR est codé en C++. D'après les statistiques de téléchargement et les discussions sur la mailing-list du projet, la dernière version de FlowVR sortie en avril a été téléchargée par plus d'une centaine d'utilisateurs.

5.2 Fonctionnement de FlowVR

Une application FlowVR est composée de *modules* échangeant des données à travers un *réseau FlowVR*. Un module est souvent un code déjà existant qui a été modifié pour appeler les routines FlowVR. Un module est un processus indépendant ou un thread. Cela permet de modifier plus facilement une application existante pour l'adapter en module FlowVR.

Du point de vue de FlowVR, les modules ne sont pas conscients de l'existence des autres modules. Cela les rend comparables de ce point de vue aux composants. Un module échange des données uniquement par l'intermédiaire d'un démon FlowVR fonctionnant sur la même machine. Un ensemble de démons fonctionnant sur un cluster de PC sont en charge de la gestion du réseau FlowVR qui relie tous les modules. Cette approche permet de développer une librairie de modules qui peuvent être ré-assemblés dans différentes applications. Le réseau peut être modifié sans avoir besoin de recompiler les modules eux-mêmes.

Le réseau FlowVR peut contenir des simples connexions module à module ou bien des schémas de communications complexes. Le réseau peut implémenter des synchronisations, des opérations de filtrage de données, d'échantillonage, de la

prédiction, etc. Il est donc possible d'adapter les méthodes de transfert de données selon le contexte d'utilisation comme l'architecture d'un cluster. Ceci permet d'optimiser les performances de l'application.

Il y a trois étapes lors du développement d'une application FlowVR. Il faut réunir les modules nécessaires à l'application. Certains modules sont disponibles dans la distribution FlowVR. Si le module souhaité n'existe pas, il est possible de le programmer grâce à une API simple. Les fonctions offertes par FlowVR seront détaillées dans la section 5.2.2. La deuxième étape est la construction du réseau. Les premières versions de FlowVR proposaient des scripts Perl permettant de construire le réseau. L'objectif de ce chapitre est de présenter un langage hiérarchique permettant de construire un réseau représentant les transferts de données entre modules. La dernière version de FlowVR propose donc une implémentation en C++ de ce modèle hiérarchique. Enfin, la dernière étape du développement consiste à adapter l'application par rapport à une architecture cible donnée. Cela consiste à donner un placement de l'ensemble des objets FlowVR sur l'architecture cible ou d'instancier des paramètres permettant de spécialiser l'application.

A partir de ce placement, le réseau FlowVR se met en place par les démons fonctionnant sur chacune des machines. Si un module doit envoyer un message sur le réseau FlowVR, alors il alloue un tampon dans un segment de mémoire partagée géré par le démon local. Si ce message doit être transmis à un module situé sur la même machine, le démon a juste besoin de transmettre au module cible l'adresse du message dans la mémoire partagée. En effet, tous les modules en local peuvent accéder en lecture au segment de mémoire partagée. Par contre, si le message doit être transmis à un module situé sur une machine distante, alors le démon a la charge d'envoyer le message au démon hébergé sur la machine distante. Cet autre démon reçoit alors le message, le copie dans son propre segment de mémoire partagée et en donne l'adresse au module cible. En utilisant une mémoire partagée, il est possible de réduire le nombre de copies de messages. Cela peut améliorer les performances.

5.2.1 Messages

Un message FlowVR contient une entête composée d'une *liste d'estampilles*. Une estampille est une donnée qui permet d'identifier le message. Certaines estampilles sont automatiquement mises en place par FlowVR. L'utilisateur peut aussi définir de nouvelles estampilles si nécessaire. Une estampille peut être un ordre entre les messages, un identifiant de l'entité qui a généré le message ou des données plus complexes comme une boîte englobante pour les données 3D par exemple. Il est possible de faire des calculs directement à partir des estampilles sans avoir besoin de lire le contenu du message. La liste des estampilles peut être routée indépendamment du contenu du message dans le cas où la destination n'a

besoin uniquement que des estampilles. Il est donc possible d'envoyer le minimum d'informations requis sur le réseau.

5.2.2 Modules

Les tâches de calculs sont encapsulées dans les modules. Chaque module définit une liste de ports d'entrée ou de sortie. A l'exécution, le module itère sans fin. A chaque itération, il lit les données sur les ports d'entrée et produit de nouveaux résultats sur les ports de sortie. FlowVR propose les routines suivantes permettant de développer les modules :

- La fonction *wait* déclenche une nouvelle itération. Cette fonction est bloquante. La fonction rend la main uniquement quand un nouveau message est prêt sur tous les ports d'entrée. Avec cette sémantique, il est donc indispensable qu'un nouveau message soit reçu sur chacun des ports pour qu'une nouvelle itération démarre.
- La fonction *get* permet de récupérer le message disponible sur un port.
- La fonction *put* permet d'écrire un message sur un port de sortie. Il est possible d'écrire au maximum un seul message par port et par itération.

Les modules ont deux ports prédéfinis appelés *beginIt* et *endIt*. Le port *beginIt* est un port d'entrée recevant des signaux d'activation. Il est possible de contrôler l'exécution du composant grâce à ce port. Par exemple, en branchant sur ce port un objet qui génère un signal périodique, cela permet de contraindre la fréquence du composant.

Le port *endIt* est un port de sortie qui indique la fin d'itération du composant. Une utilisation possible est l'implémentation d'une politique client-serveur. Le signal du port *endIt* peut être relié à un serveur qui produira des données à la demande du composant.

Il suffit donc d'ajouter ces trois fonctions dans un code existant pour le transformer en module FlowVR. La communauté FlowVR a déjà développé des modules pour encapsuler les logiciels Sofa [89], Elkano [1], VTK [15], VRPN [88], Ogre3D [2] ou Gromacs [90].

5.2.3 Compilation et déploiement de l'application

La compilation d'une application FlowVR est représentée par la figure 5.1. Il y a deux chemins distincts. Le premier, grâce aux outils de compilation classique du C++, permet de compiler les processus qui seront les modules de l'application. Le deuxième construit le réseau. Le réseau est donné sous la forme d'un fichier XML.

1. http://iparla.labri.fr/software/elkano/
2. http://www.ogre3d.org/

Ce fichier XML avant l'implémentation du modèle hiérarchique présenté dans cette partie, était généré grâce à des scripts en Perl. La visualisation du fichier XML est réalisable grâce à l'outil *flowvr-glgraph* inclus dans la distribution FlowVR. Les différents schémas d'applications FlowVR de ce manuscrit sont tous générés grâce à cet outil.

Pour le déploiement de l'application, un module particulier (*controller*) est en charge de créer les différents modules. Une fois que les modules sont lancés, ils s'inscrivent auprès du démon local. Une fois l'opération terminée, chaque module se signale alors au contrôleur. Le contrôleur peut alors diffuser aux démons la description du réseau FlowVR. A la réception de la description, les démons construisent le réseau (chargent les plugins, créent les liens entre les ports, ...). Quand le réseau est complet, l'application peut alors démarrer.

5.2.4 Objets formant le réseau FlowVR

Les démons utilisent un système de *plugins* qui permet d'étendre les fonctionnalités du démon. Par exemple, un plugin TCP ajoute le support aux communications entre machines. Il est donc possible de développer d'autres plugins qui supporteront des réseaux hautes-performances. Ces plugins peuvent aussi être utilisés pour construire des nouveaux objets FlowVR.

Le modèle de programmation des modules peut quelquefois être trop contraignant pour la réalisation de certaines tâches. En programmant des objets directement au niveau du démon, il est possible d'utiliser des appels de méthodes bas-niveau mais plus souples. Pour compléter la description de FlowVR, nous présentons une courte liste des objets formant le réseau FlowVR. Ces objets seront utilisés dans la suite du manuscrit.

Connexions

Les connexions transmettent les messages d'un port de sortie à un port d'entrée. Ce sont des connexions point à point. Une connexion est implémentée avec une file FIFO. L'ordre des messages est inchangé sur la connexion. Dans les graphes FlowVR, elles sont représentées par des flèches.

Les *ConnectionStamps* sont des connexions transmettant uniquement les estampilles des messages. Elles sont représentées par des flèches en pointillés. Certains traitements comme le routage se réalisent uniquement sur les estampilles des messages. Il n'est pas nécessaire alors d'envoyer l'ensemble du message pour réaliser ces opérations. Les *ConnectionStamps* ont été créées pour réaliser ces optimisations.

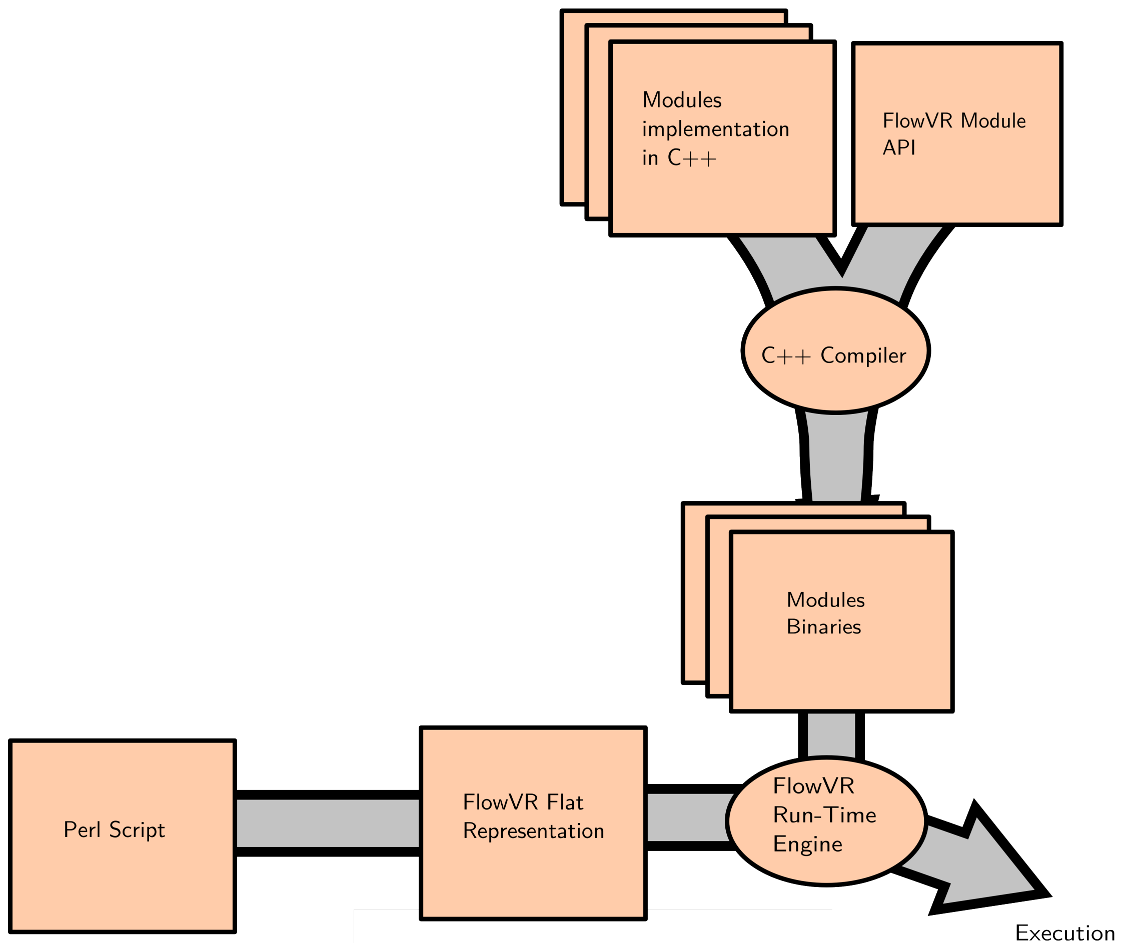


FIGURE 5.1 – Chemin de compilation d'une application FlowVR utilisant les scripts Perl pour la définition du réseau. De haut en bas, la compilation des modules à partir du code C++. De gauche à droite, la génération du réseau FlowVR.

Filtres

Les *Filtres* sont des objets pouvant transformer les flux de messages. Un exemple de filtre est le *FilterMerge* qui concatène au sein du même message des messages provenant de sources différentes. Ces filtres sont aussi capables de supprimer ou de rejouer des messages. Ils sont représentés par des losanges bleus. Ils sont exécutés par des threads créés par le démon. Contrairement aux modules, les filtres ont accès à des fonctions bas-niveau du démon. Grâce à ces fonctions, il peuvent modifier leurs politiques de synchronisation ou la gestion des messages au niveau des ports. Ils autorisent donc plus de souplesse que les modules.

Synchroniseurs

Les *Synchroniseurs* sont des filtres spécialisés dans le traitement des estampilles. Ils sont essentiellement utilisés pour contrôler les filtres (routage des messages, modification du flux de message,...) ou pour contrôler l'exécution des modules (mise en pause, contrainte de la fréquence,...).Ils sont représentés par des rectangles roses. Ils sont exécutés par des threads créés par le démon.

5.3 Bilan

Nous avons présenté dans ce chapitre un aperçu de FlowVR. Cette présentation détaille l'état du logiciel en 2006 au commencement de mes travaux de thèse. A cette date, les principaux développements sur le logiciel avaient eu pour objectif l'optimisation du moteur d'exécution.

Nous avons utilisé FlowVR pour la validation des contributions présentées dans la suite du manuscrit. Les autres middlewares de réalité virtuelle proposent un langage de description similaire à FlowVR. Des scripts permettent de décrire un graphe plat de l'application. Pour cette raison, nous pensons que FlowVR est un représentant de l'ensemble de ces middlewares. Nos contributions ne se limitent pas uniquement à FlowVR et pourraient être adaptées aux autres logiciels.

Chapitre 6

Hiérarchie de composants

6.1 Motivation

Comme nous l'avons montré dans l'état de l'art, les applications interactives de grande taille comptent plusieurs centaines de composants. Le réseau représentant les transferts de flux entre les tâches est un graphe. La plupart des middlewares utilisent un langage de script pour la description de graphe. Ces scripts sont essentiellement basés sur une méthode *link* qui relie deux composants entre eux. FlowVR dans ses premières versions proposait des scripts Perl qui généraient une représentation XML du graphe.

Certaines structures dans les graphes d'applications se retrouvent de façon récurrente. Il y a par exemple les communications NxM pour le couplage de codes parallèles. Mais on peut retrouver dans la plupart des applications des schémas de synchronisation entre les composants. Le besoin d'encapsuler ces schémas dans des méthodes apparaît donc rapidement. Cela permet de réutiliser ces schémas et de mieux structurer l'application.

Avec les langages de script, il est difficile de réaliser cette encapsulation. Ces schémas sont souvent des patrons de conception. Ils doivent être instanciés selon le contexte d'utilisation. Par exemple pour un arbre de diffusion, sa structure dépendra du placement et du nombre de modules qui seront reliés aux feuilles de l'arbre. Il est donc nécessaire de passer l'ensemble de l'application en paramètre de la fonction qui va réaliser cet arbre. L'écriture de ces patrons devient alors plus complexes.

L'idée principale qui a motivé le modèle est de proposer un moyen simple pour encapsuler ces patrons de conception. Les communications NxM ou les synchronisations ont un impact important sur les performances de l'application. Il est important de proposer un mécanisme qui permet facilement de réutiliser des schémas efficaces pour l'ensemble des applications.

Le modèle de description hiérachique produit après compilation une répresentation d'un graphe. Le graphe produit est indépendant du middleware. Pour valider le modèle, nous en avons écrit une implémentation dédiée à FlowVR. Mais, il est possible d'utiliser notre modèle pour l'adapter au dessus d'autres middlewares.

Ce chapitre présente le modèle de composant hiérarchique. Il est inspiré de la norme Fractal. Par conséquent, on peut retrouver des concepts et du vocabulaire communs (*Composite, Primitive, Controllers*, etc ...). L'exemple présenté au chapitre précédent permettra d'illustrer certaines définitions.

6.2 Interface

Un composant est muni d'une interface définie par un ensemble de ports. Il existe deux types de ports :

Port d'entrée Muni d'une file de messages FIFO, à chaque itération le port consomme le message en tête de file.

Port de sortie La tâche à chaque itération produit des messages dans la file du port de sortie.

Ce système de typage est faible, cependant le modèle n'empêche pas d'étendre ce système pour proposer aux développeurs un système de typage fort.

6.3 Type de composants

Nous définissons maintenant deux types de composants :

Composants *primitifs* Ce type de composant est la brique de base. Un composant primitif ne peut pas contenir d'autres composants. En pratique, ces composants représentent les objets physiques qui seront exécutés dans l'application. Par exemple, ce sont les modules, les filtres, les nœuds de routage ou alors les canaux de communications ou connexions. Ces composants représentent une boucle d'itération.

Composants *composites* Un composant composite contient d'autres composants (composite ou primitif). Les ports de l'interface de ce composant sont aussi bien visibles de l'intérieur ou de l'extérieur du composant. Ces ports sont les points de passage pour que les données traversent la membrane du composant. Dans le modèle, l'encapsulation est stricte. Un composant peut être directement contenu au maximum dans un seul composant.

Il est à noter que dans le modèle l'ensemble des objets sont considérés comme des composants. Une connexion est aussi un composant. C'est un composant dédié

au transfert de données. Tous les composants ont des ports. La connexion est donc elle aussi munie d'un port d'entrée et d'un port de sortie.

Comme l'encapsulation est stricte, il est possible de représenter les relations entre les composants par une structure d'arbre. Les composants primitifs sont donc les feuilles de l'arbre et les composites sont tous les autres nœuds. Un composant A est contenu dans un composant B si A est fils de B. L'équivalence avec la structure d'arbre, permet d'utiliser le même vocabulaire pour exprimer les relations entre composants. Par exemple, nous appellerons le père d'un composant, le composant qui le contient. La racine de l'arbre représente le composant qui encapsule l'ensemble de l'application.

6.4 Les liens

Un lien relie deux ports. L'encapsulation étant stricte, nous imposons que les liens ne puissent pas traverser les membranes du composant. Un lien entre deux ports est possible uniquement dans les deux cas suivants :

- Un lien selon l'ascendance (*parent link*) relie le port d'un composant composite à un des ports de ses enfants. Ce lien doit connecter deux ports de même type (port d'entrée vers port d'entrée ou port de sortie vers port de sortie).
- Un lien entre frères (*sibling link*) relie deux ports de deux composants ayant le même père. Ce lien doit connecter deux ports ayant des types conjugués (un doublet port d'entrée / port de sortie).

6.5 Application sur un exemple

Pour illustrer le modèle, nous nous basons sur un exemple simple (Fig. 6.1). Cette application simule la trajectoire d'une balle tombant dans un bassin d'eau. Le résultat de la simulation permet de générer une scène virtuelle. La scène est alors rendue selon un point de vue donné par l'utilisateur.

La structure de cette application est classique pour les applications interactives. Un composant *Compute* calcule à chaque itération la position de la balle. Les résultats sont envoyés à travers un composant de communication *Connect* au composant de rendu *Render*. Le composant *Capture* récupère les mouvements de la souris pour obtenir le point de vue de l'utilisateur. Ce point de vue est lui aussi envoyé au composant *Render* pour produire l'image finale.

Pour ne pas surcharger l'exemple et faciliter la compréhension du modèle, nous imposons que l'application soit synchrone. Par conséquent le composant *Render* peut démarrer une nouvelle itération uniquement quand il reçoit une nouvelle donnée de *Computes* et *Captures*. Dans la pratique, ce comportement est peu utilisé.

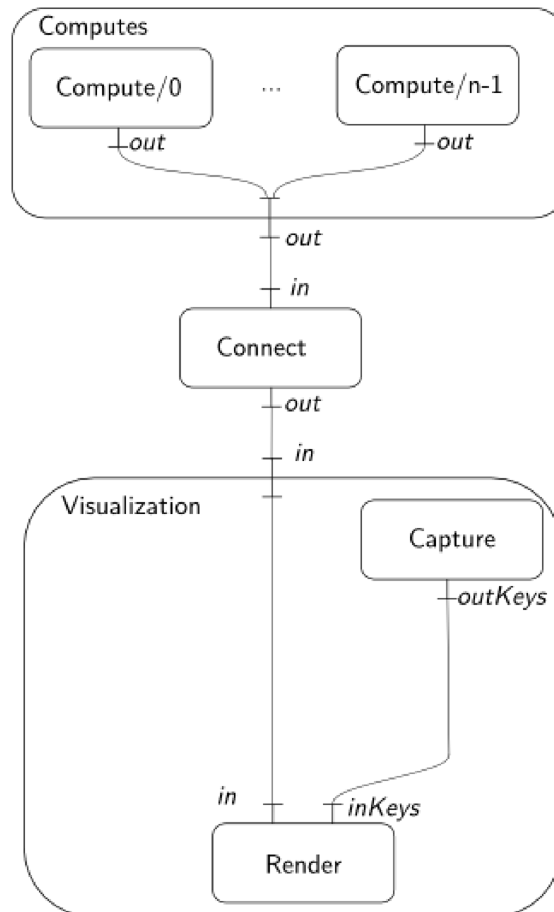


FIGURE 6.1 – L'application servant d'exemple. *Computes* simule la trajectoire d'une balle tombant dans un bassin d'eau. Les résultats sont envoyés au composant *Render* pour le rendu. *Capture* envoie les mouvements de la souris à *Render* qui les utilise pour rendre la scène depuis le point de vue contrôlé par l'utilisateur.

La plupart des applications permettent aux composants de simulation et de rendu de fonctionner à des fréquences différentes. Cet asynchronisme peut s'implémenter en utilisant des composants spécialisés dans l'échantillonnage de données. Dans la section 7, nous présentons une méthode pour réaliser ces schémas avancés de communication.

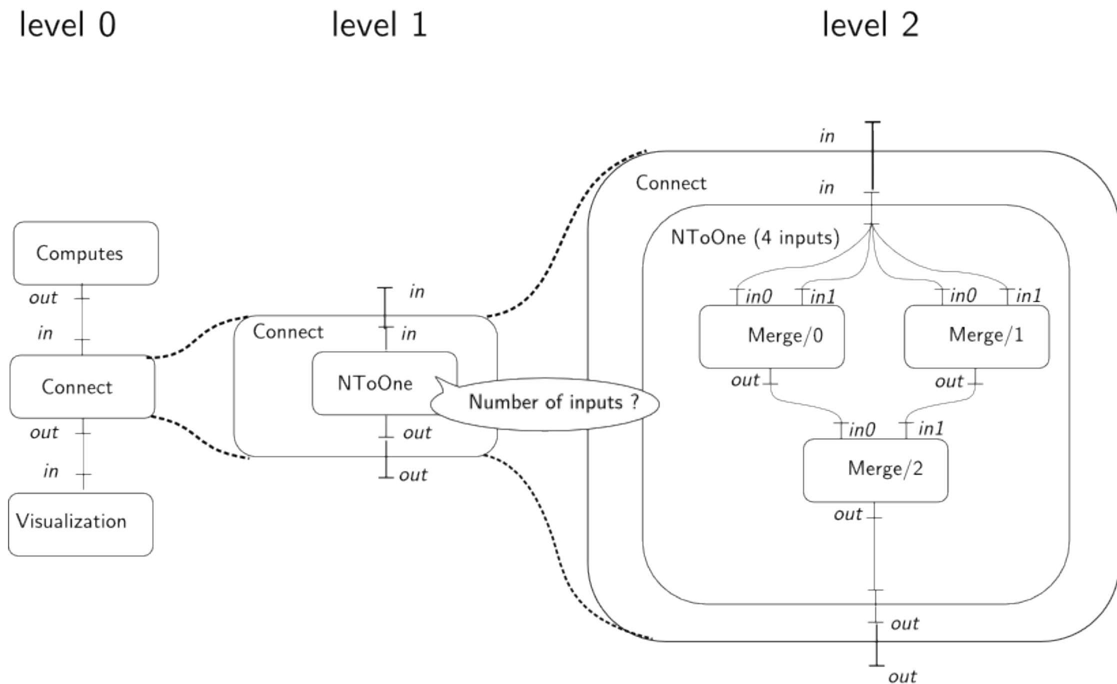
En utilisant le modèle hiérarchique, l'exemple est structuré de la façon suivante (Fig. 6.1) :

Les composants *Capture* et *Render* sont fortement liés l'un à l'autre. Ils sont souvent utilisés en même temps pour obtenir la visualisation d'une scène. Pour cette raison, ils sont rassemblés au sein d'un même composant composite *Visualization*. Cette encapsulation permet de réutiliser plus facilement cette association. En effet, en utilisant le composant *Visualization*, le développeur obtient directement un processus de rendu de la scène gérant le point de vue grâce à la souris.

Le composant *Computes* est une application parallèle qui instancie n processus. Cela permet d'accélérer la simulation en la calculant sur plusieurs processeurs. *Computes* est donc un composant composite avec pour interface un port de sortie *out* qui envoie les résultats de simulation après chaque itération. Ce composant contient n fils *Compute/0*, ... *Compute/n-1*. Chacun de ces fils a pour interface un port de sortie *out* relié au port *out* de *Computes*. Le nombre effectif de processus n et le placement de chacun des processus sur l'architecture cible n'est pas connu lors du développement de l'application. Le placement devra en effet être calculé selon le contexte d'exécution pour s'adapter à l'architecture cible. Le mécanisme en charge du calcul est présenté dans la section 6.7.

D'autre part, il existe certainement des communications entre les différents processus *Compute/i*, mais elles ne sont pas modélisées ici. En effet, nous considérons qu'elles sont liées à l'implémentation de l'algorithme parallèle et ne sont pas visibles à notre niveau. Cette représentation est donc indépendante de l'algorithme ou du middleware utilisé pour implémenter les n processus *Compute*. Une implémentation utilisant des pthreads ou une utilisant MPI seront représentées par le même graphe hiérarchique. Cependant, dans la pratique, des différences entre les deux implémentations peuvent apparaître dans la description du déploiement. Nous verrons en détail ce point dans la section 6.6.

Computes est un composant parallèle : chaque processus *Compute* est en charge de calculer une partie de la simulation. *Visualization* est capable de lire l'état complet de la simulation. Les résultats de chacun des processus *Compute* doivent donc être rassemblés avant d'être envoyés à *Visualization*. L'opération d'union pourrait être incluse dans le composant *Visualization*. En effet, l'opération d'union des messages se réaliserait dans le port du composant de visualisation. Nous écartons cette approche, car elle consisterait à changer l'interface du composant *Visualization* selon la nature de *Computes*. Cela est en contradiction avec le principe des



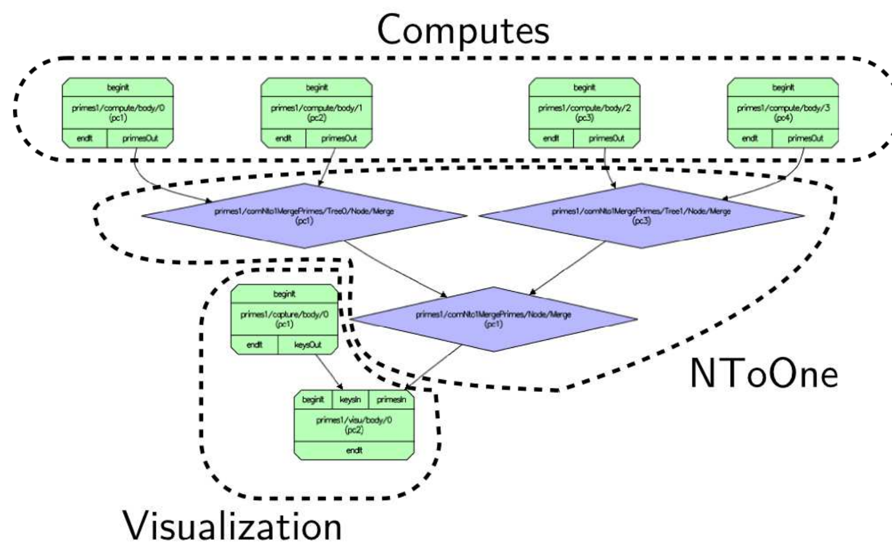


FIGURE 6.2 – a) Les deux niveaux de hiérarchie pour le composant *Connect*. Le squelette du composant *NtoOne* est construit à partir du nombre d'instances du composant primitif *Compute*. b) Le graphe de flot de données plat de l'application. Les groupes de composants délimités par les tirets sont le résultat de la compilation de chacun des composants composites. Les connexions sont représentées par les flèches, les modules sont les objets en vert et les filtres en bleu.

composants qui considère que l'interface d'un composant doit être indépendante de son contexte d'exécution. Ici nous préférons déléguer l'opération d'union à un nouveau composant spécialisé (*Connect*). Cette solution a l'avantage de proposer une méthode générique pour coupler des composants parallèles avec des composants séquentiels. *Connect* rassemble donc les résultats partiels de chacun des processus *Compute/i* et produit un message unique contenant le résultat global de la simulation. Ce message est alors envoyé au composant *Visualization*.

Connect est un composant composite (Fig. 6.2.a). Ce composant est construit à partir du composant *NtoOne*. Ce composant implémente de façon générique un squelette d'arbre de redistribution de données. Le composant *Connect* instancie ce composant *NtoOne* avec les paramètres adéquats : l'arité de l'arbre (deux dans l'exemple) et le type des composants des nœuds de l'arbre (le composant *Merge* dans l'exemple). Le nombre de feuilles de l'arbre est le nombre de processus *Compute*. Ce nombre n'est connu uniquement qu'après la construction du composant *Computes*. L'arbre ne peut donc être créé qu'après la construction du composant *Computes*.

Au final, après le dépliement total de l'application, celle-ci comportera les composants suivants : *Computes, Visualization, Connect, NtoOne, Merge/0, Merge/1, Merge/2, Compute/0, Compute/1, Compute/2, Compute/3, Capture, Render*. Le composant *Merge* permet de juxtaposer au sein du même message les deux messages arrivant sur ses deux ports d'entrée *in/0* et *in/1*. On peut remarquer que s'il y a un seul composant *Compute/0* dans *Computes* alors le composant *Connect* devient une simple connexion point à point.

6.6 Contrôleur

On appelle *dépliement*, l'opération qui consiste à construire les fils d'un composant et à produire les liens entre les ports. Sur l'exemple précédent, nous remarquons que nous devons respecter un ordre pour déplier l'application. En effet, ici le composant *Computes* doit être déplié avant le composant *Connect*. Dans cette partie, nous allons présenter les différentes opérations (*Controller*) que l'on peut appliquer sur les composants et une méthode simple (*Traverse*) pour déterminer un ordre correct d'exécution d'une opération sur une application complète.

Les contrôleurs sont en charge d'appliquer localement sur un composant une opération. A chaque contrôleur est affecté un unique composant. Un contrôleur peut modifier uniquement ce composant. Par contre, ils peuvent lire l'état des composants reliés directement ou non à ce composant. Un même composant peut avoir plusieurs contrôleurs. On distingue deux types de contrôleurs :

Les contrôleurs d'introspection accèdent à l'état du composant. Par exemple, un contrôleur peut afficher dans un flux de sortie le nom du composant

Les contrôleurs de configuration modifient l'état du composant. Dans l'exemple précédent, les processus n *Compute* ont été créés par un contrôleur de configuration.

Le dépliement d'un composant peut potentiellement se réaliser en plusieurs étapes. Dans notre implémentation, nous appelons *execute*, le contrôleur qui est en charge de la première étape du dépliement de chaque composant. Ce contrôleur doit être implémenté par tous les composants.

Par exemple, le contrôleur *execute* du composant *Computes* crée n composants primitifs *Compute/i*. Ce contrôleur produit aussi les liens entre les ports. Ici, le port *out* de chaque *Compute/i*, sera relié au port *out* de *Computes*.

Les contrôleurs des composants dédiés aux communications collectives ont souvent besoin de récupérer des données venant des composants dans leurs entourages. Par exemple, le contrôleur du composant *NtoOne* a besoin de connaître le nombre de composants primitifs *Compute/i* pour obtenir le nombre de feuilles de son arbre.

Il est possible de créer de nouveaux contrôleurs. Un contrôleur pourra implémenter un aspect de l'application. Par exemple, il pourra effectuer le placement des composants primitifs sur les processeurs de l'architecture. De cette manière, lors de la conception de l'application, le développeur ne se souciera pas du placement. La méthode de placement sera contenue au sein de ce nouveau contrôleur. Le design de l'application se fera donc de façon générique sans prendre en compte le placement ou l'architecture cible. Il sera facilement possible de modifier la méthode de placement ou l'architecture cible sans modifier le design de l'application. Dans FlowVR, le contrôleur *execute* est toujours appelé en premier avant tous les autres contrôleurs.

Voici un autre exemple de contrôleur : les modules FlowVR sont souvent des processus UNIX. Il faut donc générer un fichier rassemblant toutes les lignes de commandes nécessaires à leurs lancements. Ce fichier sera exécuté lors du déploiement de l'application. Ces lignes de commandes sont générées par un contrôleur spécifique. Ce contrôleur construit la ligne de commande à partir des informations du réseau FlowVR (liste des machines hôtes, nombre de processus), de fichiers de configuration (description de l'architecture cible) ou de paramètres donnés par l'utilisateur.

Ce contrôleur se retrouve fréquemment dans les applications FlowVR. Pour faciliter son utilisation, nous avons créé un composant composite particulier qui embarque directement ce contrôleur. Ce composant s'appelle le *métamodule*. Le métamodule encapsule tous les composants générés par l'exécution d'une ligne de commande. En général un metamodule représente un seul processus UNIX, il aura donc un seul fils. Mais dans l'exemple précédent, le composant *Computes* est un métamodule qui lancera tous les processus MPI représentés par les composants primitifs *Compute/i*. Il contiendra alors n composants primitifs. Ici, le contrôleur

devra générer une ligne de commande qui utilise un lanceur MPI comme *mpirun*.

Au final, le métamodule est donc un exemple de relation logique entre plusieurs composants. Ici la hiérarchie permet de regrouper au sein du même composant, tous les composants qui sont générés par la même ligne de commande.

La hiérarchie et les contrôleurs permettent d'appliquer des comportements à une partie de l'application. Prennons l'exemple d'une application répartie sur deux clusters administrés indépendamment. Il est fort probable que les versions de logiciels diffèrent entre les deux ensembles de machines. Si les deux clusters n'utilisent pas le même logiciel de déploiement de processus, il sera possible ici, par l'intermédiaire des contrôleurs de générer les lignes de commandes adéquates selon le cluster. Le contrôleur a une action locale sur un composant, il pourra grâce aux fichiers de configuration connaître la version du lanceur utilisé sur le cluster hébergeant le composant. Il générera donc pour ce composant la ligne de commande correspondante. Au sein de la même application, il sera donc possible de gérer l'hétérogénéité de l'architecture cible. Par contre au niveau de l'implémentation du graphe (programmation des contrôleurs *execute* de l'application), cette hétérogénéité n'est pas visible.

Il y a de nombreuses similitudes entre les contrôleurs et les aspects (dans le sens programmation par aspect). Néanmoins, ce modèle ne peut pas se revendiquer comme un modèle de programmation par aspect. Les mécanismes mis en place ici sont plus simples. Par exemple, il n'y a pas de mécanisme de tissage de code (*code weaving*) pour insérer les aspects dans l'application finale. Ici, les contrôleurs sont seulement liés au composant par l'action du programmeur.

6.7 *Traverse*

Nous avons vu que l'ordre d'appel aux contrôleurs est important. Certains contrôleurs nécessitent que leurs voisins soient exécutés pour être eux-mêmes à leur tour exécutés. Il est nécessaire de trouver un ordre de parcours respectant ces contraintes. Nous utilisons ici un algorithme glouton appelé *Traverse*. L'algorithme détecte automatiquement les contraintes en tentant d'exécuter les contrôleurs.

Pour pouvoir détecter les contraintes en tentant d'exécuter les contrôleurs, l'implémentation doit être capable de :

- Détecter si l'exécution du contrôleur a échoué ou réussi ;
- Rétablir l'état d'un composant à l'état où il était avant l'appel du contrôleur (opération *rollback*).

Pour connaître le succès de l'exécution du contrôleur, nous nous basons sur les exceptions du C++. Pour signaler son échec, un contrôleur doit envoyer une exception. Cette exception est alors récupérée par l'algorithme de traverse qui essaiera de faire l'opération de retour en arrière. Pour simplifier le développement

des contrôleurs, nous avons créé une série d'exceptions héritant d'une classe mère commune (`CustomException`). Pour chaque type d'erreurs, il existe une classe d'exception permettant de décrire l'erreur. Ces classes d'exceptions ont une méthode permettant d'imprimer à l'écran une description précise de l'erreur. Ce système rend plus simple le débuggage de l'application.

Pour le programmeur non-expert, il n'est pas évident d'utiliser ces exceptions. Par conséquent, pour toutes les opérations classiques (ajout de ports, de composants fils, recherche sur les voisins, ...), nous avons implémenté des méthodes basées sur ces exceptions. Par exemple, si un utilisateur veut obtenir les composants primitifs reliés à un port d'un composant, il utilisera la famille des méthodes `getPrimitiveSiblingLinks*`. Ces méthodes enverront alors automatiquement une exception si les voisins n'ont pas encore créé leurs composants primitifs. La figure 6.3 présente les recherches selon les liens implémentés dans notre modèle.

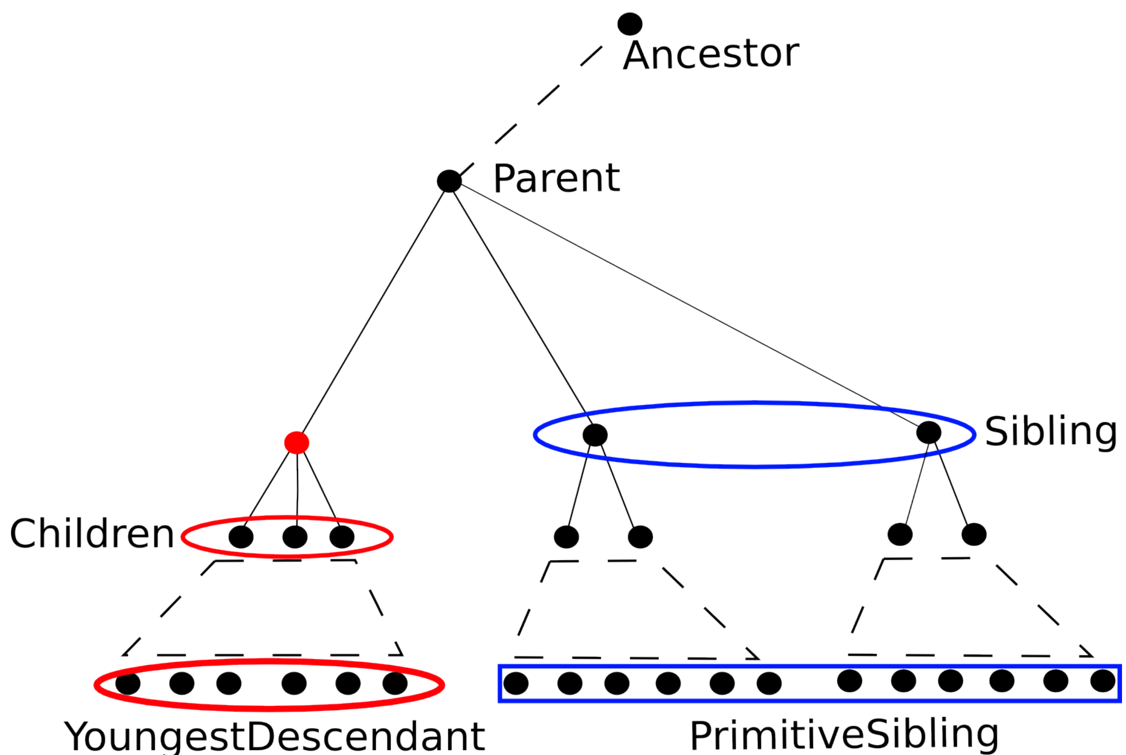


FIGURE 6.3 – Navigation dans la hiérarchie de composants. L'arbre représente la hiérarchie. Les noeuds sont les composants. Les branches de l'arbre sont les liens *Parent*. Les relations sont données par rapport au noeud en rouge. Les fonctions de l'implémentation C++ retournent soit des pointeurs soit des itérateurs vers les ensembles décrits dans cette figure

Pour implémenter l'opération de *rollback*, l'algorithme *traverse* clone tous les

composants avant d'exécuter leurs contrôleurs. Si le contrôleur échoue, le clone remplace alors le composant dans la description de l'application. Le composant dont le contrôleur a échoué est alors détruit. Cette opération nécessite d'avoir une opération de clonage des composants. Pour cette raison, il est nécessaire de définir une méthode (appelée ici `create`) qui clone les composants.

L'algorithme *traverse* est un algorithme glouton. Il gère une file des composants inexécutés. Pour chaque composant de la file, l'algorithme va tenter d'exécuter le contrôleur sur ce composant. S'il échoue, l'état du composant est rétabli et le composant est mis au bout de la file. S'il réussit, le composant est enlevé de la file. Si ce composant a des fils, ils seront alors insérés dans la file. L'algorithme se termine soit quand la queue est vide, soit quand il n'est plus possible d'exécuter aucun composant de la file. La figure 6.4 présente l'implémentation de l'algorithme *traverse* dans FlowVR.

Preuve de l'algorithme

Nous allons prouver l'algorithme. Les propriétés que nous souhaitons sont :

- Dans tous les cas, le temps d'exécution de l'algorithme est fini : il se terminera toujours.
- S'il existe un ordre d'exécution des composants, alors et alors seulement l'algorithme dépliera l'ensemble des composants.

Nous définissons quelques notations. Soit C l'ensemble des composants de l'application. N_{comp} est la taille de C c'est-à-dire le nombre total de composants. L'algorithme parcourt grâce à un pointeur une liste de composants qui n'ont pas encore été exécutés. Nous prenons de façon arbitraire une position de la liste qui sera la tête de liste. Nous considérons que la liste est cyclique. C'est à dire quand l'algorithme a parcouru tous les éléments de la liste, le pointeur revient en tête de liste.

Dans la suite, nous définissons le nombre d'itérations de l'algorithme *traverse* comme le nombre de fois que le pointeur a parcouru la tête de liste.

Soit l'ensemble $NonExecute_k$, des composants qui n'ont pas encore été exécutés à l'itération k et l'ensemble $Execute_k$, des composants qui ont été exécutés durant l'itération k.

Si l'algorithme a atteint un point fixe, alors aucun composant n'a pu être exécuté durant une itération de l'algorithme. Par conséquent, si l'algorithme a atteint un point fixe à l'itération N alors $Execute_N = \emptyset$. Dans ce cas, l'algorithme se termine et l'ensemble des composants qui n'ont pas pu être exécutés sont $NonExecute_N$.

Soit l'itération k, on note E_k l'ensemble des composants qui ont pu être exécutés durant les k premières itérations. On a donc $E_k = \bigcup_{i=1}^{k} (Execute_i)$. L'ensemble complémentaire $\overline{E_k}$ des composants qui reste à exécuter à l'issu de l'itération k est

égal à $\overline{E_k} = \bigcup_{i=k}^{\infty} (NonExecute_i)$.

Enfin nous appelons $\overline{E_\infty} = \bigcap_{i=1}^{\infty} \overline{E_i}$, l'ensemble des composants qui ne pourront jamais être exécutés par l'algorithme.

Pour résumer, nous pouvons déduire les propriétés suivantes :

- E_k et $\overline{E_k}$ sont complémentaires, donc $C = E_k \oplus \overline{E_k}$
- Au démarrage de l'algorithme, nous avons $\overline{E_0} = C$ et $E_0 = \emptyset$

Dans un premier temps, nous allons démontrer que l'exécution de l'algorithme se termine toujours.

Proposition 6.7.1 *Il existe N tel que l'algorithme du traverse atteint un point fixe à l'itération N. On a alors $N \leq N_{comp}$ et $NonExecuted_N = \overline{E_\infty}$*

Preuve Tant que l'algorithme est en cours d'exécution, nous savons que $Executed_N \neq \emptyset$.

Par conséquent, pour tous k, nous avons :
$\overline{E_{k+1}} = \bigcup_{i=k+1}^{\infty} (NonExecuted_i) \subset \bigcup_{i=k}^{\infty} (NonExecuted_i) \subset \overline{E_k}$

En effet à chaque itération nous savons qu'au moins un composant a été exécuté.

Par conséquent la suite des ensembles $(\overline{E_N})$ est décroissante et a pour limite $\overline{E_\infty}$. Cette limite est par définition atteinte quand l'algorithme a atteint le point-fixe.

Par définition, tant que l'algorithme itère, nous savons que E_{k+1} est strictement inclus dans E_k. Comme E_0 comporte N_{comp} composants, nous pouvons en déduire qu'au maximum il y a N_{comp} itérations de l'algorithme. L'algorithme s'arrêtera, au maximum en N_{comp} itérations. Donc $N \leq N_{comp}$ ∎

6.8 Complexité

Proposition 6.8.1 *Durant l'exécution du traverse, le contrôleur est appelé au maximum N_{comp}^2 fois*

Preuve Le nombre d'appel total au contrôleur est $\sum_{k \leq N} |NonExecuted_k|$. Nous venons de démontrer précédemment que

- $N \leq N_{comp}$
- $\forall k, NonExecute_k \subset C \Rightarrow |NonExecute_k| \leq N_{comp}$

Nous avons donc $\sum_{k \leq N} |NonExecuted_k| \leq N_{comp}^2$ ∎

Interprétation

Cet algorithme est appelé lors de la compilation du graphe de l'application. Il sera donc appelé régulièrement par un développeur lors de l'implémentation de son application. Le but du modèle hiérarchique est de donner des outils au

développeur pour passer à l'échelle par rapport à la taille de son application. Par conséquent, nous visons des applications comportant plusieurs centaines voire milliers de composants. Il est donc indispensable que le *traverse* puisse passer à l'échelle pour que le modèle de programmation puisse au final être utilisé.

L'alternative à l'algorithme glouton est la recherche à partir d'une liste de contraintes d'un ordre acceptable de parcours des composants. Dans ce cas nous pouvons imaginer l'algorithme simple suivant :

1. On parcourt les contraintes pour générer un graphe de dépendance entre les composants.
2. Si le graphe est un DAG, il donne une relation d'ordre entre les composants qui correspond à l'ordre de parcours.

La complexité de ce simple algorithme est alors linéaire par rapport au nombre de composants et de contraintes. Par conséquent, au niveau des performances, il sera meilleur que le *traverse.*

Cependant, nous avons rejeté cet algorithme basé sur une liste de contraintes, car il ne passe pas à l'échelle d'un point de vue génie logiciel. En effet, la liste de contraintes devra être donnée par le développeur. Dans le cas de milliers de composants, on peut imaginer que cette liste comportera des milliers de contraintes. A cette échelle, la probabilité d'un oubli d'une contrainte est trop grand. Pour cette raison, il est indispensable que le parcours se fasse avec un minimum d'informations provenant du développeur.

L'expérience nous a montré que notre méthode est un bon compromis. Nous avons remarqué que dans le cas de nos applications comportant plusieurs centaines de composants, l'algorithme de *traverse* prenait au maximum quelques secondes. En contrepartie, aucune paramétrisation n'est demandée à l'utilisateur. L'utilisation de l'algorithme est donc assez simple.

6.9 Caractérisation de $\overline{E_\infty}$

Nous nous proposons de caractériser $\overline{E_\infty}$, l'ensemble des composants qui n'ont pas pu être exécutés. Soit $Data = \{c \in C \ / \ c$ n'a pas pu être exécuté à cause d'un paramètre manquant $\}$ et $Dep = \{c \in C \ / \ c$ n'a pas pu être exécuté parce qu'il dépend d'un autre composant qui n'a pas pu être exécuté $\}$. Il n'existe pas d'autre raison pour qu'un composant ne puisse être exécuté. Nous avons donc $\overline{E_\infty} = Data \cup Dep$.

Proposition 6.9.1 *Si* $Data = \emptyset$ *et* $\overline{E_\infty} = Dep \neq \emptyset$ *alors il y a au moins un cycle de dépendances dans* $\overline{E_\infty}$

Preuve Supposons qu'il n'existe pas de cycle de dépendances dans *Dep*. Il existe alors un plus long chemin en terme de dépendances entre composants. Soit c et d les composants aux extrémités d'un des plus longs chemins.

Data est vide. Donc c appartient à *Dep*. Il existe donc e dans *Dep* tel que c dépend de e. Le chemin de e à d est plus long que le chemin de c à d. C'est contraire à l'hypothèse que le chemin de c à d est un des plus longs chemin. Nous montrons donc par l'absurde que si $Data = \emptyset$, alors il existe au moins un cycle de dépendance dans $\overline{E_\infty}$. ∎

Cette proposition montre que l'algorithme de *traverse* peut aider à débugger l'application. Si l'algorithme échoue, l'utilisateur doit dans un premier temps vérifier qu'aucun paramètre n'a été oublié. Il est possible de programmer les contrôleurs pour qu'ils renvoient à l'utilisateur un message si un paramètre est manquant. L'utilisateur peut donc régler les problèmes liés à l'ensemble *Data*.

Si tous les problèmes ont été résolus et que l'algorithme de *traverse* échoue, la proposition nous indique qu'il existe un cycle de dépendances dans les composants qui n'ont pas pu être exécutés. L'utilisateur doit alors modifier l'application et supprimer les cycles. Dans notre implémentation, nous utilisons les exceptions C++ pour signaler l'échec d'un contrôleur. Il est donc possible de l'enrichir pour signaler la raison de son échec. Grâce à ce mécanisme, notre implémentation donne en sortie les ensembles *Dep* et *Data*.

6.10 Modification du chemin de compilation

Au final, l'implémentation du modèle pour FlowVR n'a pas modifié la partie exécutive du middleware. Nous donnons dans la figure 6.5 le nouveau chemin de compilation d'une application FlowVR. Si nous comparons avec la figure 5.1, la partie sur la compilation des modules n'a pas changé. Le démon FlowVR utilise toujours une description XML du graphe de l'application. Cette description est maintenant produite après le dépliement de la représentation hiérarchique de l'application.

En pratique, la représentation hiérarchique de l'application donnée grâce à l'interface en C++ est compilée sous forme d'une librairie partagée. Il est possible aussi de compiler un ensemble de composants hiérarchiques sous forme d'une librairie. Ces librairies peuvent donc être diffusées et être utilisées dans d'autres applications. C'est une réponse à la problématique initiale qui consistait à fournir un mécanisme pour encapsuler et diffuser des patrons de conceptions de schémas que l'on retrouve dans les applications interactives.

L'algorithme de *traverse* est implémenté sous la forme d'un exécutable. Cet exécutable prend en argument la librairie représentant l'application. Il est possible

de spécifier par l'intermédiaire de fichiers de configuration ou de la ligne de commandes des paramètres supplémentaires. Par exemple, l'utilisateur peut fournir une description de l'architecture cible sous forme d'un fichier XML à l'exécutable. Le placement des composants sur l'architecture cible peut être contraint en écrivant une feuille de tableur (au format CSV). Ce type de fichier est éditable par un tableur. Il est en projet de construire une petite interface graphique qui permettrait de donner les paramètres de chaque composant. Cela permettrait interactivement de vider l'ensemble *Data* lors du débuggage de l'application.

6.11 Bilan

Dans ce chapitre, nous avons présenté le modèle hiérarchique de description d'applications interactives. Ce modèle permet de construire un nouveau type de composant appelé *Composite* en assemblant d'autres composants. Ce modèle est inspiré de la norme Fractal.

Nous n'implémentons pas la norme Fractal. Pour s'adapter au contexte des applications interactives, nous avons défini volontairement un nouveau modèle. La hiérarchie nous semblait un élement important qui permet de structurer l'application. Notre idée qui s'inspire de la programmation par squelette est d'utiliser cette hiérarchie pour encapsuler les communications collectives. Ces communications ont un impact important sur les performances de l'application. Nous souhaitons fournir à terme une librairie implémentant quelques communications robustes.

Contrairement à de nombreuses implémentations de modèles à composants, nous ne visons pas la dynamicité. Nous faisons l'hypothèse que notre architecture n'évolue pas lors de l'exécution de l'application. Nous ne cherchons pas à reconfigurer ou modifier l'application en cours d'exécution. Pour les applications que nous visons, l'architecture est souvent dédiée à l'application. Il n'est pas gênant d'interrompre l'application pour réaliser une modification ou une reconfiguration.

Pour ces raisons, nous proposons de déplier la hiérarchie de composants grâce à une phase de compilation. Cette phase permet d'adapter l'application à l'architecture cible. Si cette architecture doit être modifiée, il n'est pas nécessaire de modifier la description de l'application. Seule une nouvelle phase de compilation est nécessaire.

Nous avons cherché un algorithme de dépliement performant. Notre objectif est le passage à l'échelle. Nous avons travaillé sur la complexité de l'algorithme. Nous souhaitons aussi que l'utilisateur fournisse le minimum d'information à l'algorithme pour éviter les risques d'erreurs. Pour aider l'utilisateur, notre algorithme est aussi capable de détecter certaines erreurs dans la description de l'application comme les cycles de dépendances entre les composants.

Nous avons enfin une implémentation du modèle au dessus de FlowVR. Cette

implémentation a validé le modèle. Elle est actuellement utilisée dans plusieurs applications de réalité virtuelle.

```
struct traverse {
    /* Public attribute. User can read exception raised during the last
       turn of traverse algorithm. */
    std::list<CustomException> listError;

    traverse() : listError() {}

    /**
    * \brief the greedy traverse algorithm
    * \param comp the component where the traverse will begin
    * \param control the contrôler
    */
    void operator() (Component* comp, Controller* control) {
        /* non executed queue */
        std::list<Component*> nonExecutedComponents;
        /* the failed components queue */
        std::list<Component*> failedComponents;
        /* initiate the traverse */
        nonExecutedComponents.push_back(comp);
        bool isModified;
        do {
            listError.clear();
            failedComponents.clear();
            isModified = false;
            while (!nonExecutedComponents.empty()) {
                Component* current = nonExecutedComponents.front();
                nonExecutedComponents.pop_front();
                /* store the initial state */
                Component* initialState = current->clone();
                try {
                    control->setComponent(current);
                    (*control)();     /* apply controller */
                    isModified = true;
                    if (current->isComposite())
                        nonExecutedComponents.insert(
                                    nonExecutedComponents.end(),
                                    current->getComponentBegin(),
                                    current->getComponentEnd()
                                    ); /* top-bottom traverse */
                    delete initialState;
                }
                catch (const CustomException& e) {
                    /* Restore initial state */
                    Composite* currentParent = current->getParent();
                    currentParent->exchange(current, initialState);
                    failedComponents.push_back(initialState);
                    listError.push_back(e);
                    delete current;
                }
            }
            nonExecutedComponents.swap(failedComponents);
        }while (isModified && !nonExecutedComponents.empty());
    }
};
```

FIGURE 6.4 – Algorithme du *traverse* utilisé dans FlowVR en C++. L'implémentation sépare les composants ayant échoués des composants non exécutes.

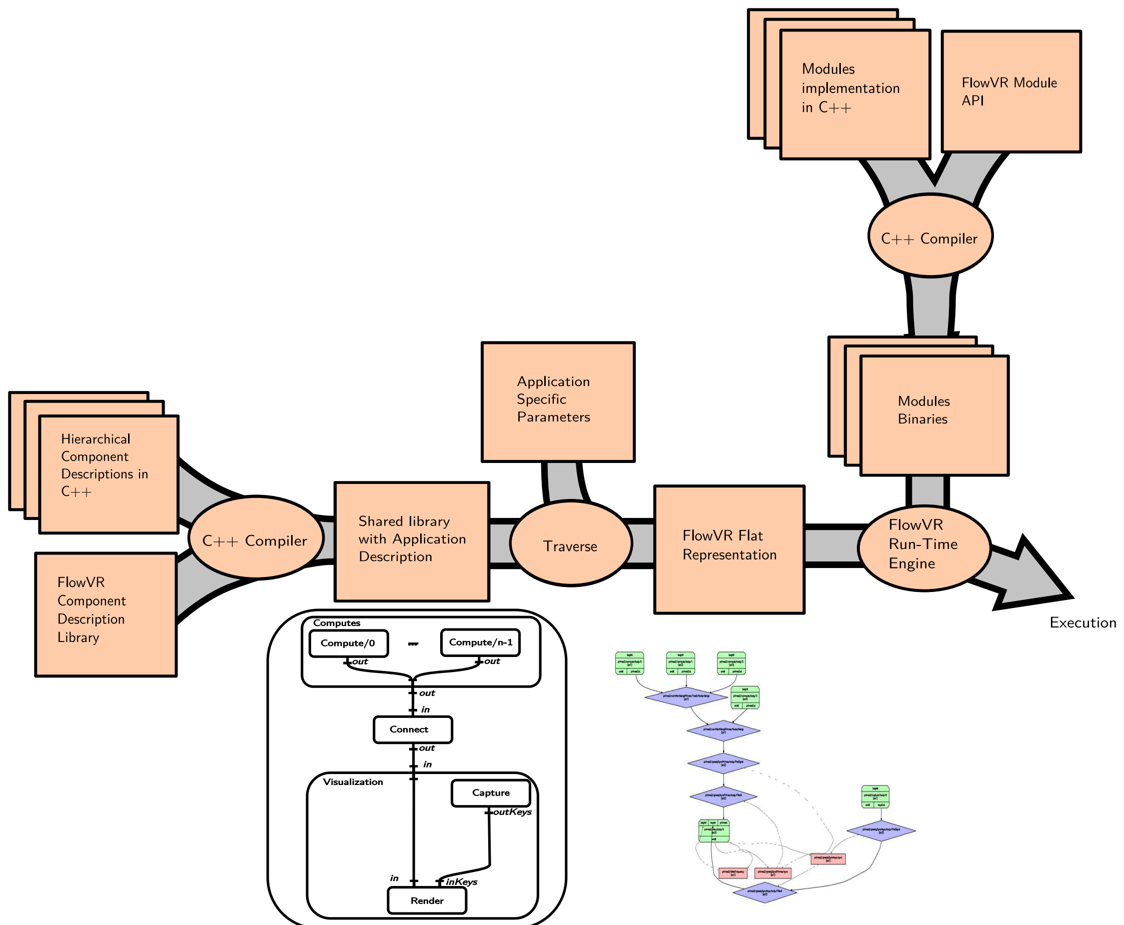


FIGURE 6.5 – Chemin de compilation d'une application FlowVR. De haut en bas, compilation des modules à partir du code C++. De gauche à droite, compilation du réseau FlowVR. Le *traverse* génère la description plate du réseau à partir d'une description hiérarchique contenue dans une libraire partagée.

Chapitre 7

Modification de la politique d'envoi des messages

Pour démontrer l'intérêt de la hiérarchie, nous montrons ici un exemple simple. A partir de cet exemple, nous allons implémenter différentes politiques de transfert de messages entre les composants. En assemblant des composants grâce à la hiérarchie, il est possible d'exprimer des schémas complexes.

L'exemple que nous prenons est une application graphique faisant partie de la suite FlowVR. Un ou plusieurs modules produisent des primitives graphiques (ces modules sont appelés *Viewer*) envoyées à un processus de rendu (appelé *ModuleRenderer*). Le composant de rendu peut être parallèle. Par exemple, il peut être exécuté sur plusieurs cartes graphiques dans le cadre d'un rendu parallèle dans un CAVE ou un casque stéréo. Les viewers ont un port de sortie appelé *scene*. Le renderer reçoit les primitives graphiques grâce à un port d'entrée appelé aussi *scene*.

La version la plus simple (Fig. 7.1) de l'application est de relier le viewer au renderer grâce à un canal de communication FIFO.

L'inconvénient de cette méthode est que l'exécution d'une itération du renderer démarre à la réception d'un nouveau message du viewer. Par conséquent, si l'utilisateur change le point de vue, le processus de rendu devra attendre de nouvelles primitives graphiques du viewer pour prendre en compte le nouveau point de vue. Ce comportement n'est donc pas approprié. Pour résoudre ce problème, nous allons désynchroniser les deux composants. Dans cette nouvelle version (Fig. 7.2), le composant de communication est capable de produire le dernier message du viewer sur demande du renderer. Ici, le renderer envoie à chaque fin d'itération un signal sur un port spécifique appelé *endIt*. Ce message est envoyé à un synchroniseur qui envoie un ordre à un filtre particulier (appelé *FilterIt*). Ce filtre retourne alors le dernier message du viewer au renderer. Dans ce schéma, la vitesse d'exécution du *ModuleRenderer* est totalement indépendante du viewer. Par conséquent, plus

rien ne le contraint. Le *ModuleRenderer* ira à sa fréquence maximale. Le groupe de composants *FilterIt, Synchronizer* qui échantillonnent le signal sont encapsulés grâce à la hiérarchie dans un composant appelé *ComSync*. Grâce aux templates C++, *ComSync* est paramétrable. Les templates utilisés lors de la création de *ComSync* donnent le type du filtre et du synchroniseur de cette communication.

Dans FlowVR, le langage de primitives graphiques est progressif [87]. En effet, certaines instructions comme l'allocation de ressources par exemple ne peuvent être rejouées ou jetées. Par exemple, si le viewer déclare une nouvelle ressource comme une texture. Il déclarera alors un identifiant unique associé à la texture. Si le renderer ne reçoit pas cette déclaration, il ne pourra pas alors utiliser cette texture dans les itérations suivantes. A l'inverse, si le message déclarant la texture est rejoué plusieurs fois par la communication d'échantillonnage, le renderer pourra détecter comme une incohérence le fait qu'un identifiant de texture est utilisé plusieurs fois. Pour résoudre ce problème, il est possible de proposer un schéma d'échantillonage plus complexe (Fig. 7.3). En effet, l'échantillonneur peut concaténer les derniers messages reçus au sein d'un message unique. Dans ce cas, la communication produira toujours des messages à la fréquence du viewer. Mais, si aucun nouveau message n'a été produit par le viewer entre deux itérations du renderer, la communication produira cette fois-ci un message vide. Le filtre *MergeIt* permet de concaténer les derniers messages au sein d'un seul message. Les templates C++ permettent ici de paramétrer le composant *ComSync* avec ce nouveau comportement.

Pour obtenir une scène complexe, nous pouvons utiliser en parallèle plusieurs viewers. Par exemple, le *viewerObj* produira les primitives graphiques représentant un objet statique décrit dans un fichier et le *viewerWorld* enverra la représentation d'un monde virtuel. Dans ce nouvel exemple (Fig. 7.4), les primitives des deux viewers doivent être concaténées au sein d'un unique message pour être lues par le renderer. De plus les messages doivent toujours respecter la politique d'échantillonage précédente pour obtenir un rendu cohérent. Nous utilisons ici le filtre *Merge* qui concatène les messages arrivant sur les différentes entrées. Le *RoutingNode* est un objet de FlowVR qui permet de dupliquer un message. Dans cette exemple, nous utilisons trois schémas classiques :

Com1ToN : un arbre de diffusion de messages ;

ComNTo1 : un arbre rassemblant des messages ;

PatternParallel : N instances du même composant en parallèle.

Comme pour la programmation par squelette, ces schémas peuvent être instanciés avec différents composants. On utilise les templates C++ pour réaliser ce paramétrage. Nous utilisons donc ici ces squelettes pour dupliquer le schéma d'échantillonage et en attribuer un par viewer. Cette communication est alors reliée à un composant qui rassemble les primitives graphiques pour le *ModuleRenderer*.

Le processus de rendu peut lui aussi être parallèle. En effet, dans le cas d'un périphérique de rendu stéréoscopique (casque de réalité virtuelle par exemple), le rendu est fait par deux processus synchronisés entre eux. Le dernier exemple (Fig. 7.5) permet de faire ce type de rendu parallèle. Dans cet exemple, nous ajoutons un nouveau module *Swaplock*, qui représente un thread dans le processus de rendu. Ce thread contrôle l'échange des tampons utilisés pour le rendu. Les tampons doivent être intervertis au même instant dans tous les processus de rendu pour obtenir un affichage cohérent sur les différents processus de rendu [91]. Pour réaliser cela, nous utilisons un nouveau filtre appelé *SignalAnd* qui a le rôle d'une barrière de synchronisation. En effet, ce filtre attend d'avoir un signal sur tous ses ports d'entrée pour produire un jeton en sortie. En complément, dans cet exemple, les primitives graphiques sont dupliquées pour être envoyées à chacun des processus de rendu. Nous pouvons supposer grâce aux *SwapLock* que les processus de rendu sont synchronisés, par conséquent, il est suffisant d'utiliser le signal *endIt* d'un seul composant *Renderer* pour contrôler le schéma d'échantillonage. Comme précédemment nous relions avec un simple lien le port *endIt* du composant *Renderer* au composant d'échantillonnage.

Ce dernier schéma complexe est utilisé régulièrement dans les applications utilisant un rendu graphique. Pour cette raison, au sein d'un composant composite appelé *Renderer*, nous avons encapsulé tous les éléments nécessaires au rendu distribué. Un utilisateur manipulera ce composant comme une boîte noire et ne connaîtra pas le détail de l'implémentation. La dernière application devient alors plus facile à décrire (Fig. 7.6.a)). La hiérarchie facilite donc la modularité de l'application en proposant une méthode simple pour factoriser le code. En outre, en utilisant ce composant, l'utilisateur bénéficie d'un contrôleur de placement automatique écrit pour ce composant (Fig. 7.6.b)).

7.1 Bilan

Une des motivations de notre modèle était l'encapsulation des schémas de communication au sein de composants. Ces schémas sont largement utilisés et ont un impact important sur les performances de l'application. Ce chapitre présentait des exemples d'utilisation de cette encapsulation. Notre implémentation utilise les templates C++ pour définir les patrons de conception.

Les communications vers les processus de rendu doivent être optimisées. En effet, ce schéma se retrouve dans l'ensemble des applications de réalité virtuelle ou de visualisation scientifique. Dans le cas du rendu de scènes complexes, un flux important de données est envoyé au processus de rendu. Nous avons donc choisi un exemple très utilisé pour démontrer l'interêt de notre modèle.

```
void execute()
{
  // Create components
   Component* viewer = addObject<Viewer>("viewer");
   Component* renderer = addObject<ModuleRenderer>("renderer");
   Component* comm = addObject<Connection>("com");

   // link ports
   link(viewer->getPort("scene"), comm->getPort("in"));
   link(comm->getPort("out"), renderer->getPort("scene"));
}
```

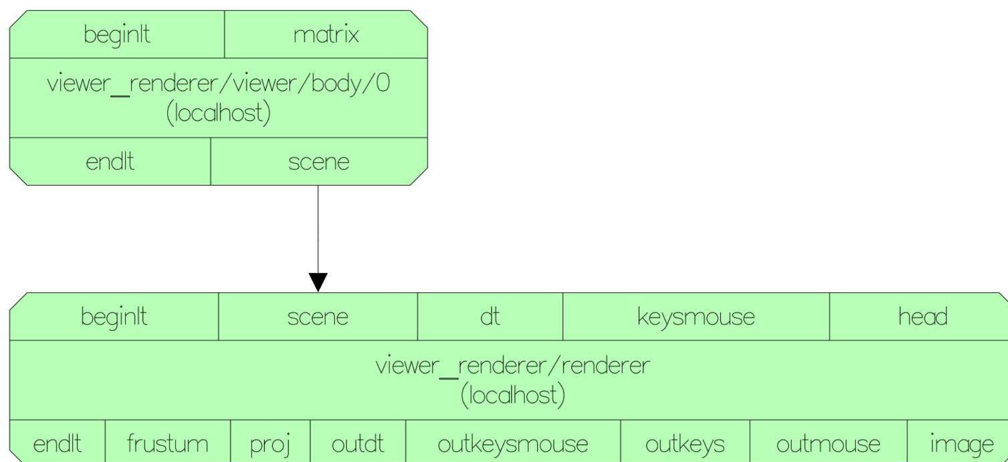


FIGURE 7.1 – a) Code C++ de l'exemple viewer-renderer avec politique FIFO. b) Réseau plat correspondant

```
void execute()
{
  // Create components
   Component* viewer = addObject<Viewer>("viewer");
   Component* renderer = addObject<ModuleRenderer>("renderer");
   Component* comm = addObject<ComSync<Connection,
                                ConnectionStamps,
                                Connection,
                      ConnectionStamps,
                                FilterIt,
                                Synchronizor> >("com");

   // link ports
   link(viewer->getPort("scene"), comm->getPort("in"));
   link(comm->getPort("out"), renderer->getPort("scene"));
   link(renderer->getPort("endIt"), comm->getPort("sync"));
}
```

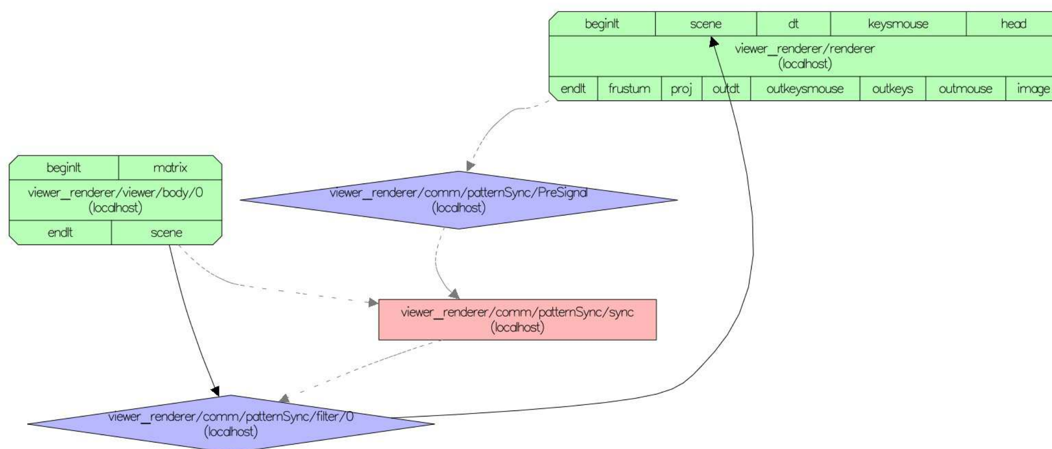


FIGURE 7.2 – a) Code C++ de l'exemple viewer-renderer, le signal du viewer est maintenant échantillionné par le couple *FilterIt, Synchronizor*. b) Réseau plat correspondant

```
void execute()
{
  // Create components
   Component* viewer = addObject<Viewer>("viewer");
   Component* renderer = addObject<ModuleRenderer>("renderer");
   Component* comm = addObject<ComSync<Connection,
                                       ConnectionStamps,
                                       Connection,
                                       ConnectionStamps,
                                       MergeIt,
                                       Synchronizor> >("com");

   // link ports
   link(viewer->getPort("scene"), comm->getPort("in"));
   link(comm->getPort("out"), renderer->getPort("scene"));
   link(renderer->getPort("endIt"), comm->getPort("sync"));
}
```

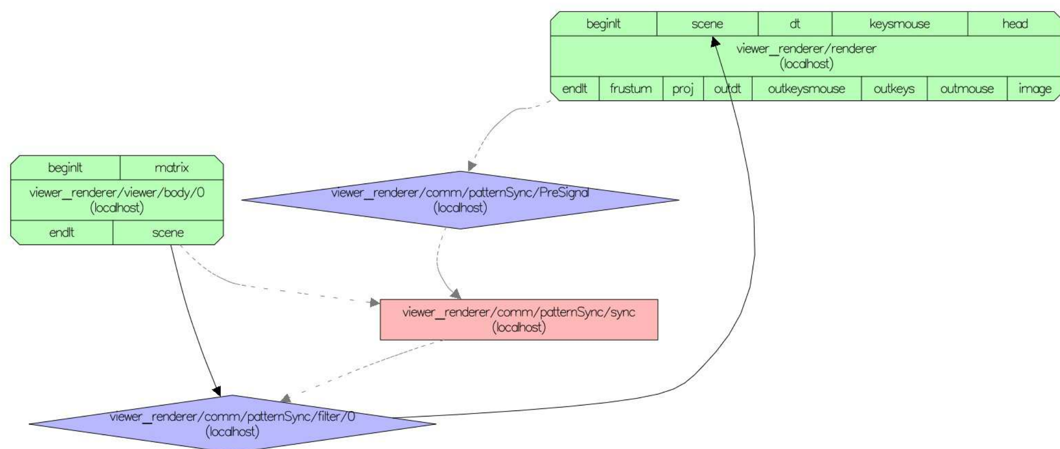


FIGURE 7.3 – a) Code C++ de l'exemple viewer-renderer, le signal du viewer est maintenant échantillionné par le couple *MergeIt, Synchronizor*. b) Réseau plat correspondant

```
void execute()
{
  // Create viewers
   Component* viewerObj = addObject<ViewerObj>("viewerobj");
   Component* viewerWorld = addObject<ViewerWorld>("viewerworld");

  // Create renderer
   Component* renderer = addObject<ModuleRenderer>("renderer");

  // Create sampling com
   Component* comm = addObject<PatternParallel<ComSync<
                              Connection,
                              ConnectionStamps,
                              Connection,
                              ConnectionStamps,
                              MergeIt,
                              Synchronizor> > >("com");

    // Create Merge data from 2 viewers
   Component* merge = addObject<ComNTo1<FilterMerge,
                               Connection> >("merge");

   // link ports
   link(viewerObj->getPort("scene"), comm->getPort("in"));
   link(viewerWorld->getPort("scene"), comm->getPort("in"));
   link(comm->getPort("out"), merge->getPort("in"));
   link(merge->getPort("out"), renderer->getPort("scene"));
   link(renderer->getPort("endIt"), comm->getPort("sync"));
}
```

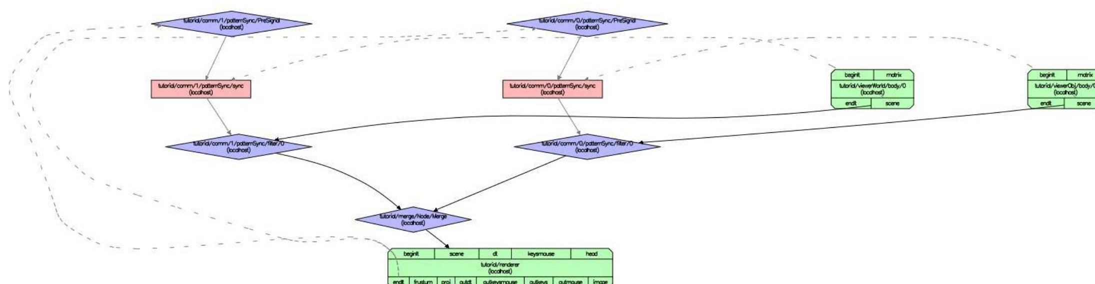


FIGURE 7.4 – a) Code C++ de l'exemple viewer-renderer avec plusieurs viewers. Le signal est échantillonné en parallèle puis rassemblé au sein d'un seul message. b) Réseau plat correspondant

```
void execute()
{
  // Create viewers
   Component* viewerObj = addObject< ViewerObj>("viewerobj");
   Component* viewerWorld = addObject< ViewerWorld>("viewerworld");
  // Create renderer
   Component* renderer = addObject<PatternParallel<Renderer> >
                          ("renderer");
   Component* swaplock = addObject<PatternParallel<SwapLock> >
                          ("swaplock");
  // Synchronization between swaplock
   Component* synchro = addObject<ComNTo1<FilterSignalAnd,
                          ConnectionStamps> >("synchro");
   Component* bcsignal = addObject<Com1ToN<RoutingNode,
                          ConnectionStamps> >("bcsignal");
  // Create sampling com
   Component* comm = addObject<PatternParallel<ComSync<
                          Connection,
                          ConnectionStamps,
                          Connection,
                          ConnectionStamps,
                          MergeIt,
                          Synchronizor> > >("com");
   // Create Merge data from 2 viewers
   Component* merge = addObject<ComNTo1<FilterMerge,
                           Connection> >("merge");
   // Broadcast Scene
   Component* bcast = addObject<Com1ToN<RoutingNode,
                          Connection> >("bcast");

   // link ports
   link(viewerObj->getPort("scene"), comm->getPort("in"));
   link(viewerWorld->getPort("scene"), comm->getPort("in"));
   link(comm->getPort("out"), merge->getPort("in"));
   link(merge->getPort("out"), bcast->getPort("in"));
   link(bcast->getPort("out"), renderer->getPort("scene"));
   link(renderer->getPort("endIt"), comm->getPort("sync"));
   link(swaplock->getPort("out"), synchro->getPort("in"));
   link(synchro->getPort("out"), bcsignal->getPort("in"));
   link(bcsignal->getPort("out"), swaplock->getPort("in"));
}
```

Figure 7.5 – Code C++ de l'exemple viewer-renderer avec plusieurs viewers et plusieurs renderers

```
void execute()
{
  // Create viewers
   Component* viewerObj = addObject<ViewerObj>("viewerobj");
   Component* viewerWorld = addObject<ViewerWorld>("viewerworld");

  // Create renderer
   Component* renderer = addObject<Renderer>("renderer");

   // link ports
   link(viewerObj->getPort("scene"), renderer->getPort("scene"));
   link(viewerWorld->getPort("scene"), renderer->getPort("scene"));
}
```

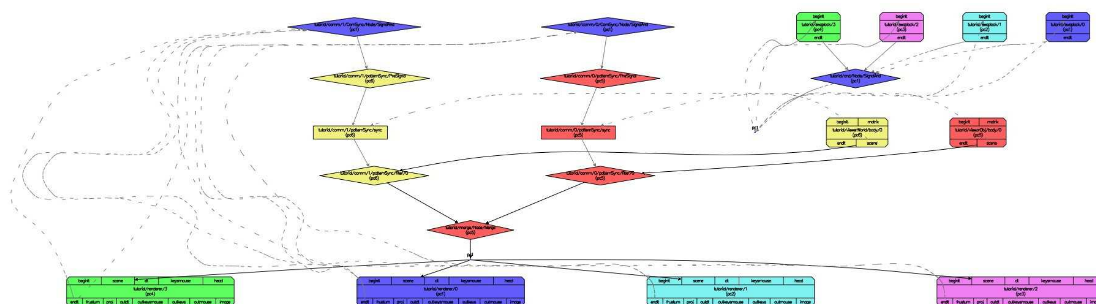


FIGURE 7.6 – a) Code C++ de l'exemple viewer-renderer avec plusieurs viewers. Le schéma générique pour le rendu de primitives graphiques est encapsulé dans le composant composite *Renderer*. b) Réseau plat correspondant. Les couleurs représentent le placement : chaque couleur correspond à une machine différente

Chapitre 8

Discussion

8.1 Introduction

Nous avons présenté le modèle hiérarchique dans les chapitres précédents. Dans ce chapitre, nous discutons de son utilisation. Depuis plus d'un an, le modèle est utilisé pour développer de nouvelles applications FlowVR. Dans cette partie, nous avons synthétisé les retours des utilisateurs du modèle et nos expériences personnelles.

Le chapitre est structuré en fonction des différentes questions que nous nous posons. L'étude de ces questions permettra de rendre le modèle plus efficace et plus adapté au développement des applications interactives.

8.2 Utilisation du modèle hiérarchique

Le modèle a été validé en développant la plateforme Grimage. Grimage est une application de réalité virtuelle permettant des interactions naturelles. Pour réaliser ces interactions, un modèle 3D de l'utilisateur est acquis grâce à un algorithme de reconstruction [63]. Ce modèle est envoyé à Sofa [89], un simulateur physique. La scène finale est rendue soit sur un mur d'image soit dans un casque de réalité virtuelle. Le démonstrateur utilisant le modèle hiérarchique a été présenté durant SIGGRAPH'09 à la Nouvelle Orléans. De plus dans le cadre du projet ANR Dalia, en collaboration avec une équipe du LABRI à Bordeaux, nous avons proposé le couplage de deux plateformes Grimage au sein de la même application. Là aussi, nous avons mis en place une démonstration présentée lors de VRST'08. Ces expériences nous ont permis de construire des prototypes assez robustes avec le modèle. Ces démonstrateurs étaient de qualité suffisante pour pouvoir être présentés lors de conférences ou manifestations. En 2008, nous avons présenté une version de l'application de réalité virtuelle lors de la fête de la science au Grand Palais à Paris.

Durant un week-end, des milliers de visiteurs ont testé l'application. De l'ensemble de ces expériences, nous avons pu en tirer des premières impressions vis-à-vis de l'utilisation du modèle.

La hiérarchie semble assez naturelle à utiliser. A haut niveau, le couplage des composites donne des schémas assez classiques. Il est fréquent de retrouver la structure que propose Shaw [14], le *Decoupled Simulation Model*, quand nous assemblons les composants de plus haut-niveau. Il est donc possible d'écrire un squelette simple de composites représentant l'ensemble de l'application.

Certaines communications collectives faisant partie de la distribution FlowVR comme les arbres de diffusion ou les schémas d'échantillonnage sont largement utilisées dans les applications. Il semblerait que le modèle permette d'encapsuler et de distribuer ces patrons de conception.

En contrepartie, notre implémentation demande d'écrire un code assez verbeux. Tous les assemblages de composants se font obligatoirement au sein d'un composite. La création d'un nouveau composant demande l'implémentation au minimum d'une nouvelle classe C++ avec 3 méthodes. Pour du prototypage, il est quelque fois un peu laborieux de construire cette classe.

Nous avons remarqué qu'il est difficile lors de l'apprentissage du modèle de comprendre la différence entre les liens et les connexions. En effet, les liens sont purement symboliques et ne représentent pas les transferts de données. Ils mettent en relation deux ports. Les connexions sont des composants qui représentent le média par lequel les données vont être transférées. L'ambiguïté entre les deux notions peut être une source d'erreur. Pour faciliter l'apprentissage du modèle, nous étudions un mécanisme de génération automatique des connexions. Si une connexion est omise, le système en proposera une automatiquement. Le développeur expert pourra toujours imposer un type de connexion particulier s'il considère que le choix de l'algorithme n'est pas optimal.

Certaines notions ont été ajoutées au fur et à mesure au modèle pour en faciliter l'utilisation. Ces extensions n'ont pas toutes été présentées dans les chapitres précédents. Nous avons introduit les paramètres par exemple. Chaque composant contient un ensemble d'éléments (Clés/Valeurs) qui représentent des paramètres associés au composant. Les contrôleurs peuvent durant leurs exécutions accéder à cette table. Comme pour l'héritage de la programmation par objet, nous considérons qu'un composant hérite des paramètres de son père dans la hiérachie. Ces outils ont été ajoutés au fil des expérimentations. Le modèle doit être facilement extensible afin de répondre aux besoins des développeurs.

Enfin, une limite importante vient de la notion de port des composites. Nous allons détailler en détail ce point dans la section 8.8. Le modèle doit évoluer pour simplifier le couplage de composants composites.

8.3 Une librairie C++ comme ADL

Nous avons choisi d'implémenter notre ADL pour les applications FlowVR sous forme d'une librairie C++. La plupart des modèles à composants utilisent le langage XML pour décrire leurs ADL. En effet, XML facilite la création d'un langage de description. En décrivant la grammaire à l'aide d'un fichier DTD (*Document Type Definition*), il est possible de définir totalement un ADL. Dans ce cas, les développeurs peuvent alors bénéficier d'une large palette d'outils XML (vérificateurs de syntaxe, transformateurs XSLT, éditeurs de fichier XML, etc ...) fonctionnant sur la plupart des systèmes. Un ADL décrit à l'aide de XML est alors très portable. Cela explique son succès au sein des communautés liées aux systèmes distribuées. Dans la communauté réalité virtuelle, InTml [34] est un middleware basé sur le XML et ses outils.

Néanmoins XML ne donne pas accès aux mêmes structures de contrôle que la plupart des langages de programmation. L'expression de certain schémas tels que les boucles n'est pas possible sauf par l'utilisation d'extensions exotiques ou de langages pouvant être mixés avec du XML tel que PHP. Les applications de grande taille conduisent alors à des descriptions complexes, peu modulaires, difficiles à débugger et à maintenir. En utilisant un langage de programmation tel que C++ nous bénéficions de nombreuses fonctionnalités que nous avons exploitées pour favoriser le passage à l'échelle :

Héritage spécialisation d'un composant sans redéfinir son interface.

Template passage des paramètres aux squelettes génériques. C'est ici la base de la programmation par squelette.

La STL utilisation des conteneurs et algorithmes haut niveau pour construire le réseau.

Les exceptions

Les outils de développement C++

Il est à noter que d'autres intergiciels comme VTK ou StreamIt utilisent aussi un langage de programmation C ou C++ pour définir leurs ADL. Enfin, notre implémentation permet de produire toute la structure hiérarchique sous forme d'un fichier XML. Cette fonctionnalité a été implémentée pour faciliter le débuggage de l'application en utilisant les outils de traitement XML. La figure 8.1 donne un exemple de visualisation d'un graphe produit par le modèle hiérarchique.

8.4 Visualisation de graphes hiérarchiques

La distribution FlowVR contient un visualisateur de graphe. Il permet de visualiser le graphe plat décrit en XML. Pour faciliter le développement avec notre

modèle, il est nécessaire d'étendre l'outil pour lui permettre de visualiser la hiérarchie de composant. Nous avons proposé un stage sur la visualisation hiérarchique de graphe de flot de données. Fabien Benureau, stagiaire durant l'été 2008, a produit une extension au visualisateur de graphe FlowVR qui permet d'utiliser la hiérarchie pour représenter plus clairement une application FlowVR (Fig. 8.1). Son état de l'art a mis en évidence que les solutions actuelles de représentation de graphes (Graphviz [92], Tulip [93] ou OverView [94]) ne permettent pas de représenter simplement des graphes de flots de données hiérarchiques. La solution envisagée actuellement est de considérer que chaque niveau de hiérarchie est un graphe indépendant à représenter. Nous pouvons donc calculer séparément le placement de tous les objets.

Les liens parents font la liaison entre les niveaux de hiérarchies. Ils doivent relier des ports présents sur deux niveaux de hiérarchies différents. La représentation de ces liens ne peut donc être calculée qu'après avoir obtenu tous les placements. Actuellement, il n'y a pas d'implémentation robuste de notre solution. Cela pourrait être le sujet d'un futur travail.

La méthode qui consiste à représenter tous les niveaux de hiérarchie pourrait utiliser l'algorithme de *Traverse*. Il est possible de construire un contrôleur qui appelerait des routines graphiques donnant le graphe d'un composant. En appelant le contrôleur sur l'ensemble des composants avec l'algorithme de *Traverse*, nous obtiendrons alors le graphe hiérarchique de l'application. Cela pourrait être un autre exemple d'utilisation des contrôleurs.

Nous avons besoin d'étudier plus en détail les méthodes de navigation dans la représentation du graphe hiérarchique. Pour l'instant, nous utilisons uniquement le mécanisme de gestion de vue d'OpenGL pour naviguer dans le graphe en zoomant ou en se déplaçant. Cette solution simple n'est peut-être pas suffisante dans le cas de graphe d'une dizaine de niveaux de hiérarchie et d'une centaine de composants.

Les méthodes de placement des objets dans le graphe que nous utilisons proviennent de la librairie Graphviz. Notre expérience nous a montré que ces algorithmes sont très sensibles aux changements dans le graphe. En effet, même si deux graphes sont très proches (1 connexion de différence par exemple), les résultats des placement des composants seront très différents. Dans le cas de débuggage d'un graphe, ce comportement peut-être très gênant. Si les changements effectués dans le graphe sont minimes, l'utilisateur s'attendra à ce que la disposition des objets ne varie que très légèrement. Ici, à chaque modification du réseau FlowVR, le développeur est obligé de se réhabituer à une nouvelle disposition des objets avant de détecter si sa modification est correcte. Le placement des objets doit donc être plus stable.

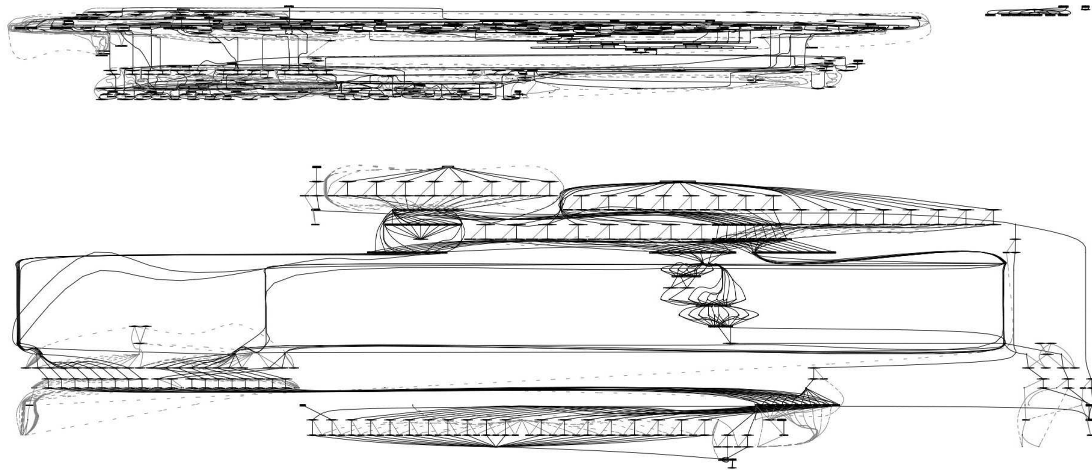


FIGURE 8.1 – a) Une application comportant une centaine de composants. La disposition du graphe n'utilise pas l'information de hiérarchie. b) La même application utilisant l'information de hiérarchie. Certaines structures logiques deviennent alors visibles.

8.5 Interface graphique de conception de réseau

Actuellement, il n'existe pas d'interface graphique haut niveau pour la conception de réseaux FlowVR. Nous avons remarqué que certains développeurs novices étaient réticents à utiliser le modèle pour cette raison. De plus, le modèle hiérarchique pousse à dessiner à la main la structure hiérarchique d'une application avant de la programmer. Il est assez frustrant de ne pas pouvoir programmer une application en la dessinant. L'utilisation d'une interface graphique pourrait faciliter l'utilisation du modèle. Notre expérience nous pousse à penser que l'interface convient uniquement au développement des composants de plus haut niveau. Les squelettes génériques distribués dans FlowVR utilisent les structures de contrôles du C++ (boucles, récursivité, conditionnels), il serait difficile de les implémenter avec une interface graphique.

Nous envisageons de développer une interface SWIG [95] de l'API actuel en C++. Cela permettrait d'utiliser d'autres langages de programmation, comme les langages de scripts comme Python et Ruby pour la conception de réseau FlowVR. Ces langages faciliteraient le prototypage d'applications FlowVR. Ils pourraient permettre aussi d'interfacer plus facilement des librairies existantes comme la gestion d'une interface graphique pour la modification de graphes.

Les langages modernes comme Ruby s'interfacent facilement avec du code C. Ils pourraient être une réponse au problème du prototypage. Il est possible à partir des

librairies de composants compilées du C++ de construire les interfaces pour que les composants soient vus comme des classes Ruby. Nous pourrons alors construire un composant composite dont le contrôleur *execute* appelle un interpréteur. En Ruby, l'appel en C à la méthode `ruby_init` instancie un interpréteur Ruby. Il serait possible de faire l'assemblage des composants depuis cet interpréteur et de prototyper l'application. Tester l'assemblage de composants serait plus simple. Car pour chaque essai, il ne serait pas nécessaire de créer une nouvelle classe C++. Par contre, cette méthode demande de mixer les langages. Ce mélange nécessite d'apprendre ces deux langages pour utiliser cette interface. Nous pensons qu'il est préférable d'unifier les langages de programmation dans FlowVR.

Pour l'instant, le développement d'une interface graphique n'est pas prioritaire dans FlowVR. Nous consacrons nos efforts à étendre le modèle pour faciliter l'écriture d'application de grande taille. Une interface graphique au dessus de notre modèle serait finalement assez similaire aux MVE utilisés en visualisation scientifique. Elle viserait le développement d'applications comportant une dizaine de composants.

8.6 Librairie de schémas de communication

La version 1.6 de FlowVR comporte déjà quelques schémas génériques :

Com1ToN ou ComNTo1 : ces composants sont des arbres de diffusion ou de réunion de données

PatternParallel : squelette instanciant N objets en parallèle

ComNtoN : communication entre deux ensembles de N objets en parallèle

ComSync : communication échantillonnant les données

Cette liste représente les squelettes indispensables pour la conception d'une application interactive comportant des composants parallèles. Comme ces composants sont la base de l'écriture de l'ensemble des applications, il est nécessaire que ces schémas soient stables et robustes. C'est pour cette raison, que pour l'instant nous ne supportons pas d'autres schémas.

Néanmoins, cette liste de schémas est appelée à s'élargir. Il est nécessaire d'identifier des schémas récurrents qui pourraient faire l'objet d'un squelette générique et répondant aux besoins des utilisateurs de FlowVR. C'est pour cette raison, qu'il est nécessaire en parallèle du développement de FlowVR, de s'impliquer dans le développement de grosses applications. Cela apporte l'expérience nécessaire pour détecter ces schémas récurrents.

8.7 Utilisation du modèle hiérarchique dans l'application Grimage

Notre implémentation du modèle hiérarchique est utilisée depuis plus d'un an en particulier sur la plateforme Grimage L'application Grimage propose des interactions naturelles. Un réseau de caméras filment un espace de reconstruction. Le système calcule en temps réel un modèle géométrique des objets dans la zone de reconstruction. Ce modèle est envoyé à une simulation physique. La simulation permet d'interagir par le biais du modèle avec des objets virtuels. Cette interaction est naturelle car l'utilisateur n'a pas besoin d'utiliser des périphériques spécifiques comme un gant, une manette, ou des marqueurs visuels pour interagir. Un démonstrateur de Grimage avait été présenté lors des *Emerging Technologies* de SIGGRAPH 2007 [62]. Ce démonstrateur était développé avec l'ancien langage de FlowVR en Perl qui n'utilisait pas la hiérarchie.

Durant l'été 2008, avec Benjamin Petit, nous avons réalisé une nouvelle implémentation de l'architecture de l'application Grimage. Nous avons utilisé le modèle hiérarchique. Cette implémentation a mené à un nouveau démonstrateur présenté lors de la conférence VRST 2008 en octobre à Bordeaux [7]. Ce travail a permis aussi la production d'un article présenté lors d'une conférence sur la télévision 3D [3].

Ce nouveau démonstrateur couple grâce à un lien réseau deux plateformes de reconstruction temps réel. Deux clusters indépendants, l'un administré par l'équipe du LABRI de Bordeaux et l'autre administré par notre équipe exécutent l'application. Grâce à notre modèle hiérarchique, ce démonstrateur a pu être implémenté entre avril et septembre 2008. En effet, la tâche principale a consisté à implémenter un réseau proposant les fonctionnalités de l'application présentée en 2007. Nous avons utilisé une des premières implémentations de notre modèle, il n'existait aucune librarie de composants *Composites*. Dans un premier temps, nous avons écrit une librairie de composants. Nous utilisons ajourd'hui ces composants dans nos applications.

L'application utilisant un seul réseau de caméra a été encapsulée au sein d'un composant *Composite*. L'ajout d'une deuxième plateforme a consisté à instancier un deuxième composant de ce type. La phase de compilation a permis d'adapter l'application aux deux clusters. Nous rappelons que les deux clusters étaient administrés séparément ; les clusters étaient donc hétérogènes. La phase de compilation a permis de résoudre cette difficulté. Au final, notre implémentation nous a permis une intégration simple de l'application sur les machines du LABRI. Cette integration a nécessité une petite semaine.

Après cette intégration, le travail a consisté à optimiser le composant réalisant les communications entre les deux plateformes.

Depuis janvier 2009, Thomas Dupeux et Benjamin Petit ont repris l'architecture de l'application afin d'y ajouter de nouvelles fonctionnalités. Ils ont intégré un casque de réalité virtuelle dans l'application. Le casque est un nouveau périphérique parallèle qui ajoute de nouveaux composants dans l'application. Il est couplé à une caméra infrarouge qui capture la position de l'utilisateur. Ces évolutions ont permis la présentation d'un nouveau prototype de l'application lors des *Emerging Technologies* de SIGGRAPH 2009 [8]. Le modèle à composants a permis la factorisation du code. Les développements ont pu utiliser les composants programmés pour les précédents démonstrateurs.

Tous ces travaux ont permis l'écriture d'un article de synthèse sur l'application Grimage [5]. Cet article vient d'être accepté et sera publié prochainement. Il présente le système et reprend une partie de la discussion sur les bénéfices du modèle hiérarchique.

8.8 Ports Composites

La hiérarchie de composants est une solution efficace pour masquer à haut-niveau le détail de l'implémentation. Un composite peut être vu comme une boîte noire encapsulant l'implémentation d'une fonctionnalité. Une application complexe exécutant des centaines de threads indépendants peut être vue grâce à la hiérarchie comme le couplage d'une dizaine de composants composites. La hiérarchie permet donc d'avoir un contrôle sur le degré de complexité présentée à l'utilisateur.

Comme pour tous modèles à composants, pour le couplage, il est nécessaire de connaître précisément la nature des messages échangés. La source et la destination le long d'une connexion doivent avoir connaissance de la structure de données échangée. Pour les modèles à composants habituels (EJB, Corba, etc...), il existe de multiples façons de partager cette connaissance. Ces structures peuvent être référencées dans un cahier des charges précis partagé entre les développeurs par exemple. Les composants peuvent aussi s'appuyer sur une librairie commune qui implémente les structures de données et les méthodes de sérialisation. Enfin, pour ajouter plus de dynamicité, il est possible que les messages comportent une description de la structure de données dans leurs entêtes. Dans le modèle hiérarchique, ces mêmes méthodes peuvent être utilisées entre les composants primitifs.

Il serait naturel d'appliquer les mêmes méthodes pour coupler des composants composites dans le modèle hiérarchique. Notre modèle dans l'état actuel ne le permet pas. En effet, les liens parents dans le modèle sont uniquement symboliques. Après dépliement de la hiérarchie, ces liens disparaissent car le graphe obtenu après l'algorithme du *traverse* est plat. Il est donc impossible dans la majorité des cas de décrire simplement le flux de données qui transite le long de ces liens. La figure 8.2 est un exemple de ce problème. Deux composants primitives dans cet exemple

sont reliés par des liens parents à un même port d'un composant composite. La sémantique de ce port est ambiguë. Il est possible d'arranger le flux des deux ports sources de façons très différentes. Nous pouvons par exemple, concaténer les deux messages dans un ordre précis au sein d'un nouveau message ou alterner la source à chaque message. Les composants reliés à ce port parent peuvent s'exécuter à des fréquences différentes, cela ajoutera alors de nouvelles contraintes temporelles pour mêler le flux.

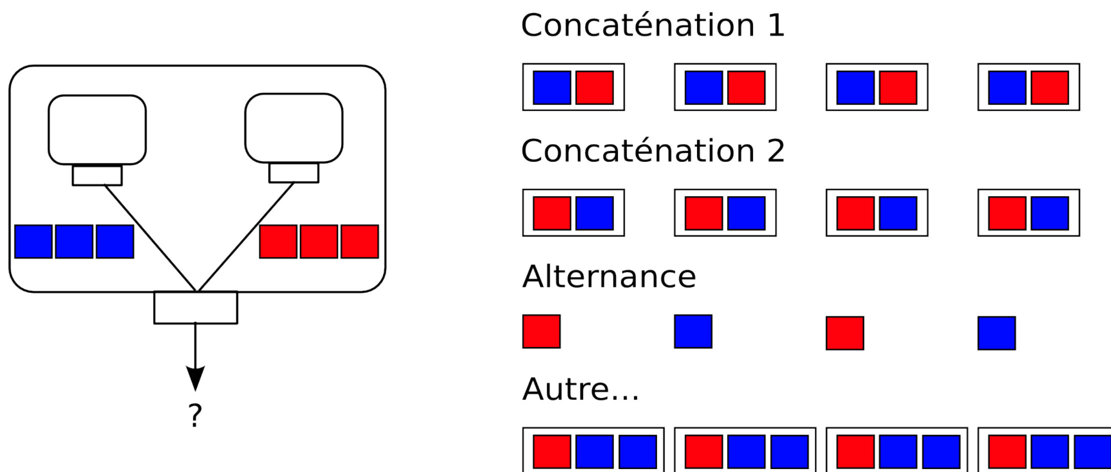


FIGURE 8.2 – Cas où deux composants primitifs sont reliés au même port parent. Les ports produisent des messages de natures différentes (symbolisées par les couleurs). Si le développeur n'exprime pas explicitement la manière de combiner les deux flux, il y a une ambiguïté sur la nature des messages produits par le composite. La partie droite du schéma présente quelques possibilités de couplage.

Dans l'exemple précédent, l'union des deux ports par les liens parents est purement symbolique. Il y a donc une ambiguïté au niveau de ce port parent sur la nature des messages produits. Cela rend particulièrement difficile le couplage de deux composants composites. Prenons un exemple très simple, où un composant composite a un port de sortie de type *Voiture*. La *Voiture* est composée d'une *Carrosserie* et d'un *Moteur*. Ceux-ci sont produits par deux composants composites différents ayant respectivement pour type de ports de sortie les types *Carrosserie* et *Moteur*. Si nous relions directement ces ports au port parent *Voiture*, du point de vue extérieur au composite, en raison de la hiérarchie, nous n'avons pas conscience de l'existence des deux composants primitifs. Au niveau du port *Voiture*, comme les liens parents sont purement symboliques, il n'y a pas réellement d'objets de type *Voiture* mais un flux de *Carrosseries* et de *Moteurs*. Pour se connecter à ce niveau, il est donc nécessaire de coupler la *Carrosserie* au *Moteur* pour obtenir la *Voiture*. Il faut donc exposer l'implémentation du composite pour savoir que la

Carrosserie et le *Moteur* proviennent de sources différentes. Cela va à l'encontre du modèle hiérarchique.

Nous voyons deux solutions simples. La première consiste à combiner la *Carrosserie* avec le *Moteur* directement dans le composite. Nous ajoutons donc un primitif qui construit l'objet *Voiture* avec la *Carrosserie* et le *Moteur*. Le résultat est envoyé sur le port parent. Dans ce cas, nous pouvons coupler le composant composite avec d'autres composants ayant un port d'entrée de type *Voiture*. L'inconvénient principal est que dans le contexte distribué, cela implique de rassembler les données sur un noeud central qui va générer la *Voiture*. Si nous sommes dans un contexte d'une communication NxM, il sera peut-être plus efficace de créer l'objet de type *Voiture* de façon distribuée directement dans une communication collective. Cette solution qui consiste à rassembler les deux flux au sein du composite, peut donc générer des goulots d'étranglement dans un contexte distribué. En pratique, une implémentation possible de cette méthode consisterait à indiquer au niveau des ports comment mixer ou diviser ces flux. Nous retrouvons cette idée dans l'implémentation ProActive de la norme Fractal. Dans cette implémentation, ils définissent les interfaces *multicast* et *gathercast* au niveau des ports composites [96]. Ces interfaces définissent les schémas de communication un port vers N ports.

L'autre solution, consiste simplement à ne pas présenter de port de type *Voiture*. Nous présentons directement au niveau du composite les ports *Carrosserie* et *Moteur*. Sans dévoiler l'implémentation du composant composite (les deux ports peuvent appartenir à un même composant primitifs ou à deux composants primitifs différents), permet le couplage avec des communications collectives performantes. Il est bien sûr dans ce cas nécessaire d'exprimer comment coupler la *Carrosserie* au *Moteur* pour obtenir la *Voiture*. L'inconvénient de cette méthode est qu'elle oblige à exposer à haut-niveau un très grand nombre de ports. Il n'est pas possible pour les raisons exprimées précédemment d'établir une hiérarchie sur les types de ports. C'est l'approche que nous utilisons actuellement pour nos applications. Nous obtenons des applications formées d'une dizaine de composites ayant chacun une dizaine de ports. La conséquence est que ce petit nombre de composants sont reliés par des centaines de liens. Le code devient alors assez complexe.

Dans l'état actuel, nous n'avons pas de solutions efficaces pour résoudre ce problème. De notre point de vue, la solution se situe entre ces deux propositions. Dans notre exemple, nous souhaitons à haut-niveau ne voir que le couplage se fassent grâce à un lien entre deux ports *Carrosserie*. L'assemblage des composants sera alors présenté de façon simple. Par contre, pour réaliser ce couplage, il est nécessaire de voir le détail des ports à la manière de la deuxième solution. Au final, il est nécessaire de présenter ces deux niveaux de complexité et donner les outils pour basculer de l'un vers l'autre. Une notion spéciale pour les ports composites

devrait être introduite dans le modèle. Le travail en cours est de comprendre comment enrichir le modèle hiérarchique pour faciliter le couplage des composants composites.

8.9 Bilan

Pour l'implémentation du modèle, nous avons priviligié l'utilisation du langage C++. Grâce au mécanisme de templates, nous avons pu construire des schémas génériques de composants. Notre librairie de schémas est encore restreinte. Les travaux futurs viseront à étendre cette librairie en identifiant des schémas récurrents dans les applications interactives.

Les premiers retours utilisateurs nous montrent que le modèle doit être accompagné d'outils performants qui faciliteront son utilisation. Pour faciliter le débuggage de l'application, nous avons travaillé sur un visualisateur de graphe. Le travail en cours intègre la visualisation de la hiérarchie dans cet outil.

Le langage est à l'origine destiné à décrire des applications de grandes tailles. Une conséquence est que pour de petites applications, l'utilisation du modèle peut être laborieuse. Cela implique aussi que le prototypage est encore difficile. Une interface vers les langages de script pourrait être une piste pour faciliter le prototypage rapide.

Le couplage de composants composites n'est pas simple. Notre expérience de développement de nos applications a montré que la hiérarchie permet de structurer une application entière. Nos applications, même si elles instancient des centaines de threads de calcul sont l'assemblage d'une dizaine de composants composites. En contrepartie, nous avons remarqué une explosion du nombre de ports au niveau de ces composites. Cela implique qu'à haut-niveau des centaines de liens relient les différents composants. Nous travaillons à introduire de nouveaux concepts pour réduire la complexité du couplage entre composants.

Troisième partie

Couplage de tâches itératives

Cette partie présente le travail en cours sur le couplage de tâches itératives. Ce travail est dans la continuité des résultats obtenus durant mon master. Les résultats n'ont pas encore été publiés. Un article résumant les chapitres 9, 10 et 11 a été soumis à IEEE VR 2010.

Chapitre 9

Introduction

9.1 Motivations

Nous avons vu que les problèmes d'ordonnancement dans le cadre des applications interactives sont souvent durs (cf. section 4.3.2). Sans prendre en compte l'échantillonnage, le problème de l'ordonnancement sur un simple pipeline est rapidement NP-dur dès que nous nous plaçons dans des situations non triviales [79]. Nous avons vu dans l'état de l'art, qu'une des spécificités des applications interactives est l'utilisation de l'échantillonnage. Ces applications interactives exécutent des tâches itératives fonctionnant à des fréquences différentes. Il est donc nécessaire d'utiliser des mécanismes que nous appelons échantillonnage qui adaptent la fréquence des flux d'une tâche à l'autre. L'échantillonnage apporte de nouvelles difficultés par rapport au problème de placement. Nous pouvons citer par exemple l'apparition d'artefacts liés à des tâches concurrentes fonctionnant à des fréquences différentes [46].

Nous allons dans cette partie caractériser certaines de ces difficultés. Nous ne savons pas actuellement dans quelle mesure l'échantillonnage modifie les résultats de complexité du problème de placement. Sans connaissance approfondie des mécanismes liés à l'échantillonnage, nous ne sommes pas capables d'énoncer une modélisation correcte des applications interactives. Par exemple, un algorithme d'optimisation de la latence est inutilisable sur des applications interactives s'il ne prend pas en compte le surcoût lié à l'échantillonnage.

Dans ce chapitre, nous allons faire un rappel sur la modélisation des applications interactives. Dans l'état de l'art, nous avons montré que cette modélisation est largement adoptée aussi bien par la communauté liée à la réalité virtuelle que celle liée à la visualisation scientifique. Nous allons énumérer les différentes sources d'échantillonnage dans une application interactive. Enfin, nous rassemblerons quelques résultats existants sur la latence créée par l'échantillonnage.

Dans la suite de cette partie III, nous étudierons un mécanisme simple d'échantillonnage. Il consiste à utiliser toujours la donnée la plus récente. Ce mécanisme permet d'échantillonner tous types de données. Il est largement utilisé au sein des applications interactives. Nous caractériserons la latence apportée par cet échantillonneur. A partir de cette caractérisation, nous proposerons des solutions pour contrôler la latence. Enfin dans un dernier chapitre, nous regarderons plus particulièrement le parallélisme de pipeline et la régulation pour les applications interactives.

9.2 Applications interactives

Une application interactive est un ensemble de processus itératifs. Chaque processus est une boucle infinie qui à chaque itération consomme les données sur ses ports d'entrée, puis exécute un calcul et enfin produit des messages sur les ports de sortie. Ces données sont échangées le long de connexions. Une connexion relie un port de sortie à un port d'entrée. Nous considérerons qu'il n'existe pas de cycles dans l'application. C'est à dire que l'application est modélisable par un graphe orienté sans cycle (DAG) noté $G = (V, E)$. V est l'ensemble des sommets et représente l'ensemble des tâches itératives. E est l'ensemble des arêtes et représente l'ensemble des connexions.

Il existe deux types de communications : les communications représentant les flux de données et celles représentant des événements. Les flux de données sont composés en général de messages de taille importante. Ces messages sont produits à une fréquence assez régulière. Il est possible de mesurer cette fréquence en Hz. Les événements sont souvent des messages de petite taille. Contrairement aux flux, il n'est pas possible de mesurer leur fréquence d'envoi. Par exemple, si nous prennons le cas des événements clavier, ces événements sont générés par l'utilisateur. Ces événements apparaissent souvent sous la forme de rafales. C'est à dire que la plupart du temps aucune touche n'est pressée. Durant les courtes périodes où des événements sont produits, la cadence d'écriture sera de l'ordre de plusieurs dizaines de touches par minute. D'autres événements générés par les composants eux-même peuvent être utilisés pour faire de la synchronisation. Par exemple, l'implantation d'une barrière de synchronisation peut se faire par l'échange d'évènements entre les tâches concernées, les processus peuvent s'échanger des événements à travers ces communications. Dans la suite, nous nous intéresserons exclusivement aux communications représentant les flux de données. Les hypothèses seront donc que le flux de message est régulier et que la taille des messages peut être importante (de 500 Ko à 5 Mo).

Les middlewares qui permettent de programmer les applications interactives utilisent en très large majorité ce modèle de flot de données. Les tâches ou boucles

sont implémentées par les utilisateurs. Ces tâches forment alors des composants qui peuvent être assemblés en réseau. Une API permet de décrire le DAG représentant ce réseau. Pour FlowVR, ce langage de description a été détaillé dans la partie précédente. Les tâches sont souvent programmées avec une API similaire à celle donnée dans la figure 9.1. La fonction `wait` est une fonction bloquante fournie par le middleware qui permet de synchroniser les itérations de la tâche. La méthode `get` permet de lire les messages disponibles sur les ports de la tâche. La méthode `put` permet de signaler au middleware que des messages sont disponibles sur les ports de sortie de la tâche. L'utilisateur doit alors définir la fonction `compute` qui décrit la tâche de calcul. Le modèle impose très peu de contraintes sur l'implémentation. Les tâches peuvent être exécutées sur tous types de support, par exemple des processus, des threads, des kernels CUDA, etc.

```
void Loop::start()
{
    while(wait())
    {
        get(....); // read messages
        compute();
        put(....); // write messages
    }
}
```

FIGURE 9.1 – Une implémentation en C++ des tâches itératives

9.3 Echantillonnage

Les tâches dans une application intéractive peuvent fonctionner à des fréquences différentes. Il est nécessaire d'introduire des mécanismes qui adaptent les fréquences des flux de données entre les composants. Ce sont ces mécanismes, appelés échantillonneurs, qui permettent de réaliser la communication entre les tâches fonctionnant de façon asynchrone.

Un exemple d'échantillonneur largement utilisé est le *double-buffering*. Un producteur de données écrit dans un tampon mémoire. En parallèle, les consommateurs de données peuvent lire dans un autre tampon mémoire. Quand le producteur de données a terminé d'écrire, une requête pour échanger le contenu des tampons est lancée. L'opération est réalisée dès qu'il est possible de faire l'échange en toute sécurité. Cette méthode permet de réaliser les lectures en même temps que l'écriture. Cela permet d'accélérer la fréquence des tâches.

Si le producteur de données est plus rapide que le consommateur, il peut écraser les anciennes données qui ne seront donc jamais utilisées. A l'inverse, si le consommateur est plus rapide, il réutilisera les données précédentes pour son calcul. Il n'aura pas à attendre la fin de l'écriture pour éxécuter sa tâche. Cette méthode est souvent utilisée pour faire l'interface entre un périphérique d'entrée et l'application ou entre la carte vidéo et le périphérique de rendu.

Des implémentations similaires de ce mécanisme peuvent être réalisées en utilisant des listes de tampons. Le consommateur utilise les N données les plus récentes et jette toutes les autres. Par rapport au simple *double-buffering*, ce mécanisme permet d'avoir une fenêtre comportant les dernières valeurs. Cet historique permet par exemple d'extrapoler les valeurs ou d'appliquer des filtres sur les données (filtre passe-bas en moyennant les valeurs par exemple). La méthode est utilisée pour interfacer les capteurs de positions [47] ou dans les applications virtuelles distribuées pour extrapoler la position des joueurs distants [97].

Cette notion d'échantillonnage se retrouve dans les middlewares de réalité virtuelle. Elle peut prendre des noms différents. Dans OpenMask, les échantillonneurs sont les *polateurs*. Dans FlowVR, ils sont appelés *connection greedy*. Ces deux middlewares proposent un mécanisme d'échantillonnage par défaut qui retourne toujours la donnée la plus récente. Ce mécanisme est équivalent au *double-buffering*.

9.4 Sources d'échantillonnage

L'échantillonnage est indispensable dans les applications interactives. Nous avons identifié trois propriétés des applications interactives qui induisent de l'échantillonnage.

Multimodalité

Une application interactive, spécifiquement en réalité virtuelle, utilise souvent plusieurs types de capteurs ou de périphériques de rendu. Chacun de ces objets fonctionne à sa fréquence propre. Par exemple, le rendu haptique fonctionne à plusieurs kHz. Un rendu visuel fonctionne entre 30 et 60Hz. Relier ces différents composantes va nécessiter d'échantillonner les flux.

Prenons l'exemple simple d'une application qui rend à l'écran un pointeur contrôlé par un bras haptique comme un *phantom*. Il sera nécessaire d'échantillonner le signal produit par le *phantom* pour pouvoir générer un rendu à 60 images par seconde.

Il est à noter que sur un même périphérique de rendu, plusieurs rendus de vitesses différentes peuvent être exécutés simultanément. Springer [46] présente une application où un rendu volumique lent est exécuté en même temps qu'un

rendu rapide d'objets simples. La réunion des deux rendus peut se faire grâce à des échantillonneurs.

Boucles à état

Certaines tâches itératives utilisent les résultats des itérations précédentes dans leur calcul. Nous les appelons boucles à état (en opposition aux boucles sans états - *stateless loops*). Les simulations physiques sont des exemples de boucles à état. Ces boucles génèrent à chaque itération la position des objets virtuels. Ces positions sont aussi une entrée du calcul. Deux itérations d'une de ces boucles sur les mêmes données en entrée, produiront des résultats différents. En effet, entre ces deux itérations, les positions des objets virtuels ont varié. Il est souvent nécessaire d'exécuter ces boucles à une fréquence élevée. Dans le cas d'une simulation physique par exemple, certains algorithmes produisent un résultat plus stable si la fréquence d'itération est élevée. La fréquence de ces boucles est souvent indépendante du reste de l'application. Par conséquent, des échantillonneurs sont utilisés autour des ports de ces tâches.

Sélection des messages dans un flux

Pour certaines applications, une latence trop importante dégrade les performances de l'utilisateur [98]. Dans certains cas, les capacités de calcul ne permettent pas à une application de traiter un flux de données dans sa globalité. Il est donc préférable de répondre rapidement même si le résultat est incomplet. Les rendus par niveaux de détail [99] sont une illustration de ce principe. Cet algorithme de rendu permet de réduire la résolution d'objets placés loin du point de vue de l'utilisateur. Ces objets au loin occuperont quelques pixels dans l'image finale. Il n'est pas nécessaire de les rendre avec une résolution maximale car au final le périphérique de rendu ne pourra pas restituer cette résolution. En diminuant la résolution de certains objets, nous pouvons représenter des scènes complexes en temps réel.

Plus spécifiquement, il existe une classe d'applications qui a pour but de traiter des flux importants d'informations. Ces flux d'informations viennent de différentes sources (données financières, logs d'applications, base de données, etc ...). L'objectif de ces applications est de traiter des flux tellement importants qu'ils ne pourraient être traités en intégralité en temps réel. Les logiciels dédiés à ce contexte comme Aurora [100], Boraleis [101] ou TelegraphCQ [102] sélectionnent les informations les plus pertinentes par un système de filtres.

Dans le cadre du streaming video, les capacités du réseau qui sert à la transmission peuvent être limitées. Là aussi, il est préférable d'ignorer certaines images ou de réduire la qualité de l'encodage pour diminuer la taille du flux. Les protocoles de qualité de service (QoS) comme RTCP [76] sont basés sur ce principe.

Dans le domaine de la réalité virtuelle, la plateforme de télépresence de Berkeley [103] est aussi une illustration de ce type d'échantillonnage. Comme pour le streaming video, une des problématiques du projet consiste à rechercher les méthodes de compression adéquates qui permettront d'envoyer à distance en temps réel des maillages représentant les utilisateurs.

9.5 *Synchronization Lag*

Mine [104] a proposé une première étude pour mesurer les sources de latence. Cette étude était accompagnée d'une liste des sources de latence dans une application interactive. Cette liste fait référence pour décrire la latence dans une application interactive. Les différentes sources identifiées par Mine sont :

- le temps d'itération des capteurs ;
- le temps de communication des données depuis le capteur ;
- le temps de calcul de l'application ;
- le temps de calcul du rendu ;
- le temps de rafraîchissement du dispositif de rendu.

Cette liste a été mis en place à partir de l'étude d'un système simple contenant un capteur de position et un dispositif de rendu. Mais il est possible de l'élargir au cas plus général.

Wloka [105] a proposé une étude plus générale de la caractérisation de la latence. Wloka a notamment détecté qu'un mécanisme d'échantillonnage simple apporte une latence importante. Il ajoute à la liste de Mine, une nouvelle source de latence qu'il nomme le *synchronization lag*. La définition de cette latence est le temps que la donnée attend dans un échantillonneur entre deux tâches avant d'être lue pour la première fois. Comme les tâches sont indépendantes, il est probable que durant l'execution, les données arrivent au niveau de l'échantillonneur juste après que l'échantillonneur ait produit une donnée. Dans ce cas, il est nécessaire d'attendre une itération complète de la tâche destination avant que les données soient lues.

A partir d'observations, il conjecture que cette latence est de l'ordre de la moitié du temps d'itération. Cette latence est donc fonction des fréquences des tâches de l'application. Cette constatation permet de comprendre pourquoi le temps de rafraîchissement du dispositif de rendu contribue à la latence générale.

Wloka a aussi remarqué que cette latence varie au cours du temps. La variation de la réactivité du système réduit de façon importante les performances de l'utilisateur [51]. Pour l'utilisation des casques de réalité virtuelle, Azuma a conclu [47] qu'il est préférable que la latence des données depuis le capteur soit constante pour qu'un système de prédiction de mouvement soit efficace. Si un échantillonneur est utilisé entre l'algorithme de prédiction et le capteur, il est préférable que la latence

générée par l'échantillonneur soit constante. En s'inspirant de l'étude de Mine, de nombreux protocoles de mesure de la latence ont été proposés [106, 75]. Par contre, il existe peu de résultats de mesure de la variation de la latence. Les descriptions de systèmes complets [107] proposent quelques résultats d'évaluation de la gigue (variation de la latence) totale du système.

Les développeurs ont imaginé diverses solutions pour réduire la latence dans les applications de réalité virtuelle [108]. Ces solutions cherchent principalement à réduire le temps de calcul des tâches. La latence a un impact particulièrement important sur l'utilisation des casques de réalité augmentée (ou casques *see-through*). Ces casques grâce à un système de caméras permettent de superposer les objets virtuels sur le monde réel. Pour rendre le monde réel, il n'est pas nécessaire d'utiliser un capteur de position, nous pouvons utiliser directement le flux d'images des caméras placées sur le casque. Il n'y a donc pas de latence perceptible dans le rendu du monde réel. Par contre, pour connaître la position des objets virtuels dans l'image finale, il est nécessaire d'utiliser un capteur pour connaître la position de la tête de l'utilisateur. Ensuite, il est nécessaire de réaliser des traitements sur ces positions pour rendre les objets à partir du point de vue correct. Si cette suite d'opérations apporte une latence importante, elle sera facilement détectée par l'utilisateur : les objets virtuels auront du retard par rapport au monde réel. Cette différence peut nuire à l'interactivité [98].

Il est primordial que les processus de visualisation soient performants pour permettre l'interactivité. La méthode classique d'échantillonnage pour le rendu graphique est le *double-buffering* que nous avons décrit précédemment. Un rendu graphique fonctionne en moyenne à 30 Hz. La conjecture de Wloka prédit une latence de l'ordre de la moitié du temps d'itération. Ici, la latence due au *double-buffering* serait de $16.5ms$. C'est une latence critique qui a été identifiée expérimentalement. Par conséquent, ce problème a été étudié et de nouvelles méthodes de rendu proposent de remplacer le *double-buffering* par de nouveaux échantillonneurs. Par exemple, la méthode du *frameless-rendering* [109] se base sur le rendu par lancer de rayons. Le lancer de rayons consiste pour chaque pixel à simuler le trajet de rayons lumineux partant du point de vue de l'utilisateur et passant par ce pixel. Une couleur est calculé en fonction des caractéristiques des objets rencontrés sur le chemin. La somme des couleurs des rayons donnera la couleur du pixel. Le *frameless-rendering* propose de calculer la couleur de chaque pixel de façon asynchrone. Par conséquent à chaque mise à jour du rendu, seul un sous-ensemble aléatoire de pixels ont été mis à jour. Cette méthode donne la sensation que la fréquence du rendu de l'image est égale à la fréquence de rafraîchissement du périphérique de rendu. L'*image-wrapping* [110] est une autre méthode de rendu qui remplace le *double-buffering* par des programmes qui extrapolent le contenu de la mémoire vidéo. Ces programmes simples peuvent être exécutés à la fréquence de

rafraîchissement. Ils mettent à jour l'image à chaque rafraîchissement du périphérique de rendu. Dans les deux cas, ces méthodes tentent d'accélérer la fréquence de rendu et limiter la latence due à l'échantillonneur.

Indirectement, les solutions réduisant le temps d'itération des tâches ont un impact sur le *synchronization lag*. En effet, la conjecture de Wloka indique que cette latence est proportionnelle au temps d'itération. Cependant, cette latence provient essentiellement du fait que le mécanisme d'échantillonnage est simple. Ces techniques ne modifient pas l'échantillonnage, et ne peuvent pas être considérées comme des solutions au problème du *synchronization lag*. En parallèle aux techniques précédentes, nous pouvons chercher à comprendre et caractériser cette latence due à la synchronization. Cela permettrait de trouver des solutions qui résoudraient la latence et la gigue dûe à l'échantillonneur.

9.6 Bilan

Les applications interactives sont modélisées par un graphe représentant les communications entre tâches. Dans les applications interactives, les tâches fonctionnent à des fréquences différentes. Cela introduit l'utilisation d'échantillonneurs. Ces objets permettent d'adapter la fréquence des flux de données entre les composants.

Il existe de nombreux mécanismes d'échantillonnage. Certains algorithmes d'échantillonnage ne peuvent être utilisés que sur un seul type de données. Nous regardons un mécanisme d'échantillonnage simple et générique. Nous étudions l'échantillonneur simple qui retourne toujours la donnée la plus récente. Ce schéma est très largement utilisé.

Des études expérimentales ont montré que ce mécanisme génère de la latence. Cette latence est appelée *synchronization lag*. Dans les chapitres suivants, nous nous proposons d'étudier formellement cette latence. Notre objectif est de décrire et comprendre cette latence pour pouvoir la combattre.

Chapitre 10

Etude de l'échantillonneur simple

10.1 Introduction

Nous avons vu que l'échantillonnage est nécessaire dans une application interactive. Il est fréquent d'échantillonner des données non-extrapolables. Par exemple, dans un cadre général, les images représentant une scène ne sont pas extrapolables. Pour réaliser l'extrapolation, il est courant d'utiliser un échantillonneur qui pour répondre à une requête envoie toujours la donnée la plus récente. Ce mécanisme est similaire au *double-buffering* qui a servi d'exemple dans le chapitre précédent. Une instruction *swaplock* permet d'échanger les tampons vidéos au niveau de la carte graphique. Cet échange est réalisé quand les données sont prêtes et l'opération peut être réalisée en toute sécurité. La donnée dans le tampon de lecture est donc la donnée la plus récente.

Wloka [105] dans son étude a montré que ce type d'échantillonnage produit une latence. Il désigne cette latence comme le *synchronization lag*. Son étude porte sur une description plus large de toutes les formes de latence dans une application parallèle. La caractérisation du *synchronization lag* est empirique. Il estime que la latence moyenne induite par ce mécanisme vaut $\frac{2}{f}$ où f est la fréquence de la tâche destination. Il remarque que cette latence varie au cours du temps sans quantifier cette variation. En analysant la latence moyenne, il conclue que cette source de latence contribue à hauteur de 50% à la latence totale de l'application.

Le couplage casque de réalité virtuelle et capteur de position est critique. En effet, un mauvais couplage est une cause de désagrément important dans l'utilisation d'une application de réalité virtuelle. La présence de gigue (variation de la latence) lors du couplage d'un capteur avec un périphérique de rendu gène la prédiction du mouvement [47]. A la vue des conclusions de Wloka, nous pouvons considérer que le *synchronization lag* peut contribuer en partie à cette gigue. Dans [106], cette latence est mesurée pour deux fréquences 60Hz et 20Hz. Cette fréquence est

celle d'un capteur de position. La conclusion de l'expérience montre que dans ce cas précis la latence due à l'échantillonnage est plus importante que celle due aux communications réseaux (20-25 ms contre 15-20 ms).

Cette latence induite par ce mécanisme d'échantillonnage simple est présent dans de nombreuses applications interactives. Nous proposons dans ce chapitre d'étudier en particulier cette latence et de la caractériser. Nous allons étudier une application comportant deux tâches itératives. Ces deux tâches sont reliées par un échantillonneur simple muni des règles suivantes :

- Le taille de la file de messages de l'échantillonneur est 1
- Sur réception d'un message de la source, l'échantillonneur remplace le message dans la queue par le nouveau message
- Sur demande de la destination, transférer une copie du message à la destination.

Le message dans la queue peut dont être retransmis plusieurs fois au destinataire. Par soucis de performance dans le cas où cela implique un transfert ou une copie de données, le message retransmis sera un simple identifiant prévenant le récepteur qu'il doit réutiliser le dernier message reçu.

L'étude consiste à étudier et caractériser q_n le temps d'attente du n^{ieme} message lu. Nous ne considérons donc pas les messages jetés par l'échantillonneur et qui ne seront jamais lus.

10.2 Etude de q_n

Nous allons caractériser q_n, le temps d'attente du n^{ieme} message dans un cas simple. Nous supposons que les deux tâches ont une période d'itération fixe. Nous notons τ_s et τ_d, les périodes d'itération de la source et de la destination. Nous considérons qu'il n'y a pas de coûts de communication. C'est à dire que les messages produits arrivent immédiatement au niveau de l'échantillonneur. La lecture est elle aussi immédiate. Ces hypothèses ont été choisies pour éliminer toutes formes de latences extérieures à l'échantillonnage. La valeur que nous calculons, q_n, représente la latence induite par l'échantillonneur simple entre deux tâches fonctionnant à des fréquences différentes. Il est possible avec ces hypothèses de trouver une relation de récurrence qui définit q_n par rapport à q_{n-1}.

Nous appelons dans la suite `Source`, la tâche qui produit des données, sa période est τ_s. Nous appelons `Destination`, la tâche qui lit des données et sa période est τ_d.

La `Source` et la `Destination` ont des fréquences différentes. Certains messages peuvent donc être lus plusieurs fois par la `Destination` ou bien n'être jamais lus. Nous cherchons ici à mesurer le temps d'attente du message lu durant la n^{ieme} itération de la `Destination`. Il existe m tel que ce message a été généré durant

la m^{ieme} itération de la `Source`. Pour lever les ambiguïtés, nous utilisons l'indice n quand l'énumération se fait selon la fréquence de la `Destination` et l'indice m quand l'énumération se fait selon la fréquence de la `Source`.

Theorème 10.2.1 *Si les périodes τ_s et τ_d sont constantes, alors pour tout n, nous avons*

$$q_n = (q_{n-1} + \tau_d) \mod \tau_s$$

Preuve Nous allons énumérer l'ensemble des variables utiles à la démonstration :

d_n la date de lecture du n^{ieme} message, c'est à dire la date où le message est transféré à la destination. Si un même message est lu deux fois, il existera un entier n tel que d_n est la date de sa première lecture et d_{n+1} la date de sa seconde lecture.

s_m la date d'insertion dans la queue de l'échantillonneur du m^{ieme} message.

$\bar{s}_n$ est la sous-suite de s_m qui correspond aux dates d'insertion dans la queue des messages qui au final seront lus. Cette suite est numérotée selon la fréquence de la `Destination`.

k_n est le nombre de messages qui seront insérés dans la queue de l'échantillonneur entre la date $\bar{s_n}$ et $\bar{s_{n+1}}$. Cela correspond au nombre de messages générés par la `Source` entre l'itération n et $n+1$ de la `Destination`.

Entre l'itération n et $n+1$ de la `Destination`, la source a effectué k_n itérations.

$$\bar{s}_{n+1} = \bar{s}_n + k_n \tau_s$$

A chaque itération de la `Destination`, un message est lu. Nous en déduisons la relation : $d_{n+1} = d_n + \tau_d$. Nous allons maintenant exprimer q_n. D'après la définition de la sous-suite $\bar{s}_n$, ce message a été créé à la date $\bar{s}_n$. L'attente du n^{ieme} message est donc le temps entre l'insertion du message dans l'échantillonneur à la date $\bar{s}_n$ et sa lecture à la date d_n. Ce qui se traduit par :

$$q_n = d_n - \bar{s}_n$$

Nous allons établir la relation entre q_n et q_{n+1}. Nous exprimons le temps d'attente du message $n+1$. En reprenant la relation précédente, $q_{n+1} = d_{n+1} - \bar{s}_{n+1}$. Nous remplaçons d_{n+1} et $\bar{s}_{n+1}$ par les relations que nous venons d'exprimer :

$$\begin{aligned} q_{n+1} &= (d_n + \tau_d) - (\bar{s}_n + k_n \tau_s) \\ q_{n+1} &= q_n + \tau_d - k_n \tau_s \end{aligned}$$

Il n'est pas possible que q_{n+1} soit supérieur à τ_s. En effet, si l'attente est supérieure à τ_s, la `Source` aura fait durant ce temps au moins une itération complète. Par conséquent, le message dans l'échantillonneur aura été remplacé par un message plus récent. Cette remarque permet de calculer k_n. k_n est l'entier tel que $q_n + \tau_d - k_n\tau_s$ est compris dans l'intervalle $[0, \tau_s[$. Cette opération est la définition de l'opération modulo. Par conséquent, nous avons la relation :

$$\forall n, q_{n+1} = (q_n + \tau_d) \mod \tau_s$$

10.3 Comparaison avec les générateurs pseudo-aléatoires

Les générateurs congruentiels linéaires sont une méthode pour obtenir un générateur pseudo-aléatoire [111]. Ils se basent sur la formule suivante :

$$X_{n+1} = (a.X_n + c)\ mod\ m \text{ où } a, c, m \text{ sont des entiers.}$$

Ce générateur simule une distribution uniforme sur $[0, m]$. Les paramètres a, c, m et X_0 influencent la qualité de la simulation. Cette méthode est assez populaire, car simple à mettre en oeuvre.

La formule que nous avons énoncée précédemment liant q_n et q_{n+1} correspond à la formule de ce générateur pseudo-aléatoire. Ici, les paramètres sont des réels imposés par le contexte d'exécution : $a = 1, c = \tau_d$ et $m = \tau_s$.

Nous allons vérifier statistiquement que la distribution des temps d'attente correspond réellement à celle de la loi uniforme. Nous utilisons les tests d'adéquation pour vérifer si les distributions sont générées par la même loi. Il existe de nombreux tests d'adéquation. Le test le plus connu s'appelle le test du χ^2. Ce test est simple à utiliser pour des variables aléatoires ayant un nombre fini de valeurs. Par exemple, ce test permet de vérifier qu'un dé à 6 faces réalise des tirages uniformes.

q_n est continu. Même si le test du χ^2 peut s'adapter aux variables continues, nous préférons utiliser le test de Kolmogorov-Smirnov. Ce test est plus adapté au cas des variables continues. Le résultat de ce test est une probabilité appelée p-valeur. Il évalue la probabilité que les deux distributions soient générées par la même loi. Par exemple, une probabilité de 1 signifie qu'il est certain que les deux distributions soient générées par la même loi.

Dans le cas présent, nous comparons avec la distribution de la loi uniforme. L'hypothèse à vérifier est que *le temps de queue est donné par une loi uniforme.*

En utilisant l'expression récursive (10.2), nous simulons les distributions de temps d'attente. Nous choisissons τ_d égale à $43ms$ $(23Hz)$. Nous faisons varier la fréquence de la source qui correspond à $\frac{1}{\tau_s}$ dans l'intervalle $[5Hz, 130Hz]$. Nous

effectuons le test d'adéquation pour chaque distribution obtenue par la simulation. Les résultats sont donnés par la figure 10.1.

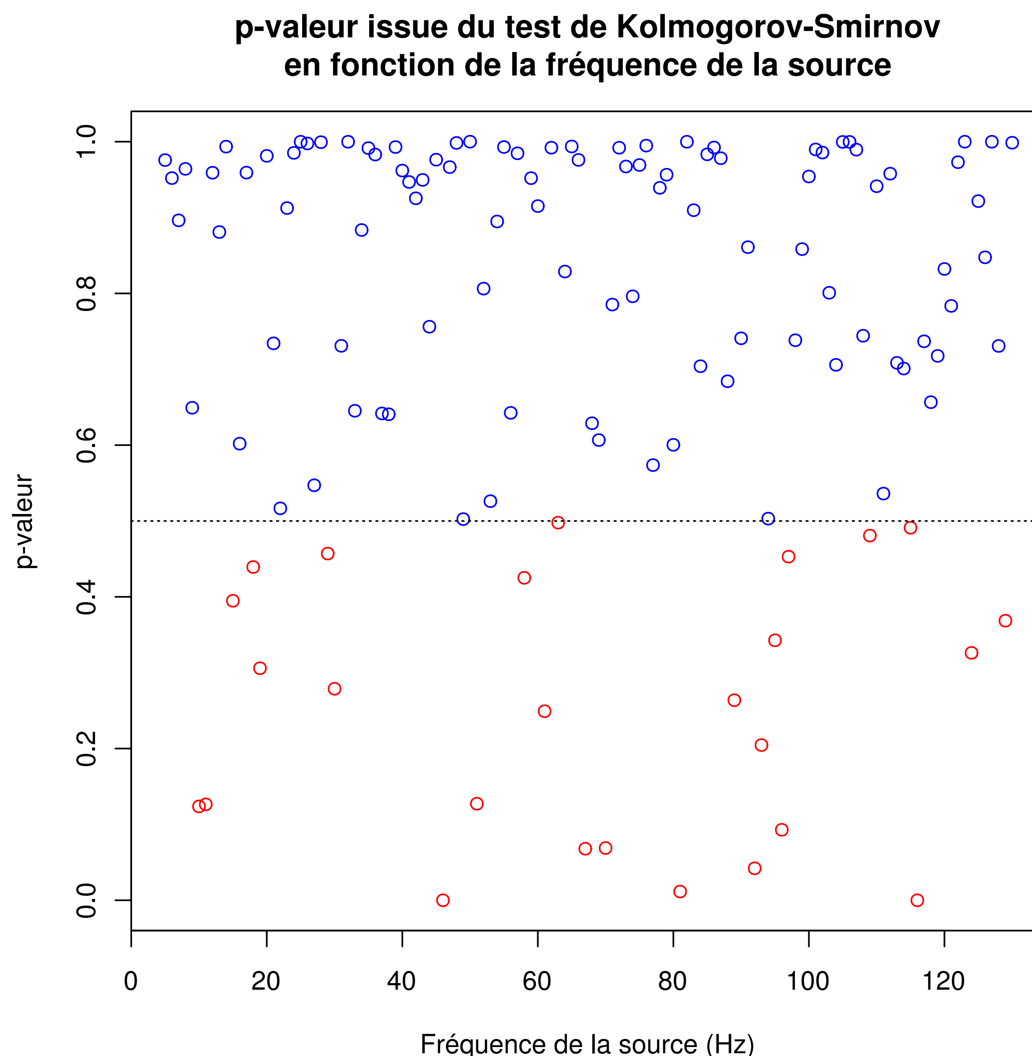


FIGURE 10.1 – Test de Kolmogorov-Smirnov vérifiant que la distribution des temps d'attente dans la queue est en adéquation avec la distribution générée par la loi uniforme. Pour chaque fréquence, nous calculons la p-valeur du test. La ligne en pointillés représente une probabilité de 0.5 au test. On accepte l'hypothèse pour les points ayant une probabilité supérieure à 0.5 et on rejette pour les autres. En bleu sont représentés les points où l'hypothèse est valide et en rouge les points où l'hypothèse est rejetée. La destination fonctionne à 23Hz.

Les valeurs nous indiquent que pour la plupart des fréquences, l'hypothèse est

valide. Pour de nombreuses valeurs, la probabilité que l'hypothèse soit vraie est même supérieure à 0,8. Ces résultats dans l'ensemble semblent confirmer l'interprétation que le temps d'attente des messages dans l'échantillonneur est donné par un générateur pseudo-aléatoire simulant une loi uniforme.

10.4 Moyenne et variance du temps d'attente

Sur les points où le test de Kolmogorov-Smirnov est valide, nous faisons l'hypothèse que le temps d'attente q_n est donnée par une loi uniforme sur l'intervalle $[0, \tau_s]$.

En partant de cette hypothèse, nous pouvons exprimer le temps d'attente moyen (Q) et la variance (V) du temps d'attente. Ces valeurs sont celles de la loi uniforme sur $[0, \tau_s]$

$$Q = \mathrm{E}[q_n] = \frac{\tau_s}{2} \tag{10.1}$$

$$V = \mathrm{V}[q_n] = \frac{\tau_s^2}{12} \tag{10.2}$$

Il est intéressant de calculer le coefficient de variation CV. Cette valeur sans unité normalise l'écart-type par rapport à la moyenne. Il permet de quantifier la dispersion des valeurs indépendamment de leur échelle. Son expression est :

$$CV = \frac{\sqrt{V}}{Q} = \frac{1}{\sqrt{3}} \simeq 0.58$$

Ce coefficient est constant. Nous pouvons proposer une interprétation. L'espérance correspond en pratique au temps d'attente moyen. L'écart-type correspond à la gigue. Si τ_s est extrêmement petit, nous avons d'après les expressions précédentes que Q et V seront également très petits. Les phénomènes de latence et de gigue ne seront plus détectables par un observateur humain. Cependant, en se ramenant à l'échelle de ces valeurs, la distribution q_n sera toute autant dispersée. Pour un algorithme fonctionnant à haute fréquence, la dispersion sera tout autant importante. Le problème ne sera pas résolu en diminuant τ_s. Certains algorithmes détecteront toujours la gigue même si nos sens n'en seront plus capables. Il est donc nécessaire d'étudier des mécanismes permettant de supprimer cette gigue.

Cette dispersion est importante car elle modifie le signal. Si le coefficient CV tend vers 0, alors la gigue est négligeable par rapport au temps d'attente. Nous pouvons considérer que tous les messages attendent une durée constante dans l'échantillonneur. Le signal représenté par ces messages est translaté dans le temps d'une valeur constante. Il n'y a pas de déformation. Si CV est non nul, l'écart type, donc la gigue, n'est plus négligeable. L'échantillonneur décalera dans le temps les

messages d'une durée non constante et aléatoire. Le signal est alors déformé. En pratique, cette déformation peut être interprétée comme du bruit.

10.5 Vérification expérimentale

10.5.1 Dispositif expérimental

Nous proposons de vérifier expérimentalement les résultats précédents. Nous avons développé grâce à FlowVR une application vérifiant les hypothèses de l'étude. En utilisant le modèle hiérarchique, l'application servant d'expérience est donnée par la figure 10.2. L'application consiste à relier deux modules `Source` et `Destination` par un échantillonneur. Nous contraignons la fréquence du module `Source` par un synchroniseur. Ce synchroniseur est un objet FlowVR qui permet de produire périodiquement un signal d'activation. La fréquence du signal est paramétrable. La fréquence de la source ne pourra pas dépasser la fréquence du synchroniseur. Nous pouvons fixer la charge des modules. Nous choisissons la charge du module `Source` pour qu'il puisse atteindre une fréquence maximale de 150 Hz. Le module `Destination` sera plus lourd et aura une fréquence maximale comprise entre 25Hz et 30 Hz. Les charges de ces modules seront constantes.

Nous nous plaçons dans le cadre simple utilisé dans la démonstration. Comme nous l'avons signalé en introduction du résultat, ces conditions permettent d'isoler l'impact de l'échantillonneur des autres facteurs de latence (réseaux, perturbation, etc.). Dans notre application, ces hypothèses se traduisent par les conditions suivantes :

- `Source` produit des messages de taille négligeable (quelques octets).
- `Source` et `Destination` sont fixés sur deux coeurs différents de la même machine. Les filtres et synchroniseurs FlowVR sont sur un troisième coeur. Ces trois coeurs ont un accès uniforme à la même mémoire.
- Il n'y a pas d'autres processus mis à part ceux du système d'exploitation s'exécutant sur la machine utilisée pour l'expérience.

Le démon FlowVR qui réalise la communication entre les processus utilise une mémoire partagée. Les messages sont stockés dans cette zone de mémoire partagée par tous les processus FlowVR. Les pointeurs vers cette zone mémoire sont échangés entre les différents processus. Cette solution permet d'éviter les copies de messages lors des communications entre processus. Le temps de communication sera celui de l'échange d'un pointeur entre deux processus.

Grâce à ces 3 conditions, nous pouvons reproduire les hypothèse du théorème 10.2.1.

- Les temps de communication sont négligeables vis à vis des temps d'itération des tâches.

– Les périodes d'itération τ_s et τ_d sont constantes.

Voici les paramètres de l'expérience pouvant être modifiés :

f_{source} Fréquence du synchroniseur fournissant le signal d'activation à `Source`. Cette fréquence est la fréquence de la `Source`.

T_{xp} Durée de l'expérience. Dans la suite, nous prendrons 2 minutes pour chaque exécution.

La machine utilisée est la machine `idkoiff` du Laboratoire d'Informatique de Grenoble (LIG). Les processeurs sont des *Dual Core AMD Opteron Processor 875*. Le système d'exploitation est linux.

10.5.2 Validation de la loi sur q_n

Nous souhaitons vérifier le résultat du théorème 10.2.1 expérimentalement. Nous mesurons avec le dispositif décrit à la section 10.5.1 la valeur $(q_{n+1} - q_n) \mod \tau_s$. Nous présentons ici uniquement les résultats utilisant les paramètres de l'expérience précédente : la `Source` fonctionne à 40Hz et la destination à 27Hz.

Le résultat est donné par l'histogramme de la figure 10.3. La ligne verticale rouge correspond au résultat théorique $(\tau_d \mod \tau_s)$. Nous mesurons grâce à l'expérience que $(q_{n+1} - q_n) \mod \tau_s = 17ms = \tau_d \mod \tau_s$. Les résultats expérimentaux sont ici en accord avec le résultat théorique. La moyenne correspond à la valeur attendue. La variance est faible.

En conclusion, q_{n+1} est relié par une relation déterministe à q_n. Cette relation est donnée par le théorème 10.2.1.

10.5.3 Validation de la loi sur la moyenne et la variance

En supposons que q_n est donnée par une loi uniforme, nous pouvons exprimer la moyenne et la variance de q_n. Les expressions de la moyenne et de la variance sont données par (10.1) et (10.2). Nous proposons de retrouver ces résultats expérimentalement. Nous utilisons toujours le dispositif présenté précedemment.

La figure 10.4 compare les valeurs mesurées de variance et de moyenne aux valeurs théoriques.

10.6 Bilan

Dans ce chapitre, nous avons procédé à une étude théorique du *synchronization lag*. Nous avons montré qu'une expression de récurrence définit les temps d'attente dans la queue de l'échantillonneur. Les résultats obtenus confirment la conjecture de Wloka. La latence est proportionnelle au temps d'itération des tâches. Nous avons procédé ensuite à une validation expérimentale du résultat.

```
void Two::execute()
{
    Component* first = addObject<MetaModuleMatrix>("1");

    Component* syncmax = addObject<SyncMaxFrequency>("sync");
    Component* cBegin = addObject<ConnectionStamps>("cBegin");
    Component* cEnd = addObject<ConnectionStamps>("cEnd");
    link(first->getPort("endIt"), cEnd->getPort("in"));
    link(cEnd->getPort("out"), syncmax->getPort("endIt"));
    link(syncmax->getPort("out"), cBegin->getPort("in"));
    link(cBegin->getPort("out"), first->getPort("beginIt"));

    Component* second = addObject<MetaModuleMatrix>("2");

    Component* connec = addObject<Sampler>("connec");

    link(first->getPort("out"), connec->getPort("in"));
    link(connec->getPort("out"), second->getPort("in"));
    link(second->getPort("endIt"), connec->getPort("signal"));
}
```

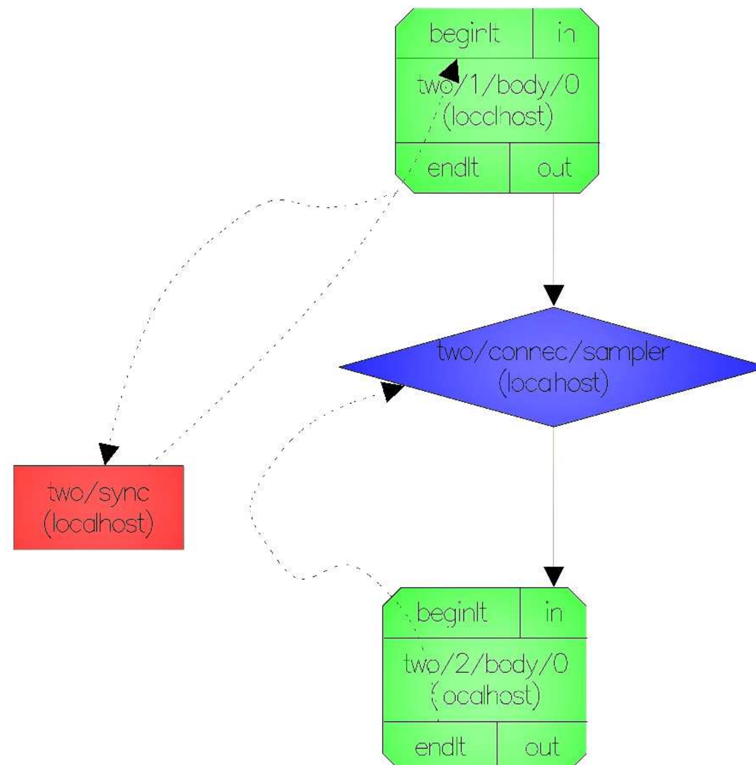


FIGURE 10.2 – a) Description de l'expérience avec FlowVR. Les MetaModuleMatrix sont des composants faisant des multiplications de matrices carrées de taille N. N est un paramètre qui permet de sélectionner la charge du composant. b) Réseau final

Nous avons montré que la gigue est proportionnelle au temps d'itération des tâches. La gigue aura pour conséquence de déformer le signal. Le coefficient de variation de la distribution peut être interprété comme une mesure du bruit. Nous avons vu que ce coefficient est constant et ne dépend pas de la fréquence des tâches.

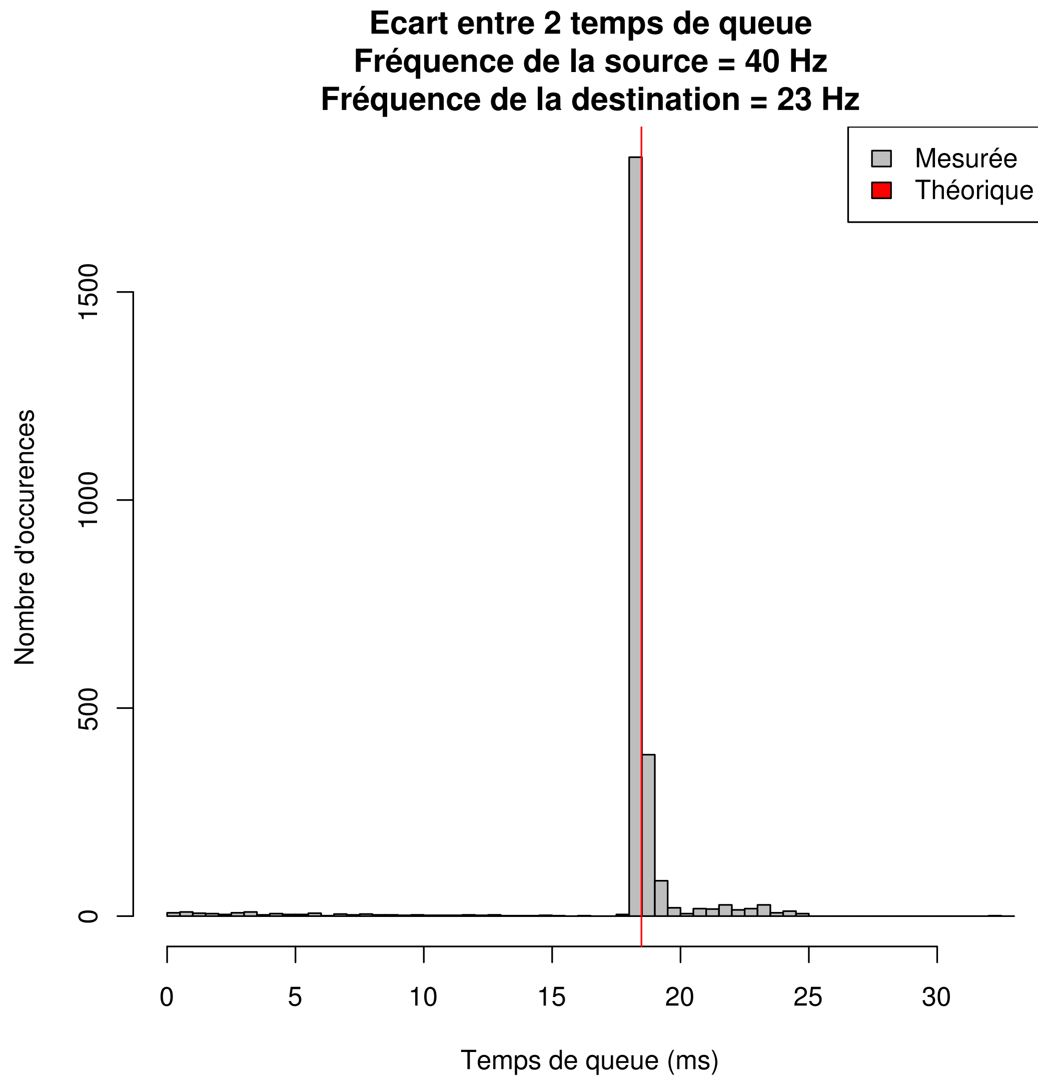


FIGURE 10.3 – Distribution du temps d'attente mesuré des messages par rapport au temps d'arrivée dans la queue du message précédent. La ligne rouge $x = \tau_d \mod \tau_s \simeq 17ms$.

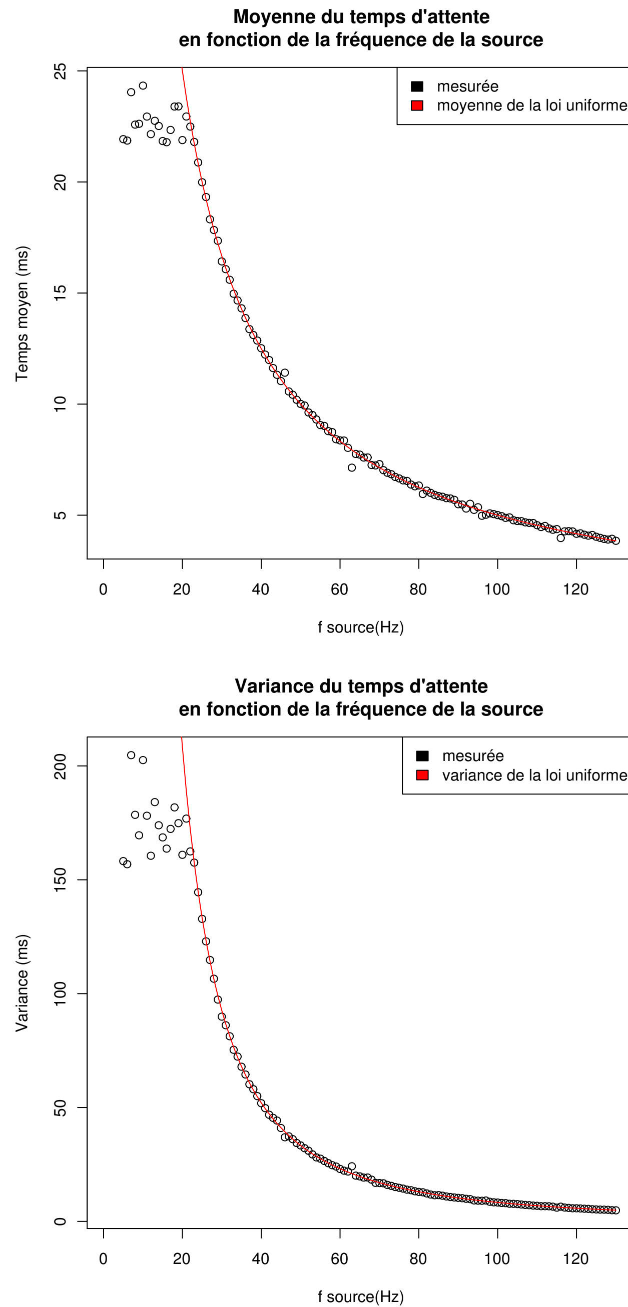


FIGURE 10.4 – Comparaison du temps d'attente constaté expérimentalement avec une loi uniforme. La destination a une fréquence de 24Hz. a)Evolution du temps d'attente moyen en fonction de la fréquence de la source. b) Variance du temps d'attente en fonction de la fréquence de la source.

Chapitre 11

Mécanismes permettant de limiter l'impact de l'échantillonnage

11.1 Introduction

Au chapitre précédent, le résultat du théorème 10.2.1 nous indique que le temps d'attente est donné par un générateur pseudo-aléatoire. Selon les paramètres de τ_s et τ_d, les périodes d'itérations, q_n simule une loi uniforme. Cette proposition confirme les résultats énumérés précédemment par Wloka. Nous en avons déduit les valeurs de la latence moyenne et de la gigue. Empiriquement, nous pouvons retrouver ces conclusions.

Le coefficient de variation CV qui représente la relation entre la latence et la gigue est une constante valant $\frac{1}{\sqrt{3}}$. La gigue est donc proportionnelle à la latence. Cette relation ne dépend pas de la fréquence des tâches. Par conséquent, l'accélération des fréquences des composants de l'application ne supprimera pas la gigue.

Nous cherchons maintenant une solution pour limiter la latence et supprimer la gigue. A partir des résultats du chapitre précédent, nous proposons maintenant des pistes pour résoudre le problème du *synchronization lag.*

11.2 Un critère pour supprimer la gigue

Les périodes d'itération des tâches τ_s et τ_d influencent la qualité du générateur pseudo-aléatoire. Pour limiter l'écart type de la distribution, nous ne cherchons pas à produire un générateur qui simule une loi uniforme. En effet, la moyenne de cette distribution est importante et la variance est non nulle. Nous essayons de trouver les paramètres pour lesquels le générateur ne simule pas une distribution uniforme.

Nous avons vu grâce au test de Kolgomorov (Figure 10.1) que pour certains rapports entre τ_s et τ_d, la distribution ne simule pas la loi uniforme. Dans ce cas, la moyenne et la variance de la distribution diffèrent de celles de la loi uniforme. En choisissant de façon pertinente les valeurs τ_s et τ_d, il est possible de modifier l'impact du *synchronization lag*.

Proposition 11.2.1 *Si* $\tau_d \mod \tau_s = 0$*, c'est à dire que la fréquence de la source est un multiple de celle de la destination, alors*

$$\begin{aligned} Q &= \mathrm{E}[q_n] = q_0 \\ V &= \mathrm{V}[q_n] = 0 \end{aligned}$$

Preuve Nous insérons l'hypothèse $\tau_d \mod \tau_s = 0$ dans le résultat du théorème 10.2.1, nous obtenons :

$$\forall n, q_n = q_{n-1} \mod \tau_s$$

Par définition, q_n est plus petit que τ_s. Par conséquent, nous avons :

$$\forall n, q_n = q_0$$

La moyenne est donc q_0 et la variance est nulle. ∎

Le cas où $\tau_d \mod \tau_s = 0$ est un cas simple où q_n ne simule pas une loi uniforme. Si nous ajoutons cette hypothèse dans le résultat du théorème 10.2.1, nous obtenons : $\forall n, q_n = q_0$.

En s'assurant que la période de la tâche destination est un multiple de celle de la source, le temps d'attente dans la queue est alors constant. La gigue est donc inexistante. Dans ce cas, il n'y a pas de déformation du signal liée à l'échantillonnage.

L'exemple de la figure 11.1 illustre ce phénomène. Un signal simple est échantillonné à deux fréquences différentes f_1 et f_2. Après cette première étape, les résultats sont de nouveau échantillonnés à la fréquence f_1. Nous remarquons que si nous échantillonnons deux fois à la même fréquence, le signal n'est pas déformé. Ici même si $f_1 < f_2$, le premier signal comportera moins de bruit que le second après le deuxième échantillonnage.

Toujours pour illustrer le phénomène, nous simulons la trajectoire d'un pendule simple par la formule suivante. $\theta(t)$ est l'angle du pendule avec la verticale à l'instant t.

$$\theta(t) = \theta_0.cos(\omega_0.t)$$

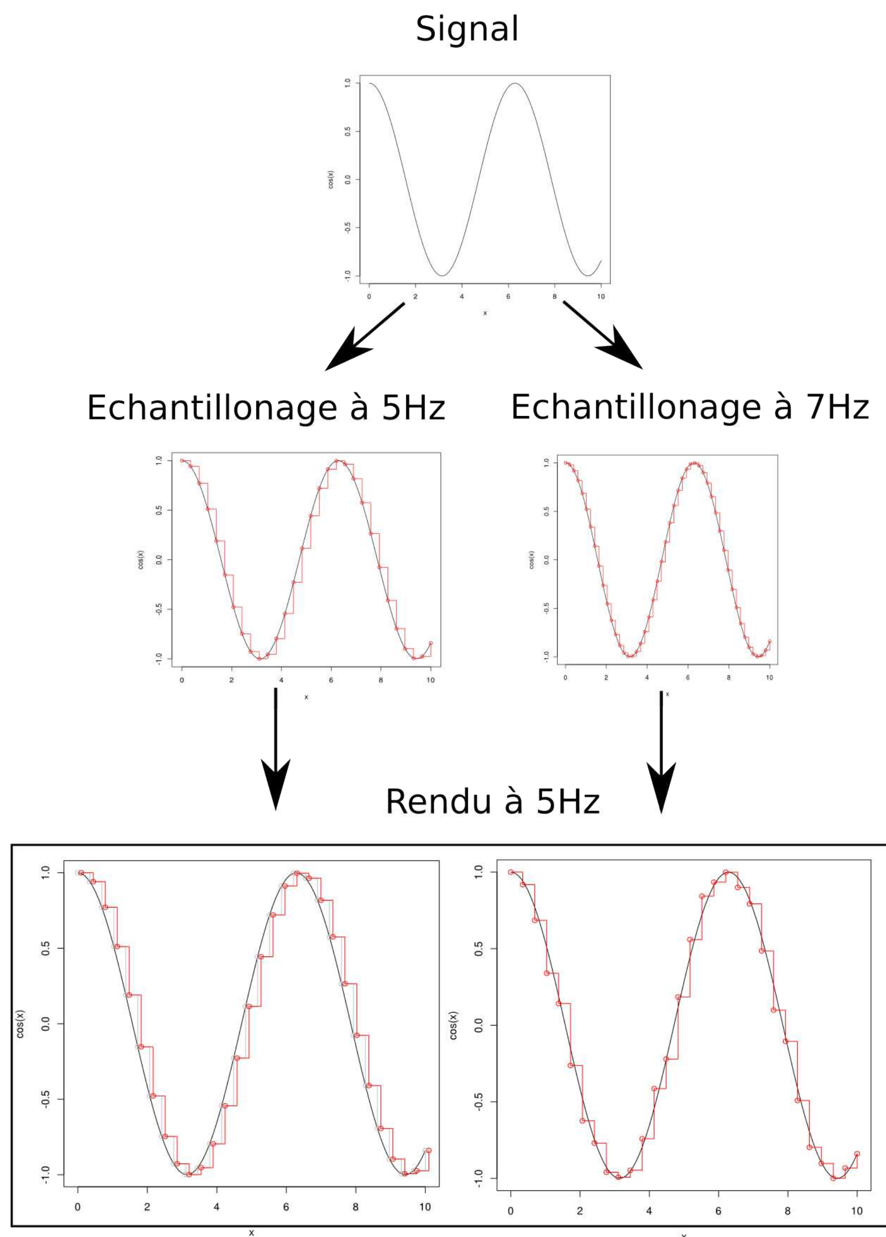


FIGURE 11.1 – Exemple de la déformation due à la gigue. Un signal simple est échantillonné à 5Hz et 7Hz. Le signal numérisé est rendu à 5Hz. Cela consiste à de nouveau échantillonner le signal à 5Hz. A gauche, le signal n'est pas déformé et il subit une latence constante q_0. A droite, en raison du phénomène de gigue, le signal est déformé.

θ_0 est l'angle où le pendule est lâché et ω_0 est une constante qui donne la fré-

quence d'oscillation. Deux composants calculent la valeur de $\theta(t)$ à des fréquences différentes. Cela revient à échantillonner le mouvement du pendule. Ces deux composants envoient au même processus de rendu les angles du pendule. Une seconde étape d'échantillonnage permet d'adapter la fréquence d'envoi des deux simulateurs à la fréquence du processus du rendu. On fixe les fréquences suivantes :

- La fréquence du simulateur 1 est de 5Hz
- La fréquence du rendu est de 5 Hz

Le simulateur 1 est un témoin. Nous faisons varier la fréquence du simulateur 2 entre 5 Hz et 20 Hz. Pour faciliter la visualisation de la gigue, nous affichons à tout instant les quatre dernières positions des pendules. Si ces positions ne sont pas espacées uniformement, c'est une mise en évidence du bruit dû à l'échantillonnage. Nous vous invitons à télécharger la vidéo présentant le résultat de l'expérience [1]. Au delà de 15 Hz, comme la gigue est bornée par la période d'itération, elle devient trop petite pour être observable. Par contre, nous remarquons sur la vidéo que la gigue est inexistante pour les fréquences de 5Hz, 10Hz, 15Hz et 20 Hz

11.2.1 Discussion sur le respect du critère

Nous cherchons à respecter $\tau_d \mod \tau_s = 0$. Nous avons donc une relation entre les périodes des deux composants. Si nous connaissons une des périodes, nous pouvons calculer l'autre période. Cela consiste en pratique à anticiper les itérations d'un composant pour caler les itérations du second composant. Cette méthode de synchronisation est d'ailleurs préconisée dans [106]. Nous avons cité précédement cet article car il proposait une mesure de l'impact du temps d'attente des messages dans l'échantillonneur.

Dans des applications complexes, respecter le critère $\tau_d \mod \tau_s = 0$ n'est pas trivial. Les différentes sources de perturbation que nous avons ignorées pour mettre en valeur le phénomène de variance existent dans la pratique. Par conséquent, les périodes d'itération peuvent varier au cours du temps. Des perturbations comme le réseau, peuvent bruiter la mesure des périodes τ_d et τ_s.

Une solution envisageable pour enlever l'impact de ces perturbations serait de choisir les périodes des tâches pour qu'elles soient toujours respectées malgré toutes les sources perturbations. Cela revient à imposer des périodes τ_s et τ_d suffisamment grandes aux tâches et leur fournir une marge pour qu'elles ne soient pas sensibles aux perturbations. En pratique, cela revient à ne pas exécuter les tâches à leur fréquence maximale et de les ralentir.

Pour choisir les valeurs des périodes d'itérations, nous pouvons réaliser des mesures de performance. Ces mesures donneraient la période maximale des tâches,

1. Une vidéo de l'expérience est disponible à l'adresse : http://moais.imag.fr/membres/jean-denis.lesage/these/pendulum.avi

l'impact du réseau, l'influence des autres processus sur la machine, etc. En pratique, cela se traduirait par une phase d'étalonnage, où des tests de performances s'exécuteraient sur la machine et l'application. Cela ajouterait une complexité supplémentaire pour l'utilisateur. Sur des applications de grandes tailles, cette phase pourrait être extrêmement coûteuse.

Pour que la méthode soit utilisable sur des applications de grande taille, il serait nécessaire de supprimer cette phase d'étalonnage. Le travail est en cours. Notre piste consiste à proposer des algorithmes adaptatifs qui mesureraient les différentes sources de perturbation en cours d'exécution de l'application. L'algorithme modifie les périodes d'itération des tâches pour s'adapter aux perturbations.

11.3 Echantillonneur à seuil

11.3.1 Description de l'échantillonneur

Respecter le critère précédent est difficile dans certains cas. Nous proposons maintenant une extension de l'échantillonneur classique. Cette solution permet de borner la latence moyenne et sa variance sans modifier le coefficient de variation. Comme nous l'avons signalé dans l'introduction de ce chapitre, le coefficient de variation reste important. Par conséquent, après normalisation, la distribution des temps d'attente est toujours dispersée. Cette solution ne sera donc pas une solution définitive. Par contre, il est possible grâce à cette solution de réduire la latence et la gigue sous un seuil où elles ne seront pas perceptibles par un utilisateur.

La motivation pour réaliser cette échantillonneur est venue de l'étude de l'échantillonneur classique. L'intuition que nous pouvons avoir après l'étude précédente est que si q_n est proche de sa valeur maximale τ_s, il est peut-être préférable de ne pas traiter ce message et d'attendre le suivant. Dans ce cas, le nouveau message sera traité dès sa création, son temps d'attente dans l'échantillonneur sera nul. De plus, l'attente apportée par ce mécanisme sera faible car égale à $\tau_s - q_n \simeq 0$ d'après notre hypothèse.

Sur l'échantillonneur précédent, nous ajoutons un seuil d'attente maximum des messages (que nous appellerons Th). L'échantillonneur intègre ainsi la règle suivante : si $q_n \geq Th$ alors `Jeter le message n`. Dans ce cas, le message est effacé de la mémoire de l'échantillonneur. Si le message est effacé, l'échantillonneur doit attendre une nouvelle donnée pour être capable de répondre aux requêtes de la `Destination`. La `Destination` est bloquée jusqu'à la production d'un nouveau message par la `Source`.

Proposition 11.3.1 *Si* $\tau_d \mod \tau_s \neq 0$ *et* $Th < \tau_s - (\tau_d \mod \tau_s)$ *alors la suite* q_n *est cyclique de période* $M = \left\lceil \frac{Th}{\tau_d \mod \tau_s} \right\rceil + 1$

De plus, les valeurs de la suite q_n sur une période sont $(0, \tau_d \mod \tau_s, 2(\tau_d \mod \tau_s), \ldots, (M-1)(\tau_d \mod \tau_s))$

Preuve A chaque fois que la queue est vidée, la destination doit attendre un nouveau message pour démarrer une nouvelle itération. La `Destination` attend jusqu'à la production de ce message. Ce message est alors directement transféré à la `Destination`. Après que la queue soit vidée, il existe n tel que $\bar{s_n} = d_n$. Donc $q_n = 0$.

Selon la valeur de q_n, nous allons décider si la queue de l'échantillonneur devra être vidée à l'itération suivante.

– Si $q_n < Th - (\tau_d \mod \tau_s)$, nous avons :

$$q_{n+1} = q_n + \tau_d \mod \tau_s < Th$$

Par conséquent la queue ne sera pas vidée à l'itération $n + 1$.

– Si $Th - (\tau_d \mod \tau_s) \leq q_n \leq Th$, nous avons

$$Th \leq q_n + \tau_d \mod \tau_s \leq Th + (\tau_d \mod \tau_s)$$

$$Th \leq q_{n+1} = q_n + \tau_d \mod \tau_s \leq \tau_s$$

Par conséquent la queue sera vidée à l'itération $n + 1$.

Grâce à ce résultat, nous allons déterminer combien de messages seront envoyés par l'échantillonneur entre deux instants où la queue a été vidée.

– Soit n tel que $q_n = 0$ et $k < \left\lfloor \frac{Th}{\tau_d \mod \tau_s} \right\rfloor$, alors on a

$$q_n + k.\tau_d \mod \tau_s = \left\lfloor \frac{Th}{\tau_d \mod \tau_s} \right\rfloor .(\tau_d \mod \tau_s) \leq Th$$

La queue ne sera donc pas vidée et $q_{n+k+1} = (k+1).\tau_d \mod \tau_s$.

– Si $k = \left\lfloor \frac{Th}{\tau_d \mod \tau_s} \right\rfloor$, alors $Th - (\tau_d \mod \tau_s) \leq q_{n+k} \leq Th$. Cela implique donc $q_{n+k+1} = 0$.

La suite q_n est donc périodique. La période est

$$M = \left\lfloor \frac{Th}{\tau_d \mod \tau_s} \right\rfloor + 1 \tag{11.1}$$

Les valeurs sur une période sont $(0, \tau_d \mod \tau_s, 2(\tau_d \mod \tau_s), \ldots, (M-1)(\tau_d \mod \tau_s))$ ∎

Avec cet échantillonneur, il est possible de calculer les valeurs des éléments de la suite q_n. Nous pouvons maintenant calculer formellement la moyenne et la variance de la distribution des temps d'attente sans utiliser l'hypothèse que la distribution q_n suit une loi uniforme.

11.3.2 Moyenne et variance de l'attente d'un message

Proposition 11.3.2 *Dans les mêmes conditions que la proposition 11.3.1, l'attente moyenne Q est comprise entre* $[\frac{Th-(\tau_d \mod \tau_s)}{2}, \frac{Th}{2}]$ *et la variance V est comprise entre* $[\frac{Th-(\tau_d \mod \tau_s)}{2\sqrt{3}}, \frac{Th}{2\sqrt{3}}]$

Preuve La suite q_n est cyclique donc Q vaut la moyenne des éléments d'une période. C'est une suite arithmétique de raison $(\tau_d \mod \tau_s)$ dont le premier élément est 0 et ayant M éléments. D'après la formule de la somme des éléments d'une suite arithmétique, nous avons :

$$Q = \frac{1}{M}.\frac{(M-1)M}{2}(\tau_d \mod \tau_s) = \frac{M-1}{2}(\tau_d \mod \tau_s)$$

Nous utilisons l'expression de M (11.1) : $M - 1 = \left\lfloor \frac{Th}{\tau_d \mod \tau_s} \right\rfloor$.
La partie entière peut être encadrée de la façon suivante :

$$\frac{Th}{\tau_d \mod \tau_s} - 1 \leq M - 1 \leq \frac{Th}{\tau_d \mod \tau_s}$$

En remplaçant dans l'expression de Q, nous obtenons l'encadrement :

$$\frac{Th - (\tau_d \mod \tau_s)}{2} \leq Q \leq \frac{Th}{2}$$

La variance V est donnée par

$$V = \mathrm{E}[q_n^2] - \mathrm{E}[q_n]^2$$

L'expression de V se simplifie en utilisant le résultat de la somme des premiers carrés d'entier.

$$\begin{aligned}
V &= \frac{1}{M}\sum_{i=0}^{M-1}(i.(\tau_d \mod \tau_s))^2 - Q^2 \\
V &= \frac{1}{M}\frac{(2.M-1)(M-1).M}{6}(\tau_d \mod \tau_s)^2 - \frac{(M-1)^2}{4}(\tau_d \mod \tau_s)^2 \\
V &= (\tau_d \mod \tau_s)^2.\frac{(M-1)}{2}.\left(\frac{2M-1}{3} - \frac{M-1}{2}\right) \\
V &= \frac{(M-1)^2}{12}(\tau_d \mod \tau_s)^2
\end{aligned}$$

En faisant le même encadrement de M que précédemment, nous avons

$$\frac{(Th - (\tau_d \mod \tau_s))^2}{12} \leq V \leq \frac{Th^2}{12}$$

Sur ces deux résultats, les bornes supérieures sont uniquement fonction du seuil Th. Très naturellement, un seuil Th bas implique une moyenne et une variance du temps d'attente faibles. Grâce à ce seuil, il est donc possible de contrôler ces paramètres. La contrepartie est que la fréquence avec laquelle l'échantillonneur vide la queue dépend aussi de ce paramètre Th. L'échantillonneur videra la queue tous les M messages. Quand la queue est vidée, la `Destination` se met en attente d'un nouveau message. M est aussi proportionnel à Th. Par conséquent, si Th est trop faible, alors il est possible que la `Destination` soit trop souvent en attente.

11.3.3 Inactivité de la `Destination`

Si Th est fixé alors M, la longueur d'un cycle de q_n, dépend uniquement de $\tau_d \mod \tau_s$. Nous cherchons maintenant à résoudre le problème suivant. A un seuil donné, à quelle fréquence doit-on faire fonctionner la `Source` et la `Destination` pour que la `Destination` attende le moins possible. Dans ce cas, la `Destination` sera la plus active.

D'après la formule (11.1), M tend vers l'infini lorsque $\tau_d \mod \tau_s$ tend vers 0. Nous avons alors τ_d est un multiple de τ_s. Par conséquent, la `Source` fonctionne à une fréquence qui est un multiple de la fréquence de la `Destination`.

Expérimentalement, nous voulons retrouver ce résultat. Nous reprenons l'expérience du chapitre précédent (Fig. 10.5.1). L'échantillonneur a été modifié par l'échantillonneur à seuil. Pour un seuil Th fixé à $5ms$ et une fréquence de la tâche destination à 28Hz, nous mesurons M. Les résutats sont donnés par la figure 11.2. Pour chaque fréquence de la tâche source multiple de la tâche destination, nous mesurons un pic de la valeur M.

Une autre méthode pour visualiser ce résultat, consiste à mesurer directement le temps d'inactivité moyen de la `Destination` en utilisant toujours l'échantillonneur à seuil. Sur le même principe que l'expérience précédente, nous mesurons sur 2 minutes le temps moyen d'inactivité de la `Destination`. Les résultats sont résumés dans la figure 11.3. La courbe noire correspond à τ_s. C'est le temps d'une itération de la source. Elle correspond au temps d'inactivité maximum de la `Destination`. En effet, il suffit que la `Source` produise un nouveau message pour réveiller la `Destination` après que la queue de l'échantillonneur ait été vidée. L'expérience confirme que si $\tau_d \mod \tau_s = 0$ alors la `Destination` est presque toujours active. C'est donc dans ce cas que la `Destination` sera la plus efficace.

11.4 Bilan

Pour conclure ce chapitre, nous avons montré que le critère simple $\tau_d \mod \tau_s = 0$ permet pour l'échantillonneur classique de supprimer la gigue. En appliquant ce

critère, nous pouvons supprimer le bruit dû à cet échantillonneur. Les travaux futurs devront proposer des algorithmes permettant de respecter ce critère.

Nous proposons un mécanisme plus simple pour limiter l'impact de la latence. Ce mécanisme ne permet cependant pas de diminuer le coefficient de variation. Il limite mais ne supprime pas la gigue. Cette extension ajoute à l'échantillonneur classique un temps de vie maximum aux messages. Au delà de ce seuil, le message est détruit dans la queue de l'échantillonneur. L'échantillonneur ne peut pas produire de messages jusqu'à ce que de nouvelles données soient insérées.

Cette extension simple permet de borner la latence et la gigue proportionnellement à la valeur du seuil. Les bornes obtenues ne permettent pas de conclure sur la diminution du coefficient de variation. La gigue reste donc proportionnelle à la latence. Le signal pourra donc être déformé à la sorti de l'échantillonneur. Cependant, en choisissant un seuil assez bas, il est possible de rendre l'impact de la latence indétectable pour un utilisateur humain.

La contrepartie de cette extension est qu'elle ralentit la fréquence d'exécution de la tâche destination. Pour réduire cette inactivité, nous devons respecter la contrainte $\tau_d \mod \tau_s = 0$. Nous retrouvons la même contrainte que précédemment. Le respect de cette condition est ici moins critique car nous avons borné la latence et la gigue. Si nous ne respectons pas le critère, seule la fréquence d'itération des tâches sera ralentie.

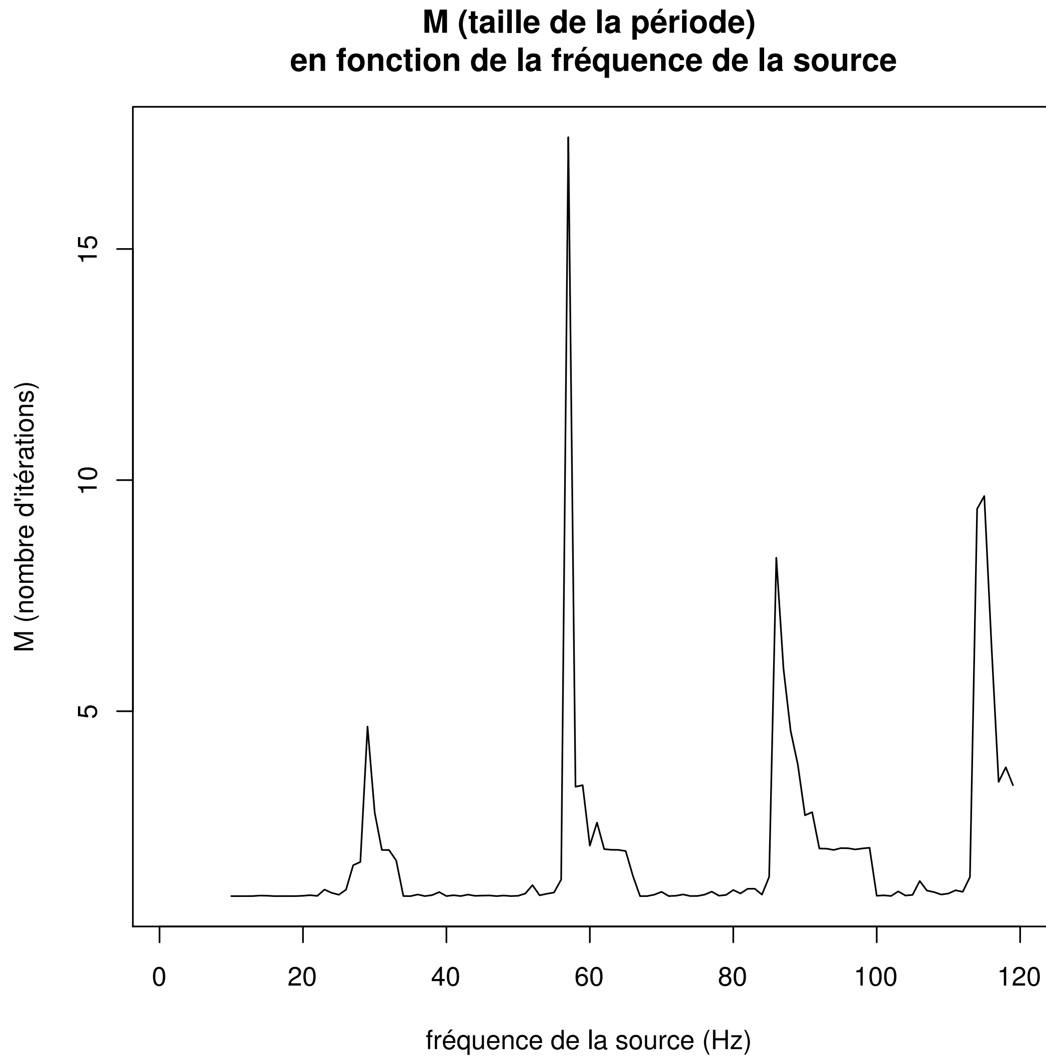


FIGURE 11.2 – Nombre de messages traités en moyenne avant d'atteindre le temps maximum d'attente en fonction de la fréquence de la source. Le temps maximum est de 5 ms et la destination a une fréquence de 28Hz

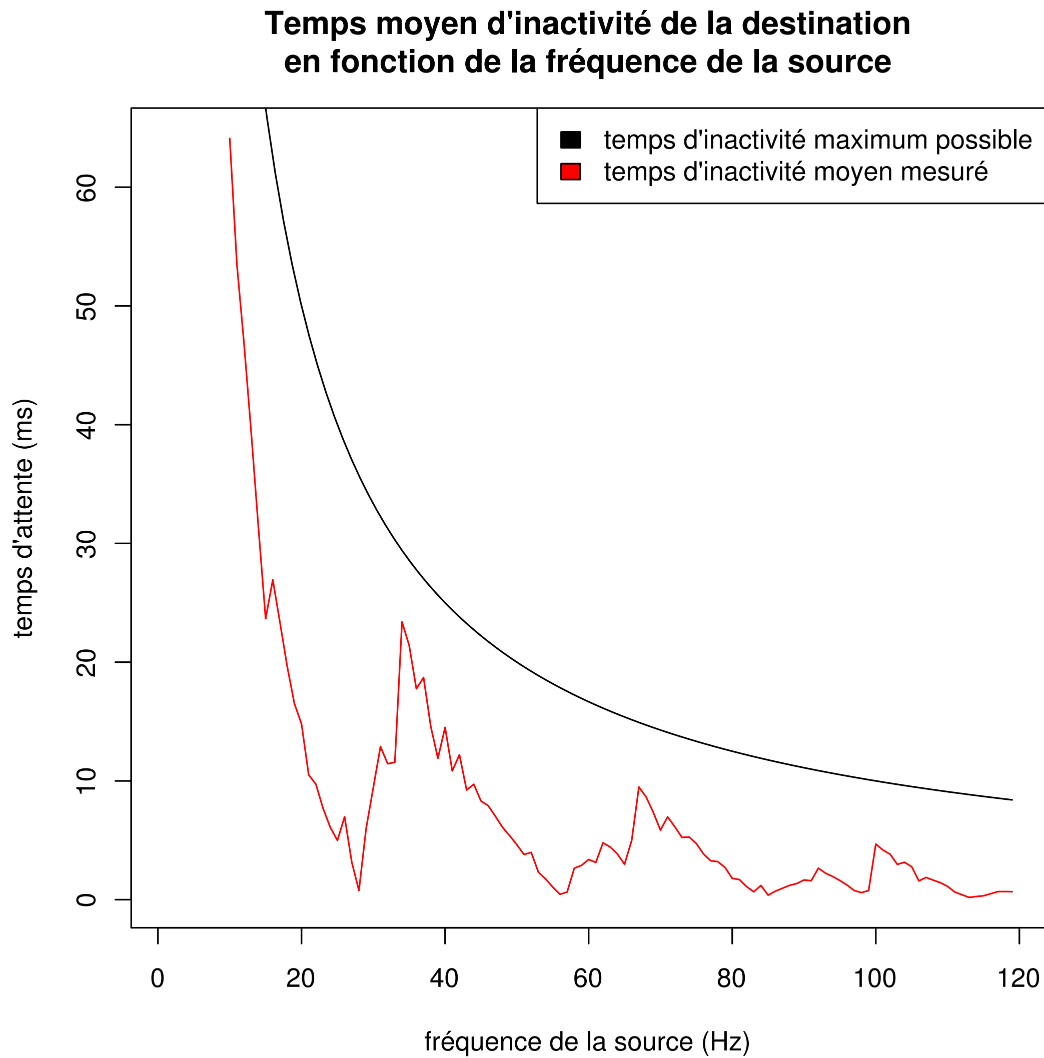


FIGURE 11.3 – Temps moyen d'inactivité de la destination dans le cas où $Th = 5ms$. La courbe théorique indique l'inactivité maximum.

Chapitre 12

Régulation pour les applications interactives

12.1 Introduction

Les chapitres précédents portaient sur l'impact de l'échantillonneur. Dans ce chapitre, nous regardons un autre aspect du couplage de tâches. Nous cherchons maintenant à utiliser efficacement le parallélisme de pipeline dans les applications interactives.

Dans l'état de l'art, nous avons cité Ahrens [52] qui a énuméré trois utilisations possibles du parallélisme dans les applications interactives. Ces formes de parallélisme sont le parallélisme de données, de tâches et de pipeline. Le parallélisme de pipeline est naturel à implémenter dès que nous disposons d'une représentation sous forme de graphe de l'application. En effet, il suffit d'associer un processeur par tâche pour utiliser le parallélisme de pipeline. Ce parallélisme est une fonctionnalité importante que doivent proposer les middlewares. C'est pour cette raison que nous y consacrons ce chapitre.

Dans le chapitre 9, nous avons identifié trois sources possibles d'échantillonnage dans les applications interactives. Nous verrons dans ce chapitre que, pour utiliser efficacement le parallélisme de pipeline, il est nécessaire de jeter les données dont le traitement ralentirait l'application. Nous étudierons les protocoles de régulation de flux. La régulation nécessite l'utilisation d'échantillonneur. Ce chapitre complète donc l'étude précédente sur le *synchronization lag*.

Ce chapitre est construit à partir des résultats d'expériences obtenus lors des derniers mois de ma thèse. Ce travail est préliminaire.

12.2 Parallélisme de pipeline

Un pipeline est une suite de traitements à exécuter sur des données. Le parallélisme de pipeline consiste à exécuter en parallèle les différentes étapes du pipeline. Ce parallélisme permet d'augmenter la fréquence de production des données. La figure 12.1 montre l'utilisation de ce parallélisme.

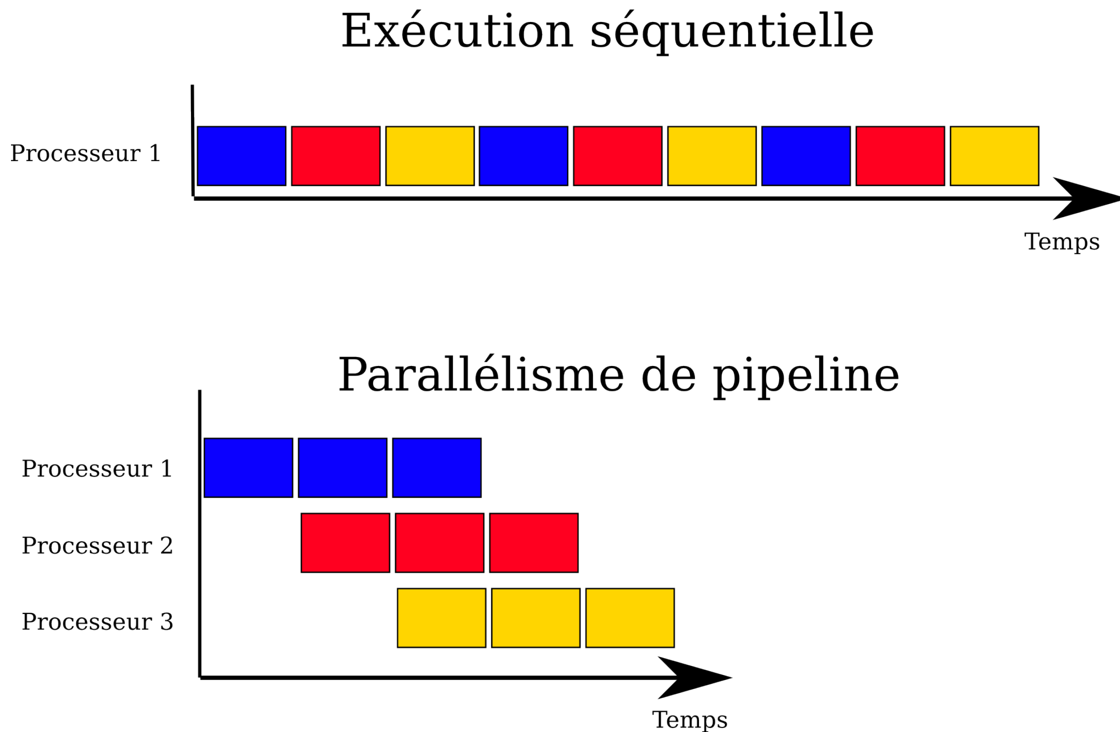


FIGURE 12.1 – Parallélisme de pipeline. Execution de trois itérations consécutives d'un pipeline comportant trois tâches représentées par des couleurs différentes. Le parallélisme de pipeline permet d'augmenter le débit total de l'application.

Ce mode de parallélisme est assez simple à mettre en oeuvre. A partir d'une representation par flot de données d'une application, nous pouvons assigner un processeur par tâche. En implémentant les connexions par des files FIFO, nous pouvons implémenter le parallélisme de pipeline. Un producteur de données dans le graphe pousse les données dans la file FIFO. Le lecteur tire les données de la file. Si la file est vide, le lecteur est bloqué jusqu'à l'apparition d'un nouveau message. Cette méthode garantit que les messages seront lus dans l'ordre de leur création et que chaque message sera lu une et une seule fois.

En pratique, la taille de la file est limitée. Si la source de messages est plus rapide que la destination, le nombre de messages dans la file va augmenter. Il

arrivera un moment où la taille de la file ne sera pas suffisante pour stocker tous les messages. Il y aura une explosion de la taille de la file (*buffer overflow*).

Pour résoudre ce problème, nous pouvons identifier deux solutions. Pour nommer ces solutions, nous nous inspirons des protocoles réseaux au dessus de IP. En effet, les mécanismes sont assez similaires.

Méthode TCP

Quand la file est pleine, la méthode qui pousse les données devient bloquante. La source de données se bloque et ne produit plus de données. Il n'y a plus de risque d'explosion de la taille de la file. Cette solution n'utilise pas d'échantillonneur, il n'y a pas de perte de messages.

Méthode UDP

Quand la file est pleine, la connexion permet de jeter ou ignorer un message. Il y a un choix dans le message à jeter ou ignorer. Nous pouvons éliminer le message venant d'être produit comme dans le protocole UDP ou le plus vieux dans la file. Dans ce cas, la file est une fenêtre contenant les N messages les plus récents. Cette deuxième variante est priviligiée dans les applications interactives. Le développeur aura tendance à conserver en priorité les messages les plus récents pour gagner en réactivité. Cette méthode permet la perte de messages, elle peut donc être vue comme un échantillonneur.

12.3 Comparaisons entre les méthodes UDP et TCP

Dans le cadre des applications multimédia web, la méthode UDP est souvent préférée. Le protocole RTP utilisé pour le transfert en ligne de média multimédia (vidéo, sons) est un protocole de régulation. La norme préconise l'utilisation d'UDP [76]. La possibilité d'éliminer des messages semble importante.

Plus généralement pour les applications interactives, nous proposons de vérifier expérimentalement ce même résultat. Cette vérification illustrera l'intêret d'utiliser l'échantillonnage au sein des applications interactives.

12.3.1 VTK

Nous avons écrit une librairie au dessus de VTK [15] qui permet de modifier la politique d'échange de messages entre des composants VTK. VTK est une librairie pour la conception d'applications de visualisation scientifique. Cette librairie est soutenue par une grande communauté. VTK est riche de centaines d'algorithmes

dédiés au traitement de données. Il existe dans la distribution des algorithmes de visualisation scientifique comme par exemple le *marching-cube* [112] ou le rendu volumique par lancer de rayons [113]. Mais cette librairie comporte aussi des algorithmes de traitement d'images, de manipulation ou de représentation de graphes ou encore de données géographiques. Tous ces algorithmes sont encapsulés dans des composants VTK. Comme pour les autres modèles à composants, un composant a une interface composée de ports d'entrée et de sortie. Un composant VTK implémente au minimum les fonctions suivantes :

- Une fonction pour lancer le calcul (souvent appelée `Update`)
- Une méthode `SetInput` qui permet de pousser une donnée sur un port d'entrée
- Une méthode `GetOutput` qui permet de récupérer une donnée produite par le composant

Les tâches VTK sont pour l'essentiel mono-threadées. Seuls quelques algorithmes dédiés au traitement d'image et héritant de la classe `vtkThreadImageAlgorithm` sont multi-threadés. Il est alors possible de contrôler le nombre de threads qui effectueront le calcul.

VTK est une plateforme intéressante pour comparer sur une application interactive les différentes politiques de couplage de composants. Il est possible de construire simplement des applications complexes. L'application est décrite sous forme d'un DAG. Les méthodes `SetInput` et `GetOutput` décrivent le graphe. Il est possible de les redéfinir pour remplacer le pipeline VTK par un nouveau pipeline générique. La fonction de calcul de chaque composant peut-être lancée manuellement grâce à la méthode `Update`. Cela permet au final d'implémenter un nouveau ordonnanceur de tâches.

Dans VTK, les dépendances entre les tâches sont décrites dans les descripteurs des données. VTK implémente une stratégie *demand-driven*. L'appel à la fonction `Update` fera remonter une requête de mise à jour le long du graphe décrivant l'application. Ces descripteurs forment une structure chaînée qui décrivent totalement le graphe de l'application (Figure 12.2). Pour briser la liste chaînée et éviter que les requêtes remontent dans le graphe, notre sur-couche doit modifier les descripteurs en supprimant l'information qui indique qui est le producteur de la donnée. Cette information sera stockée en remplacement dans une structure de graphe qui décrira l'application. Pour réaliser cette opération, il est possible soit de changer directement ce champ dans la structure, soit de cloner la donnée. Les méthodes de copies de VTK ne dupliquent pas le champ `Source`. Ce champ identifie le producteur de données et fait le chaînage entre les composants.

VTK ne propose pas de méthodes de sérialisation des données. Pour proposer une version distribuée du pipeline VTK, il est nécessaire aujourd'hui d'implémenter ces méthodes de sérialisation pour tous les types de données présents dans VTK.

Pour cette raison, notre pipeline générique ne fonctionne qu'en mémoire partagée.

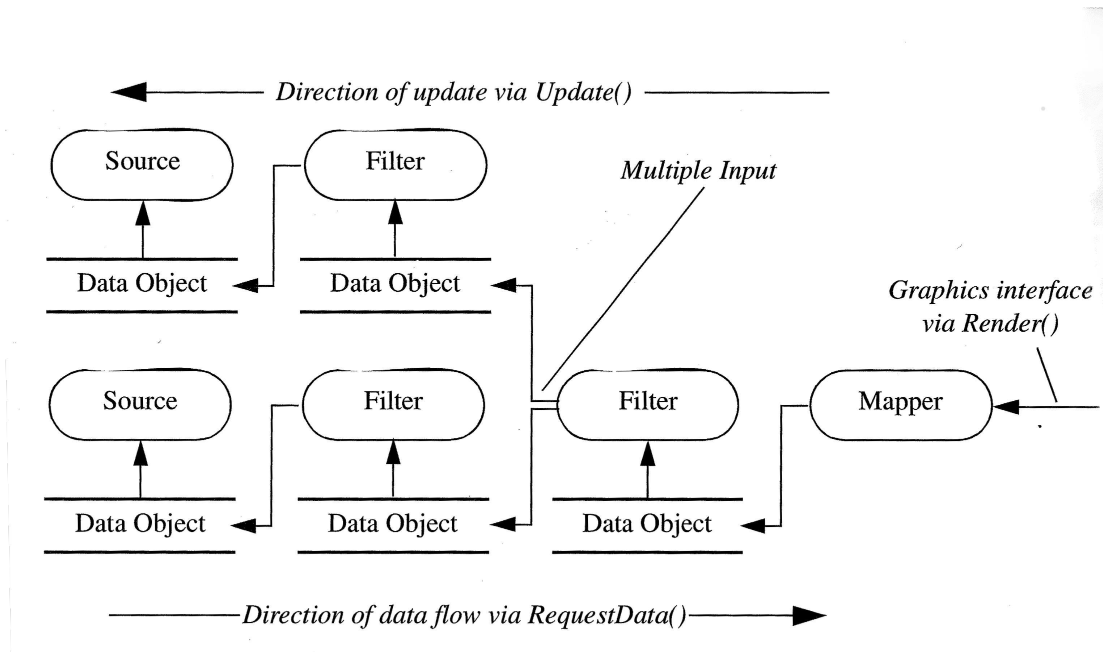


FIGURE 12.2 – Le graphe VTK est décrit par un chaînage reliant les descripteurs de données. Illustration tirée de [15] chapitre 4.

12.3.2 Dispositif expérimental

Application

L'application que nous allons utiliser prend en entrée une grille irrégulière 3D de valeurs comprises entre $[0, 1]$. Grâce à l'algorithme du *marching-cube* [112] une isosurface est calculée. Cette surface est rendue dans la carte graphique. L'image finale est transférée de la mémoire vidéo de la carte graphique vers la RAM de la machine. Un filtre va ensuite traduire l'espace de couleur de l'image de RGB (Rouge-Vert-Bleu) vers l'espace HSV (Teinte-Saturation-Valeur). Le canal de valeur V (Valeur) est extrait pour lui appliquer l'opérateur Laplacien. Ce filtre va extraire les hautes-fréquences de l'image qui seront pour l'essentiel les contours de l'isosurface. Le résultat est retraduit dans l'espace de couleurs RGB. Les pixels correspondant à des hautes-fréquences sont mis en noir et les autres en blanc. Le résultat est alors écrit dans un fichier au format PNG. La figure 12.3 résume l'ensemble de ces étapes. La valeur de l'isosurface est modifiée à chaque itération en entrée du pipeline. Les données de la grille étant statiques, la grille est chargée en mémoire uniquement lors de la première itération.

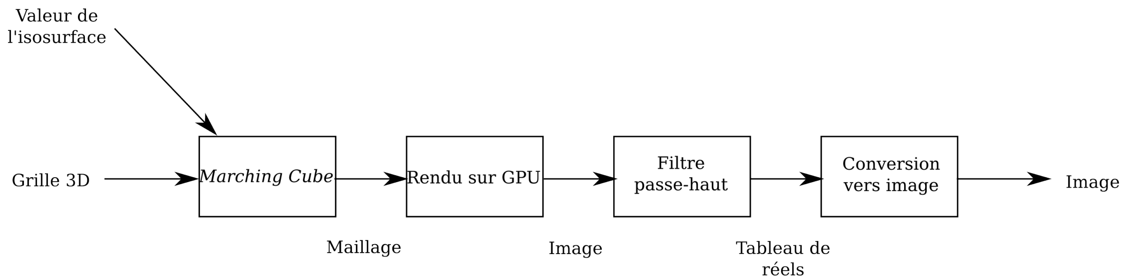


FIGURE 12.3 – Application test VTK, les principales étapes du pipeline sont représentées.

Cette application comporte au final 13 composants VTK. Le graphe décrivant l'application est une chaîne. La complexité de l'algorithme du *marching-cube* dépend de la taille de la grille. La complexité du rendu est proportionnelle à la taille de l'isosurface et la taille de l'image finale. Les dernières étapes de l'application traitant l'image ont une complexité qui dépend de la taille de l'image. Pour les tests, nous avons un jeu de données de 8Mo. Le rendu final a une résolution de 2000 pixels par 2000 pixels. Avec ce jeu de données, les tâches de traitement d'images sont plus coûteuses que le *marching cube*.

Matériel

Les expériences ont toutes été réalisées sur un noeud de la grappe Digitallis. Le noeud comporte 2 quad-core Xeon2 Nehalem.

La compilation des exécutables et de la librairie VTK a été réalisée avec gcc en utilisant les options de compilation `-O3 -march=native` pour utiliser l'ensemble du jeu d'instructions disponible.

12.3.3 Résultats

L'implémentation actuelle VTK [15] n'utilise pas par défaut le parallélisme de pipeline. Nous allons donc comparer les valeurs de la latence et de fréquence obtenues avec le parallélisme de pipeline avec celle de l'implémentation traditionnelle de VTK sur la même application. Les valeurs que nous allons mesurer sont les suivantes :

L_{VTK} **et** f_{VTK} sont respectivement la latence et la fréquence de l'application en utilisant VTK.

L_{TCP} **et** f_{TCP} sont respectivement la latence et la fréquence de l'application en utilisant la méthode TCP.

L_{UDP} **et** f_{UDP} sont respectivement la latence et la fréquence de l'application en utilisant la méthode UDP. Nous utilisons la variante qui consiste à écarter le message le plus vieux quand la file est pleine.

Nous pouvons contrôler la fréquence du composant en amont du pipeline. Cette fréquence est le paramètre de l'expérience. Pour chacune des fréquences, nous mesurons sur deux minutes d'expérience la latence et la fréquence moyenne de l'application.

La figure 12.4 compare TCP avec VTK. La figure 12.5 compare UDP avec VTK. Nous remarquons en premier lieu que le parallélisme de pipeline permet d'accélérer la fréquence de l'application. Pour les deux méthodes, l'accélération est légérement inférieure à 2. Nous remarquons cependant une différence sur la latence. La méthode TCP produit un net ralentissement de la latence. En utilisant ce protocole, nous obtenons une latence 5 fois supérieure à celle de VTK. La méthode UDP permet de limiter ce ralentissement. Comme nous l'avons suggéré précédemment, cette différence s'explique par le fait qu'UDP permet de jeter des messages. Cette expérience simple illustre l'intérêt d'utiliser un échantillonneur pour sélectionner une partie du flux. Il permet d'utiliser le parallélisme de pipeline sans payer un coût trop important vis à vis de la latence.

12.4 Efficacité des calculs

La selection de messages au sein d'un flux permet d'utiliser le parallélisme de pipeline tout en limitant le ralentissement sur la latence. Si un message a été jeté à la dernière étape du pipeline, il a néanmoins été traité par les premières tâches du pipeline. Ces calculs peuvent être considérés comme inutiles car à terme le message est jeté.

Pour illustrer ce phénomène, à partir de l'expérience précédente nous mesurons le temps de calculs sur des données qui seront à terme jetées par les échantillonneurs. La figure 12.6 donne la somme sur 2 minutes des temps de calculs sur les données qui n'ont pas atteint le bout du pipeline. Nous additionnons les temps de calculs sur l'ensemble des threads de calcul durant l'expérience. Il n'est donc pas anormal d'obtenir des valeurs supérieures à 2 minutes qui est la durée de l'expérience.

Dans cette expérience, le goulot d'étranglement se situe en fin de pipeline. Par conséquent, de nombreux messages ont traversé les premières étapes du pipeline avant d'être jetés. Cette situation produit le maximum de calculs inutiles. Il met en valeur l'intérêt de mettre en place un mécanisme de régulation.

Sur cette simple expérience, la majorité des calculs effectués ont été inutiles. Les ressources mobilisées pour ces calculs ne peuvent être ici affectées à d'autres tâches. Si nous prenons une application représentée par un graphe non trivial,

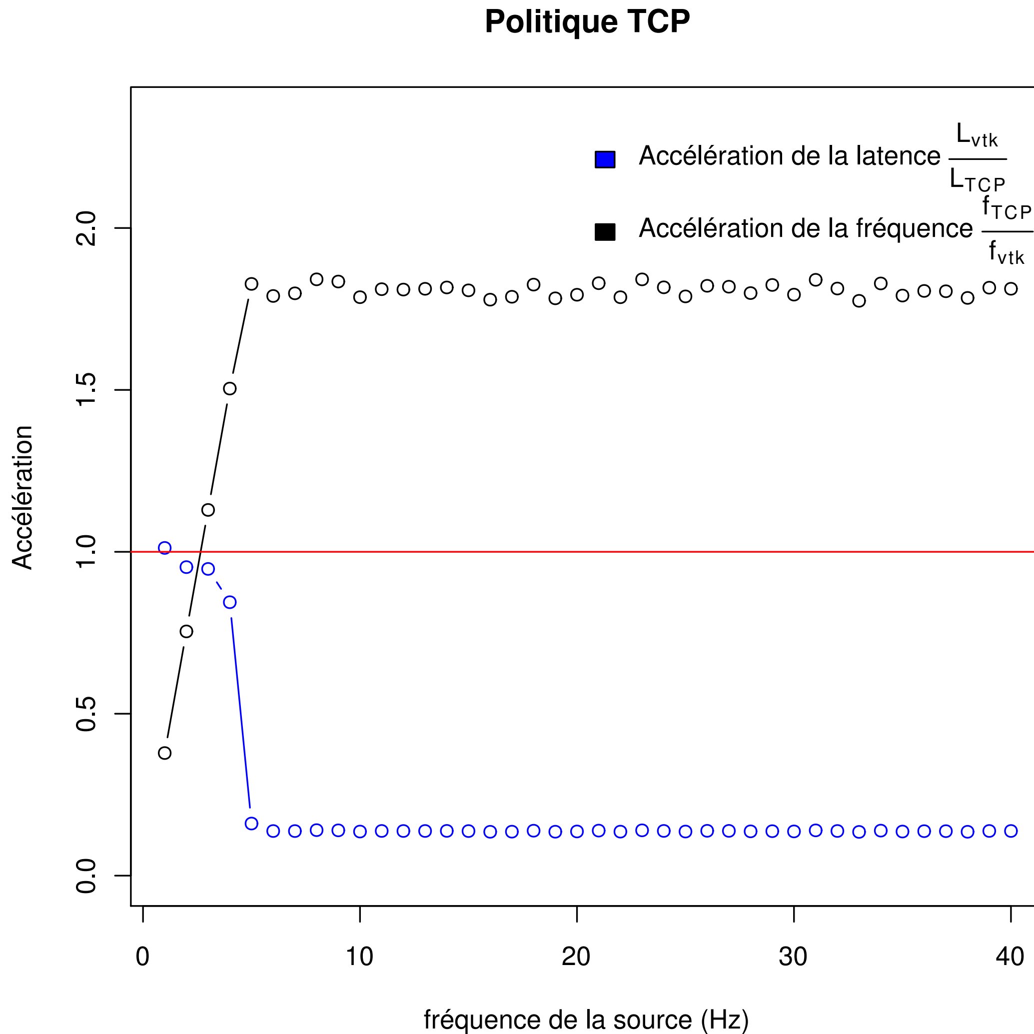


FIGURE 12.4 – Accélération de la politique TCP par rapport à VTK. La courbe noire correspond à l'accélération en terme de débit. La courbe bleue correspond à l'accélération de la latence. Les valeurs sont fonction de la fréquence maximum du composant en entrée de pipeline.

les pipelines qui composent l'application consommeront donc des ressources et ne pourront pas être utilisées pour d'autres formes de parallélisme. Nous aurons tendance à ne pas utiliser efficacement la machine.

Ces calculs inutiles génèrent de plus des données qui ne contribueront pas au résultat final de l'application. Dans le cadre d'une application distribuée, ces données nécessiteront de la ressource réseau pour transiter entre les différentes

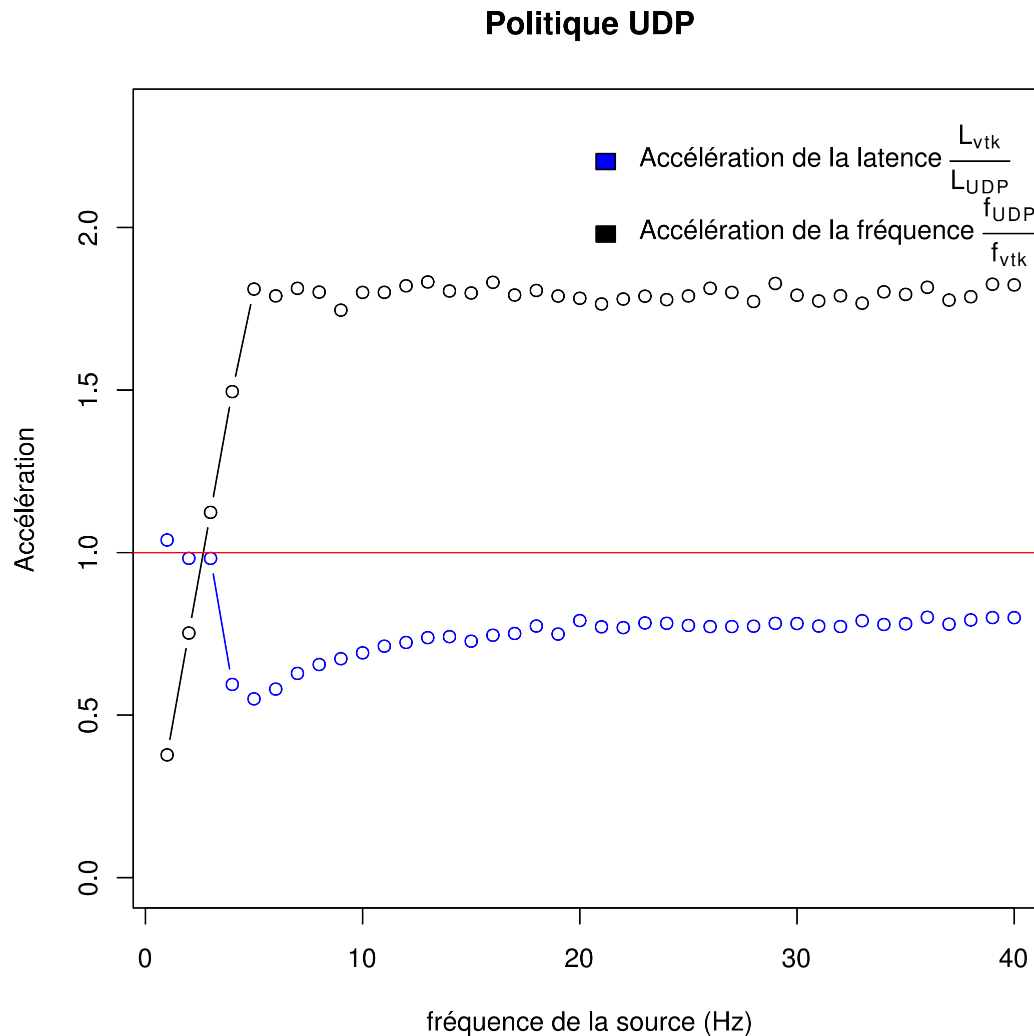


FIGURE 12.5 – Accélération de la politique UDP par rapport à VTK. La courbe noire correspond à l'accélération en terme de débit. La courbe bleue correspond à l'accélération de la latence. Les valeurs sont fonction de la fréquence maximum du composant en entrée de pipeline.

tâches. Le réseau étant une ressource critique partagée par toute l'application, ces calculs inutiles peuvent perturber l'application et dégrader les performances. Pour réaliser des interactions complexes, ces calculs inutiles vont demander l'utilisation de machines puissantes. La diminution des calculs inutiles diminuera la taille des machines nécessaires pour utiliser les applications.

Le partage efficace des ressources réseau est un problème actuellement étudié

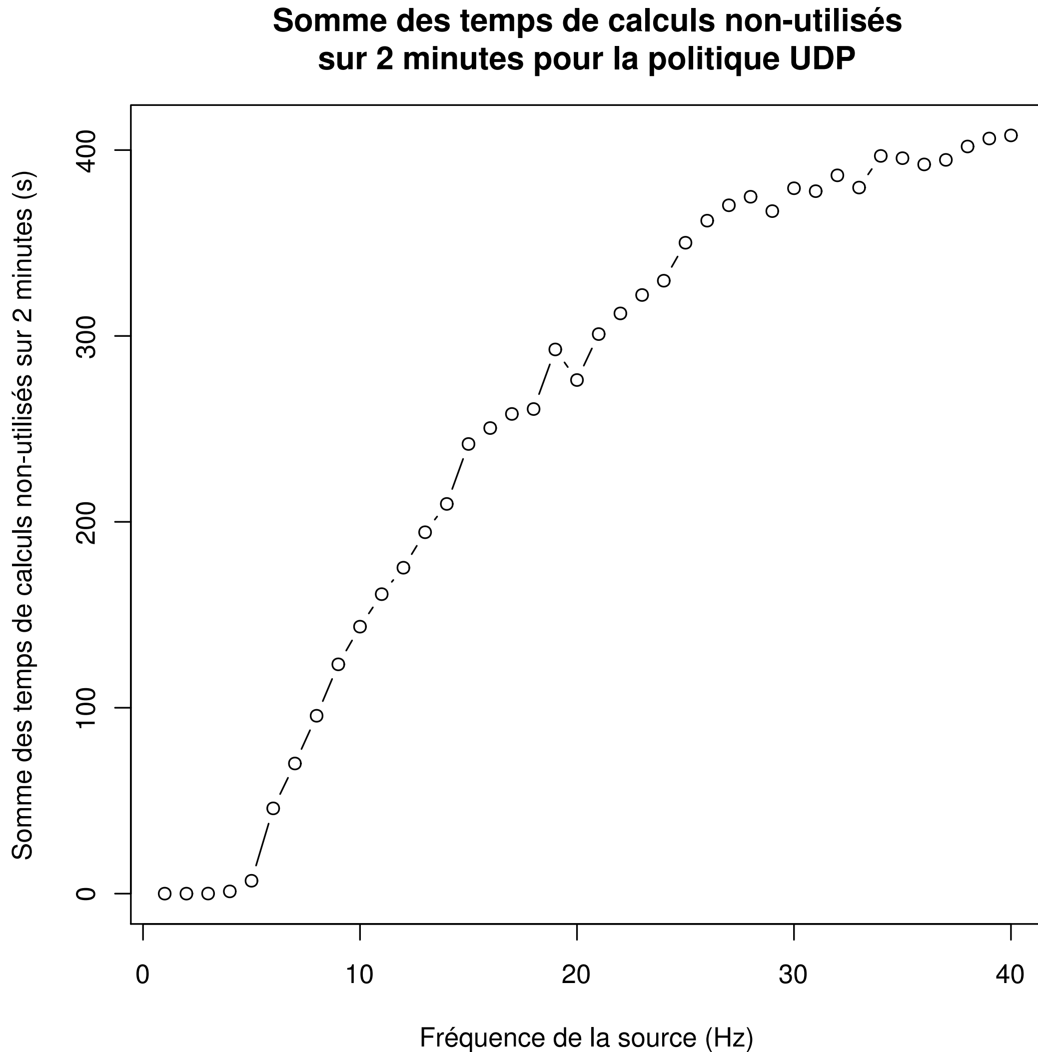


FIGURE 12.6 – Somme sur 2 minutes d'exécution des temps de calculs sur des données qui n'ont pas atteint le bout du pipeline.

par les communautés liés aux transferts de données multimédia. Les protocoles de qualité de service comme RTCP [76] sont une réponse à ce problème. De notre point de vue, le problème du partage des ressources de calculs est plus spécifiques aux applications interactives. Les applications interactives sont souvent représentés par des graphes de flots de données complexes. Il y a donc de grandes chances qu'au sein de la même application plusieurs pipelines s'exécutent sur les même machines. Le problème de régulation devient alors critique car nous venons de montrer par un exemple simple qu'un mécanisme de régulation est nécessaire pour partager les

ressources efficacement.

Il existe peu de résultats de régulation sur les applications interactives. Dans un premier temps, il serait possible d'étudier les algorithmes de qualité de service qui servent à partager les ressources reseau. Par exemple, Visit [114] pour la visualisation scientifique met en place un système de contrat entre tâche VTK. A chaque pipeline est associé un contrat. Les tâches modifient le contrat en indiquant quelles optimisations sont compatibles avec le traitement de la tâche. Le contrat résume donc l'ensemble des optimisations qui peuvent être appliquées sur les données en entrée du pipeline. Ce système est une première forme de régulation car permet d'éviter des optimisations inutiles sur les données.

Pour le problème du partage de ressources dans un cadre plus général que les applications interactives, comme nous l'avons signalé dans l'état de l'art, des algorithmes adaptatifs existent. Par exemple, [71] est un algorithme d'allocation de ressources dynamique et adaptatif. L'adaptation est possible grâce à une information sur l'usage des ressources. Si l'algorithme détecte une tâche parallèle n'utilisant pas efficacement les ressources disponibles, il diminuera alors la quantité de ressources affectées à cette tâche.

12.5 Bilan

Nous avons montré dans ce chapitre que l'utilisation du parallélisme de pipeline permet d'accélérer le débit des résultats. Ce parallélisme peut-être mis en place au niveau du middleware sans modification préalable des tâches de calcul.

Un mécanisme de perte de messages est nécessaire pour ne pas payer une latence trop importante. Un protocole comme UDP qui ignore des données quand la file de messages d'une connexion est pleine permet de réduire nettement l'impact du parallélisme de pipeline sur la latence de l'application. Cependant, ce mécanisme génère des calculs inutiles. Ceux-ci utilisent des ressources de calculs et peuvent générer du débit réseau. En surchargeant ces ressources, les performances générales de l'application peuvent se dégrader. Pour ces raisons il est nécessaire de coupler ce mécanisme d'échantillonnage avec des protocoles de régulation.

Nous avons proposé quelques pistes pour implémenter des protocoles de régulation. Le travail en cours consiste à adapter et tester sur des applications interactives des protocoles de régulation existant dans d'autres communautés scientifiques. Ce travail permettra d'écrire des middlewares qui utiliseront efficacement le parallélisme de pipeline dans les applications interactives.

Chapitre 13

Discussion

13.1 Choix des fréquences

Nous avons étudié le *Synchronization lag*. Cette latence apparait au niveau des échantillonneurs entre deux tâches fonctionnant à des fréquences indépendantes. Nous avons montré qu'un choix pertinent des fréquences permet de limiter l'impact de cette latence. Cette solution n'est pas intuitive. La première intuition que nous retrouvons dans la littérature discutant les problèmes de réactivité dans les applications interactives consiste à exécuter les tâches le plus rapidement possibles pour réduire la latence.

Nous avons vu que si les fréquences des tâches sont indépendantes, le *synchronization lag* est proportionnel au temps d'itération des tâches. Ce phénomène s'observe facilement. Ce sont probablement ces observations qui ont permis à Wloka de formuler sa conjecture. Si nous nous basons uniquement sur ces observations, nous voyons uniquement la relation entre la latence et la fréquence des tâches. L'ensemble mène naturellement à la solution qui consiste à exécuter les tâches le plus rapidement possibles.

En étudiant formellement le *synchronization lag*, nous avons pu identifier la condition nécessaire à la conjecture de Wloka. Dans le cas où les fréquences sont multiples entre elles, il n'y a pas de rapport de proportionnalité entre la latence et le temps d'itération des tâches. Ce résultat mène à une nouvelle solution. Il suffit que les fréquences ne soient pas indépendantes pour régler le problème du *synchronization lag*.

Notre solution a deux avantages. Tout d'abord, nous montrons que nous réglons le problème de la gigue. La latence est constante à la sortie de l'échantillonneur. Ensuite, notre solution est beaucoup plus simple à mettre en oeuvre. Accélérer les fréquences des tâches est un processus complexe qui nécessite d'optimiser les algorithmes. Notre alternative propose au contraire de ralentir les tâches, cela ne

demande aucune modification des algorithmes. Ce ralentissement peut être réalisé automatiquement par un middleware qui ordonnance les tâches. L'autre solution demande de modifier le code des tâches, ce qui doit être fait obligatoirement par un humain.

13.2 Un énoncé du problème de couplage

Les résultats des chapitres 10 et 11 nous ont permis de transformer le problème de couplage en un problème de choix de fréquences.

A partir de ces résultats, nous pouvons commencer à explorer des méthodes permettant de coupler efficacement les tâches. Avant d'étudier ces solutions, nous devons décrire le problème de couplage. Nous énonçons maintenant une modélisation simple du problème qui reprend les résultats de cette partie.

Nous avons une application. Le modèle pour la décrire que nous avons présenté dans l'état de l'art est le suivant : Soit $G = (V, E)$, un DAG (graphe dirigé non cyclique) représentent les flux de données dans une application interactive. V sont les noeuds et représentent les tâches. E représentent les communications. Pour chaque tâche M_i dans V, nous définissons le coût μ_i de la tâche en opérations par itération. Nous considérons le sous-ensemble de E des connexions munies d'un échantillonneur. Nous appellerons E_{sample} ce sous-ensemble. E_{FIFO} sera le sous-ensemble complémentaire rassemblant les connexions dépourvues d'échantillonneur.

Nous voulons définir le problème. Nous ne proposons pas de le résoudre dans cette partie. Nous n'allons pas non plus prouver des propriétés à partir de cette énoncé. Par conséquent, nous prenons volontairement une modélisation simple pour ne pas surcharger l'énnoncé du problème. Nous considérons que l'architecture est composée de N processeurs homogènes. Nous ne considérons pas le coût des communications réseaux. Chaque machine a une vitesse v_j (exprimée en opérations par seconde).

Le problème de couplage peut se ramener à un problème où nous cherchons le placement des tâches et leurs fréquences d'exécution. Nous notons f_i la fréquence de la tâche i. Nous notons π le placement des tâches. Pour une machine j, π_j est l'ensemble des tâches sur la machines j.

Toutes les tâches doivent être placées sur une machine et une seule machine :

$$\begin{cases} \forall j, k \quad \pi_j \cap \pi_k & = \emptyset \\ \bigcup_j \pi_j & = V \end{cases}$$

La capacité de calculs des machines apportent une contrainte sur la fréquence maximale des tâches :

$$\forall j, \sum_{i \in \pi_j} f_i.\mu_i \leq v_j$$

Nous ajoutons la contrainte sur la fréquence des tâches autour des échantillonneurs. Cette contrainte ($\tau_d \mod \tau_s = 0$) est donnée dans le chapitre 11. En prenant les fréquences, cette contraintes se traduit par $f_s \mod f_d = 0$ avec f_s la fréquence de la source et f_d la fréquence de la destination. Dans notre modèle, cette contrainte sur la fréquence se traduit :

$$\forall (s,d) \in E_{sample}, f_s \mod f_d = 0$$

Sur les connexions sans échantillonneurs, il n'y a pas de perte ni de création de messages. Pour que les tampons liés aux connexions n'explosent pas, les fréquences des tâches le long de ces connexions doivent être égaux :

$$\forall (s,d) \in E_{FIFO}, f_s = f_d$$

L'ensemble des contraintes que nous venons d'énumérer définit un ensemble de placements et de fréquences possibles. Nous avons montré que si l'application utilise un de ces placements, il n'y aura pas de gigue créée par les échantillonneurs. Si nous souhaitons écrire un problème d'optimisation, il reste à définir des métriques à optimiser. Le chapitre 12 à partir de résultats préliminaires montre que les calculs inutiles peuvent pénaliser les performances d'un système interactif. Les résultats de ce chapitre pourraient être utilisés pour motiver une métrique mesurant le nombre de données jetés par seconde.

13.3 Démarches scientifiques pour résoudre le problème de couplage

Nous venons de proposer un énoncé du problème de couplage des composants. Nous avons présenté dans le chapitre 4 des problèmes d'ordonnancement comme l'ordonnancement cyclique et l'ordonnancement de pipeline. La principale différence entre notre énoncé et ces problèmes vient des tâches pouvant fonctionner à des fréquences différentes. Notre problème a les deux spécificités suivantes :

- En plus du placement, nous cherchons la fréquence des tâches. L'espace des solutions est donc plus grand.
- Les fréquences des tâches le long des connexions comportant un échantillonneur doivent respecter la contrainte $f_s \mod f_d = 0$.

Notre problème peut être vu comme une extension des problèmes classiques d'ordonnancement. Nous pouvons reprendre une partie de l'analyse du chapitre 4. Il existe deux visions pour traiter ce problème. La première consiste en la recherche d'un ordonnancement hors-ligne. Pour l'ordonnancement de pipeline, nous avons cité les résultats de Anne Benoit [81] qui correspond à cette approche. La thèse de

Sylvain Jubertie portait sur le problème de couplage dans les applications interactives. La thèse propose une résolution du problème de couplage en se basant sur la programmation par contraintes [83]. L'avantage de ces approches est qu'elles permettent de définir et de prouver des propriétés sur le problème. Les résultats de complexité permettent par exemple de décider s'il est envisageable de chercher une solution exacte au problème. Ces résultats font avancer notre connaissance du problème.

La critique de la recherche d'algorithmes hors-ligne est qu'il est difficile de les utiliser dans notre contexte. Certaines données du problèmes varient lors de l'exécution de l'application et il n'est pas possible de les prévoir. Nous avons cité précédemment l'exemple du coût des tâches. Ce coût peut varier. C'est le cas de nombreux algorithmes de rendus dont la complexité est fonction du nombre d'objets dans la scène virtuelle. Il n'est pas possible de prévoir le coût car il dépend du contexte d'utilisation.

La seconde approche consiste à chercher des algorithmes adaptatifs ou des protocoles dynamiques de choix des fréquences des tâches. Ces algorithmes sont adaptés à notre contexte. Les algorithmes de qualité de service permettent de réguler la fréquence des tâches. Ils sont déjà utilisé dans le domaine pour la diffusion de médias sur internet. RTCP [76] est un exemple de protocole de régulation. Ces solutions peuvent être intégrées dans des middlewares et être utilisées sur des applications de grande taille.

Pour s'assurer de l'efficacité de ces solutions adaptatives et de la capacité à passer à l'échelle, nous devons mesurer le ratio entre l'efficacité de la réponse et la complexité pour la mettre en oeuvre. Un algorithme distribué simple peut ne pas être plus efficace qu'un choix aléatoire des fréquences des tâches. Nous devons donc étudier formellement ces algorithmes. Cette mesure d'efficacité va nécessiter de se comparer à la réponse optimale au problème de couplage. La réponse optimale au problème sera donnée par les études formelles.

En conclusion, les deux approches sont complémentaires. Nous devons étudier formellement le problème pour être capable de mesurer l'efficacité d'une solution. En pratique, les algorithmes adaptatifs, les heuristiques ou les algorithmes approchés peuvent facilement être intégrés dans des middlewares. Nous devons utiliser une démarche scientifique rigoureuse pour écrire ces algorithmes. Cela permettra la création d'outils robustes.

Quatrième partie

Conclusion

Chapitre 14

Conclusion

14.1 Bilan

Les applications interactives sont composées de tâches itératives s'échangeant des données. Dans cette thèse, nous avons étudié le couplage entre tâches. Dans une première partie, nous avons fait un état de l'art du domaine. Nous avons présenté la modélisation des applications intéractives, certaines problématiques liées au problème de couplage et le problème d'ordonnancement des applications interactives.

Dans une seconde partie, nous avons présenté nos contributions sur le problème de description des applications interactives. Nous avons défini un modèle à composants hiérarchiques. Ce modèle permet l'encapsulation de composants. Ces composants sont ensuite dépliés pour s'adapter à l'architecture cible. Nous avons proposé un algorithme permettant le dépliement des composants hiérarchiques. Cet algorithme ne nécessite pas l'énumération des contraintes pour le dépliement des composants. Nous avons prouvé que la complexité de l'algorithme est polynomiale par rapport au nombre de composants. Nous avons validé ces contributions grâce à une implémentation de notre modèle pour FlowVR.

Dans une dernière partie, nous avons regardé le problème du couplage dans le but d'optimiser les performances. Nous avons réalisé une étude théorique de la latence engendrée par un échantillonneur simple. Notre étude a validé les conjonctures déjà énoncées sur cette latence. Nous avons proposé des solutions pour réduire l'impact des échantillonneurs.

Enfin, dans un dernier chapitre plus prospectif nous avons présenté quelques politiques de couplage. Nous avons montré la nécessité d'introduire des mécanismes de régulation dans les applications interactives.

14.2 Perspectives

L'intêret des grandes applications interactives comme Grimage, Blue-C ne se limite pas uniquement au défi technologique. Leurs concepteurs ont tenté de démontrer qu'en se servant du parallélisme et des technologies nouvelles, nous sommes capables d'expérimenter de nouvelles formes d'interaction. L'objectif est maintenant de donner des outils pour que l'ensemble des développeurs puissent maîtriser et intégrer ces concepts dans leurs applications.

Les applications de grandes tailles mettent à profit l'évolution des architectures pour améliorer l'interactivité. Les futures applications devront utiliser les réseaux hautes-performances, les machines massivement parallèles, les grilles mettant en commun des ressources à l'échelle d'un pays, des unités de calculs spécialisées comme les cartes graphiques, etc. La maîtrise de ces technologies demandera une expertise importante. Le futur processeur *Larrabee* [115] d'Intel sera peut-être une de ces futures technologies. Son architecture est composée de plusieurs dizaines de coeurs de calculs. Un programme utilisant naïvement cette architecture ne sera pas efficace. Il sera nécessaire de prendre en compte de nombreux paramètres comme l'organisation de la mémoire par exemple. Les développeurs devront être assistés pour pouvoir maîtriser l'ensemble de ces paramètres.

Nous devons investir dans le développement de middlewares. Ces middlewares doivent assister le développeur et permettre le passage à l'échelle des applications. Il est nécessaire que ces middlewares soient robustes et mettent en oeuvre des concepts ou des algorithmes idéalement prouvés. Ce sont de nombreuses questions scientifiques que nous devons maintenant resoudre pour construire ces outils.

Les contributions de cette thèse ont permis de mettre en relief quelques unes de ces questions. Le couplage de composants parallèles est un problème complexe. Il existe des schémas éprouvés appelés communications NxM qui implémentent des communications collectives efficaces. Pour les applications interactives, le problème est encore plus complexe car il doit prendre en compte l'échantillonnage. Le projet ANR FVNano [1] de simulation moléculaire est un bon exemple de cette problématique. Une simulation moléculaire parallèle génère la position des atomes d'une molécule qui seront rendus sur des périphériques de rendus parallèles comme un casque de réalité virtuelle. Le couplage des deux composants n'est pas aujourd'hui réalisé efficacement. Les tâches de calculs de la simulation envoient leurs résultats partiels à un composant qui échantillonne et distribue les primitives graphiques vers les périphériques de rendus. Sur des molécules contenant des centaines de milliers d'atomes, ce composant central est aujourd'hui un goulot d'étranglement critique. De nombreuses applications couplent une simulation parallèle avec un rendu parallèle. Nous pouvons considérer que c'est un schéma générique à étudier.

1. http://www.baaden.ibpc.fr/projects/fvnano/

L'objectif à terme serait de proposer une librairie de communications collectives prennant en compte l'échantillonnage. Intégrée à un middleware, elle permettra de coupler efficacement les tâches parallèles dans les applications intéractives.

Dans la dernière partie de la thèse, nous avons montré que l'échantillonnage a un impact sur les performances de l'application. Nous avons montré qu'une des solutions pour réduire l'impact de l'échantillonnage sur la latence est de choisir les fréquences des tâches. Nous avons montré qu'une application peut faire une grande quantité de calculs inutiles si le choix des fréquences est naïf. Un futur travail consisterait donc à étudier les mécanismes de choix de fréquences. Nous avons dans l'état de l'art montré que nous devons utiliser des algorithmes adaptatifs et distribués. Pour réaliser le choix des fréquences, un premier domaine à étudier serait donc les protocoles de régulation et de qualité de service. La communauté de visualisation scientifique a déjà présenté des extensions [114] proposant de la qualité de service dans leurs middlewares. Nous devons chercher à savoir si ces mécanismes peuvent s'adapter à l'ensemble des applications interactives.

Les contributions de cette thèse ont été intégrées au middleware FlowVR. Elles ont pu être utilisées dans les applications de réalité virtuelle Grimage et FVNano. Développer un middleware et en parallèle contribuer à des applications contribuent au travail de recherche. D'un côté, les développements pour FlowVR ont permis de rechercher des solutions génériques aux problèmes scientifiques. De l'autre, le développement d'applications interactives permet de se confronter réellement à ces problèmes. Cette synergie est importante pour réaliser une critique et débattre sur son travail de recherche. Les futures contributions doivent être intégrées dans un middleware. Il est important que ce middleware soit de bonne qualité logicielle pour être utilisé et distribué dans la communauté. Pour que la conception de middlewares respecte une démarche scientifique, il semble primordial de prendre en compte le retour d'expériences des développeurs d'applications interactives.

Le développement de nouvelles formes d'interaction mettra en oeuvre des algorithmes complexes. Pour être exécutées dans un contexte temps réel, il sera nécessaire d'utiliser efficacement les nouvelles architectures parallèles. L'étude du couplage des tâches itératives dans les application interactives deviendra un pré-requis à l'élaboration de nouvelles formes d'interaction.

Bibliographie

[1] Lesage (J.-D.) et Raffin (B.), « A Hierarchical Programming Model for Large Parallel Interactive Applications », dans *IFIP International Conference on Network and Parallel Computing*, vol. 4672 (coll. *Lecture Notes in Computer Science*), p. 516–525, Dalian, China, September 2007. Springer.

[2] Lesage (J.-D.) et Raffin (B.), « High Performance Interactive Computing with FlowVR », dans *IEEE VR 2008 SEARIS workshop*, p. 13–16, Reno, USA, March 2008. Shaker Verlag.

[3] Petit (B.), Lesage (J.-D.), Franco (J.-S.) *et al.*, « Remote and Collaborative 3D Interactions », dans *3dtv*, Germany, May 2009.

[4] Lesage (J.-D.) et Raffin (B.), « A Hierarchical Component Model for Large Parallel Interactive Applications », *Journal of Supercomputing*, July 2008.

[5] Petit (B.), Lesage (J.-D.), Ménier (C.) *et al.*, « Multi-Camera Real-Time 3D Modeling for Telepresence and Remote Collaboration », *International Journal of Digital Multimedia Broadcasting*, 2009. To be Published.

[6] Allard (J.), Lesage (J.-D.) et Raffin (B.), « Modularity for Large Virtual Reality Applications », *Presence*, 2010. To be Published.

[7] Petit B., Lesage J.-D., Franco J.-S., Boyer E. and Raffin B., « Grimage : 3D Modeling for Remote Collaboration and Telepresence », dans *VRST*, Bordeaux, France, October 2008. Demonstrator.

[8] Petit (B.), Lesage (J.-D.), Boyer (E.) et Raffin (B.), « Virtualization Gate », dans *Proceedings of ACM SIGGRAPH 09*, New Orleans, USA, August 2009. Emerging Technology, Demonstrator.

[9] Lesage (J.-D.) et Raffin (B.), « Synchronization Lag due to Generic Samplers », dans *IEEE VR 2010*, 2010. Pending.

[10] Robertson (G.), Card (S. K.) et Mackinlay (J. D.), « The Cognitive Coprocessor Architecture for Interactive User Interfaces », dans *UIST '89 : Proceedings of the 2nd annual ACM SIGGRAPH symposium on User Interface Software and Technology*, p. 10–18, New York, NY, USA, 1989. ACM.

[11] HIPerWall. « Site Web ». http://hiperwall.calit2.uci.edu/.

[12] Cruz-Neira (C.), Sandin (D. J.), DeFanti (T. A.) *et al.*, « The Cave Audio VIsual Experience Automatic Virtual Environement », *Communication of the ACM*, vol. 35, nº 6, 1992, p. 64–72.

[13] C6 CAVE. « Site Web ». http://www.vrac.iastate.edu/c6.php.

[14] Shaw (C.), Green (M.), Liang (J.) et Sun (Y.), « Decoupled Simulation in Virtual Reality with the MR Toolkit », *ACM Trans. Inf. Syst.*, vol. 11, nº 3, 1993, p. 287–317.

[15] Schroeder (W.), Martin (K.) et Lorensen (B.), *The Visualization Toolkit An Object-Oriented Approach To 3D Graphics, 3rd Edition*. Kitware, Inc., 2003.

[16] Foulser (D.), « IRIS Explorer : a Framework for Investigation », *Journal of ACM SIGGRAPH 95*, vol. 29, nº 2, 1995, p. 13–16.

[17] Lucas (B.), Abram (G. D.), Collins (N. S.) *et al.*, « An Architecture for a Scientific Visualization System », dans *IEEE Visualization Conference*, p. 107–114, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[18] « Universal Parallel Computing Research Center (UPCRC), White Paper », 2008. http://www.upcrc.illinois.edu/UPCRC_Whitepaper.pdf.

[19] Top500. « Site Web ». http://www.top500.org.

[20] Tu (T.), Yu (H.), Ramirez-Guzman (L.) *et al.*, « From Mesh Generation to Scientific Visualization : An End-to-End Approach to Parallel Supercomputing », dans *Super Computing*, 2006.

[21] Phillips (J. C.), Braun (R.), Wang (W.) *et al.*, « Scalable Molecular Dynamics with NAMD », *Journal Computing Chemistry*, décembre 2005.

[22] « WoWvx Technology ». Philips 3D Solutions.

[23] Gross. (M.), Würmlin (S.), Naef (M.) *et al.*, « Blue-C : a Spatially Immersive Display and 3D Video Portal for Telepresence », *ACM Transactions on Graphics*, vol. 22, nº 3, 2003, p. 819–827.

[24] Kurillo (G.), Bajcsy (R.), Nahrstedt (K.) et Kreylos (O.), « Immersive 3D Environment for Remote Collaboration and Training of Physical Activities », dans *IEEE Virtual Reality Conference*, p. 269–270, 2008.

[25] Zitnick (C.), Kang (S.), Uyttendaele (M.) *et al.*, « High-Quality Video View Interpolation Using a Layered Representation », dans *Siggraph*, 2004.

[26] Franco (J.-S.) et Boyer (E.), « Efficient polyhedral modeling from silhouettes », *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2008.

[27] Smarr (L. L.), Chien (A. A.), DeFanti (T.) *et al.*, « The OptIPuter », *Communication of the ACM*, vol. 46, nº 11, 2003, p. 58–67.

[28] Koop (D.), Scheidegger (C. E.), Callahan (S. P.) *et al.*, « VisComplete : Automating Suggestions for Visualization Pipelines. », dans *IEEE Transactions on Visualization and Computer Graphics*, vol. 14, p. 1691–1698, 2008.

[29] Würmlin (S.), Lamboray (E.), Staadt (O.) et Gross (M.), « 3D Video Recorder : A System for Recording and Playing Free-Viewpoint Video », *Computer Graphics Forum*, vol. 22, nº 2, 2003, p. 181–193.

[30] Zhang (K.), Damevski (K.), Venkatachalapathy (V.) et Parker (S. G.), « SCIRun2 : A CCA Framework for High Performance Computing », dans *Proceedings of the Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments, HIPS*, vol. 00, p. 72–79, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[31] Bierbaum (A.), Just (C.), Hartling (P.) *et al.*, « VR Juggler : A Virtual Platform for Virtual Reality Application Development », dans *IEEE VR 2001*, Yokohama, Japan, March 2001.

[32] Bruneton (E.), Coupaye (T.), Leclercq (M.) *et al.*, « The FRACTAL Component Model and its Support in Java : Experiences with Auto-Adaptive and Reconfigurable Systems », *Software Practice & Experience*, vol. 36, nº 11-12, 2006, p. 1257–1284.

[33] Baude (F.), Caromel (D.) et Morel (M.), « From Distributed Objects to Hierarchical Grid Components. », dans *CoopIS/DOA/ODBASE*, p. 1226–1242, 2003.

[34] Figueroa (P.), Green (M.) et Hoover (H. J.), « InTml : A Description Language for VR Applications », dans *Web3D'02*, Tempe, Arizona, USA, February 2002.

[35] Annett (M. K.) et Bischof (W. F.), « VR for Everybody : The SNaP Framework », dans *2nd Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*, Lafayette, USA, March 2009.

[36] Rodrigues (F.), Ferraz (R.), Cabral (M.) *et al.*, « Coupling Virtual Reality Open Source Software Using Message Oriented Middleware », dans *2nd Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*, Lafayette, USA, March 2009.

[37] Gamma (E.), Helm (R.), Johnson (R.) et Vlissides (J.), *Design patterns : elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[38] Cole (M.), *Algorithmic Skeletons : Structured Management of Parallel Computation*. MIT Press, 1989.

[39] Mattson (T. G.), Sanders (B. A.) et Massingill (B. L.), *A Pattern Language for Parallel Programming*. Addison Wesley, 2004.

[40] Aldinucci (M.), Coppola (M.), Danelutto (M.) *et al.*, « ASSIST as a research framework for high-performance Grid programming environments », dans Cunha (J. C.) et Rana (O. F.), éditeurs, *Grid Computing : Software environments and Tools*. Springer, janvier 2006.

[41] Serot (J.) et Ginhac (D.), « Skeletons for parallel image processing : an overview of the skipper project », *Parallel Computing*, vol. 28, nº 12, December 2002, p. 1685–1708.

[42] Allard (J.), Gouranton (V.), Lecointre (L.) *et al.*, « FlowVR : a Middleware for Large Scale Virtual Reality Applications », dans *Proceedings of Euro-par 2004*, Pisa, Italia, August 2004.

[43] Margery (D.), Arnaldi (B.), Chauffaut (A.) *et al.*, « OpenMASK : Multi-Threaded or Modular Animation and Simulation Kernel or Kit : a General Introduction », dans Richir (S.), Richard (P.) et Taravel (B.), éditeurs, *VRIC 2002 Proceedings*, p. 101–110, 2002.

[44] Wierse (A.), Lang (U.) et Rhle (R.), « Architectures of Distributed Visualization Systems and their Enhancements », dans *Eurographics Workshop on Visualization in Scientific Computing*, Abingdon, 1993.

[45] Assenmacher (I.) et Kuhlen (T.), « The ViSTA Virtual Reality Toolkit », dans *IEEE VR 2008 SEARIS workshop*, Reno, USA, March 2008. Shaker Verlag.

[46] Springer (J. P.), Lux (C.), Reiners (D.) et Froehlich (B.), « Advanced Multi-Frame Rate Rendering Techniques », dans *IEEE VR*, p. 177–184, 2008.

[47] Azuma (R.) et Bishop (G.), « Improving Static and Dynamic Registration in an Optical See-Through HMD », dans *SIGGRAPH '94 : Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, p. 197–204, New York, NY, USA, 1994. ACM.

[48] Smit (F. A.), van Liere (R.) et Fröhlich (B.), « An Image-Warping VR-Architecture : Design, Implementation and Applications », dans *VRST '08 : Proceedings of the 2008 ACM Symposium on Virtual Reality Software and Technology*, p. 115–122, New York, NY, USA, 2008. ACM.

[49] Shade (J.), Gortler (S.), wei He (L.) et Szeliski (R.), « Layered Depth Images », dans *SIGGRAPH '98 : Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, p. 231–242, New York, NY, USA, 1998. ACM.

[50] Cheng (L.), Bhushan (A.), Pajarola (R.) et Zarki (M. E.), « Real-time 3D Graphics Streaming using MPEG-4 », dans *Proceedings of the*

IEEE/ACM Workshop on Broadband Wireless Services and Applications, 2004.

[51] Watson (B.), Walker (N.), Ribarsky (W.) et Spaulding (V.), « Effects of Variation in System Responsiveness on User Performance in Virtual Environments », dans *Human Factors, Special Section on Virtual Environments*, 1998.

[52] Ahrens (J.), Law (C.), Schroeder (W.) *et al.*, « A Parallel Approach for Efficiently Visualizing Extremely Large Time-Varying Datasets ». Rapport technique, Los Alamos National Laboratory, Los Alamos National Laboratory, 2000.

[53] Thies (W.), Karczmarek (M.) et Amarasinghe (S. P.), « StreamIt : A Language for Streaming Applications », dans *CC '02 : Proceedings of the 11th International Conference on Compiler Construction*, p. 179–196, London, UK, 2002. Springer-Verlag.

[54] Buck (I.), Foley (T.), Horn (D.) *et al.*, « Brook for GPUs : Stream Computing on Graphics Hardware », *ACM Transaction on Graphics*, vol. 23, n° 3, 2004, p. 777–786.

[55] Sugerman (J.), Fatahalian (K.), Boulos (S.) *et al.*, « GRAMPS : A programming model for graphics pipelines », vol. 28, p. 1–11, New York, NY, USA, 2009. ACM.

[56] Mark (W. R.), Glanville (R. S.), Akeley (K.) et Kilgard (M. J.), « Cg : a System for Programming Graphics Hardware in a C-like Language », dans *Proceedings of ACM SIGGRAPH 03*, p. 896–907, New York, NY, USA, 2003. ACM Press.

[57] « OpenGL Shading Language ». http://www.opengl.org/registry/doc/GLSLangSpec.1.50.09.pdf.

[58] Kirk (D.), « NVIDIA CUDA Software and GPU Parallel Computing Architecre », dans *ISMM '07 : Proceedings of the 6th international symposium on Memory management*, p. 103–104, New York, NY, USA, 2007. ACM.

[59] OpenCL. « Site Web ». http://www.khronos.org/opencl/.

[60] Lampe (O. D.), Reuter (N.), Member-Viola (I.) et Member-Hauser (H.), « Two-Level Approach to Efficient Visualization of Protein Dynamics », *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, n° 6, 2007, p. 1616–1623.

[61] Wang (C.), Gao (J.) et Shen (H.-W.), « Parallel Multiresolution Volume Rendering of Large Data Sets with Error-Guided Load Balancing », *Parallel Computing*, vol. 31, n° 2, février 2005, p. 185–204.

[62] Allard (J.), Ménier (C.), Raffin (B.) *et al.*, « Grimage : Markerless 3D Interactions », dans *Proceedings of ACM SIGGRAPH 07*, San Diego, USA, August 2007. Emerging Technology.

[63] Franco (J.) et Boyer (E.), « Exact Polyhedral Visual Hulls », dans *Proceedings of BMVC2003*, 2003.

[64] Franco (J.-S.), Ménier (C.), Boyer (E.) et Raffin (B.), « A distributed approach for real time 3d modeling », dans *Conference on Computer Vision and Pattern Recognition Workshop*, p. 31–31, 2004.

[65] Blumofe (R. D.) et Leiserson (C. E.), « Scheduling Multithreaded Computations by Work Stealing », dans *In Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS*, p. 356–368, 1994.

[66] Bender (M. A.) et Rabin (M. O.), « Online Scheduling of Parallel Programs on Heterogeneous Systems with Applications to Cilk », *Theory of Computing Systems*, vol. 35, nº 3, 2002, p. 289–304.

[67] Gautier (T.), Besseron (X.) et Pigeon (L.), « KAAPI : A Thread Scheduling Runtime System for Data Flow Computations on Cluster of Multi-Processors », dans *PASCO '07 : Proceedings of the 2007 International Workshop on Parallel Symbolic Computation*, p. 15–23, New York, NY, USA, 2007. ACM.

[68] Contreras (G.) et Martonosi (M.), « Characterizing and Improving the Performance of the Intel Threading Building Blocks Runtime System », dans *International Symposium on Workload Characterization (IISWC 2008)*, September 2008.

[69] Traore (D.), Roch (J.-L.), Maillard (N.) *et al.*, « Adaptive Parallel Algorithms and Applications to STL », dans Springer-Verlag, éditeur, *EUROPAR 2008*, Las Palmas, Spain, August 2008.

[70] Soares (L.), Ménier (C.), Raffin (B.) et Roch (J.-L.), « Work Stealing for Time-constrained Octree Exploration : Application to Real-time 3D Modeling », dans *EGPGV*, Lugano,Switzerland, May 2007.

[71] Agrawal (K.), He (Y.), Hsu (W. J.) et Leiserson (C. E.), « Adaptive scheduling with parallelism feedback », dans *PPoPP '06 : Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, p. 100–109, New York, NY, USA, 2006. ACM.

[72] Zhang (L.) et Parashar (M.), « Enabling Efficient and Flexible Coupling of Parallel Scientific Applications », dans *Parallel and Distributed Processing Symposium (IPDPS) 2006.*, April 2006.

[73] Denis (A.), Pérez (C.) et Priol (T.), « Achieving portable and efficient parallel CORBA objects », *Concurrency and Computation : Practice and Experience*, vol. 15, nº 10, August 2003, p. 891–909.

[74] Richart (N.), Esnard (A.) et Coulaud (O.), « Toward a Computational Steering Environment for Legacy Coupled Simulations », dans *Proceedings of 6th International Symposium on Parallel and Distributed Computing (ISPDC 2007)*, p. 319–326, Hagenberg, Austria, July 2007. IEEE Press.

[75] Steed (A.), « A Simple Method for Estimating the Latency of Interactive, Real-Time Graphics Simulations », dans *VRST*, p. 123–129, 2008.

[76] Schulzrinne (H.), Casner (S.), Frederick (R.) et Jacobson (V.). « RFC 3550 : RTP : A Transport Protocol for Real-Time Applications », 2003.

[77] Hanen (C.) et Munier (A.). « Cyclic Scheduling on Parallel Processors : An Overview », 1994.

[78] Lécuyer (A.), Vidal (M.), Joly (O.) *et al.*, « Can Haptic Feedback Improve the Perception of Self-Motion in Virtual Reality ? », *Haptic Interfaces for Virtual Environment and Teleoperator Systems, International Symposium on*, vol. 0, 2004, p. 208–215.

[79] Benoit (A.) et Robert (Y.), « Mapping Pipeline Skeletons onto Heterogeneous Platforms », *Journal of Parallel Distributed Computing*, vol. 68, n° 6, 2008, p. 790–808.

[80] Subhlok (J.) et Vondran (G.), « Optimal Latency-Throughput Tradeoffs for Data Parallel Pipelines », dans *Eighth Annual ACM Symposium on Parallel Algorithms and Architectures (Padua)*, p. 62–71. ACM Press, 1996.

[81] Benoit (A.), *Scheduling Pipelined Applications : Models, Algorithms and Complexity.* Habilitation à Diriger des Recherches, Ecole Normale Supérieure de Lyon, 2009.

[82] Graham (R. L.), « Bounds for Certain Multiprocessing Anomalies », *Bell System Technical Journal*, vol. 45, 1966, p. 1563–1581.

[83] Jubertie (S.), *Modèles et outils pour le déploiement d'applications de Réalité Virtuelle sur des architectures distribuées hétérogènes.* Thèse de doctorat, Université d'Orléans, December 2007.

[84] He (Y.), Ahmad (I.) et Liou (M. L.), « Real-Time Interactive MPEG-4 System Encoder Using a Cluster of Workstations », dans *IEEE Transactions on Multimedia*, vol. 1, p. 217–233, 1999.

[85] Allard (J.), *FlowVR : calculs intéractifs et visualisation sur grappe.* Thèse de doctorat, Institut National Polytechnique de Grenoble, November 2005.

[86] Ménier (C.), *Système de vision temps-réel pour les interactions.* Thèse de doctorat, Institut National Polytechnique de Grenoble, 2007.

[87] Allard (J.) et Raffin (B.), « A Shader-Based Parallel Rendering Framework », dans *IEEE Visualization Conference*, Minneapolis, USA, October 2005.

[88] Limet (S.) et Robert (S.), « FlowVR-VRPN : first experiments of a VRPN/FlowVR coupling », dans *VRST '08 : Proceedings of the 2008 ACM symposium on Virtual reality software and technology*, p. 251–252. ACM, 2008.

[89] Allard (J.), Cotin (S.), Faure (F.) *et al.*, « SOFA : an Open Source Framework for Medical Simulation », dans *Medicine Meets Virtual Reality (MMVR)*, 2007.

[90] Hess (B.), Kutzner (C.), van der Spoel (D.) et Lindahl (E.), « GROMACS 4 : Algorithms for Highly Efficient, Load-Balanced, and Scalable Molecular Simulation », *Journal of Chemical Theory and Computation*, 2008, p. 435–447.

[91] Allard (J.), Gouranton (V.), Lamarque (G.) *et al.*, « SoftGenLock : Active Stereo and Genlock for PC Cluster », dans *EGVE '03 : Proceedings of the Workshop on Virtual Environments 2003*, p. 255–260, New York, NY, USA, 2003. ACM.

[92] Graphviz. « Site Web ». http://www.graphviz.org/.

[93] Auber (D.) et Jourdan (F.), « Interactive Refinement of Multi-scale Network Clusterings », dans *IV '05 : Proceedings of the Ninth International Conference on Information Visualisation (IV'05)*, p. 703–709, Washington, DC, USA, 2005. IEEE Computer Society.

[94] OverView. « Site Web ». http://www.sandia.gov/OverView/.

[95] M.Beazley (D.), « SWIG : an Easy to Use Tool for Integrating Scripting Languages with C and C++ », dans *TCLTK'96 : Proceedings of the 4th Conference on USENIX Tcl/Tk Workshop*, p. 15–15, Berkeley, CA, USA, 1996. USENIX Association.

[96] Baude (F.), Caromel (D.), Henrio (L.) et Morel (M.), « Collective Interfaces for Distributed Components », dans *CCGRID*, p. 599–610. IEEE Computer Society, 2007.

[97] Cai (W.), Lee (F. B. S.) et Chen (L.), « An Auto-Adaptive Dead Reckoning Algorithm for Distributed Interactive Simulation », dans *PADS '99 : Proceedings of the 13th workshop on Parallel and distributed simulation*, p. 82–89, 1999.

[98] Bajura (M.), Fuchs (H.) et Ohbuchi (R.), « Merging Virtual Objects with the Real World : Seeing Ultrasound Imagery within the Patient », dans *SIGGRAPH '92 : Proceedings of the 19th Annual Conference on Computer*

Graphics and Interactive Techniques, p. 203–210, New York, NY, USA, 1992. ACM.

[99] Funkhouser (T.) et Séquin (C. H.), « Adaptive Display Algorithm for Interactive Frame Rates during Visualization of Complex Virtual Environments », dans *SIGGRAPH '93 : Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, p. 247–254, New York, NY, USA, 1993. ACM.

[100] Cherniack (M.), Balakrishnan (H.), Balazinska (M.) *et al.*, « Scalable Distributed Stream Processing », dans *Proceedings of the 2003 CIDR Conference*, 2003.

[101] Abadi (D. J.), Ahmad (Y.), Balazinska (M.) *et al.*, « The Design of the Borealis Stream Processing Engine », dans *2nd Biennial Conference on Innovative Data Systems Research (CIDR'05)*, p. 277–289, 2005.

[102] Chandrasekaran (S.), Ch (S.), Cooper (O.) *et al.*, « TelegraphCQ : Continuous Dataflow Processing for an Uncertan World », dans *Proceedings of the Conference on Innovative Data Systems Research*, 2003.

[103] Lien (J.-M.), Kurillo (G.) et Bajcsy (R.), « Multi-Camera Tele-Immersion System with Real-Time Model Driven Data Compression », *The Visual Computer*, may 2009.

[104] Mine (M.), « Characterization of End-to-End Delays in Headmounted Display Systems ». Rapport technique, Department of Computer Science, University of North Carolina at Chapel Hill, 1993.

[105] Wloka (M. M.), « Lag in Multiprocessor Virtual Reality », *Presence*, vol. 4, 1995, p. 50–63.

[106] Liang (J.), Shaw (C.) et Green (M.), « On Temporal-Spatial Realism in the Virtual Reality Environment », dans *UIST '91 : Proceedings of the 4th annual ACM symposium on User Interface Software and Technology*, p. 19–25, New York, NY, USA, 1991. ACM.

[107] Link (N.), Kruk (R.), McKay (D.) *et al.*, « Hybrid Enhanced and Synthetic Vision System Architecture for Rotorcraft Operations », dans Verly (J. G.), éditeur, *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, vol. 4713 (coll. *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*), p. 190–201, juillet 2002.

[108] Olano (M.), Cohen (J.), Mine (M.) et Bishop (G.), « Combatting Rendering Latency », dans *SI3D '95 : Proceedings of the 1995 symposium on Interactive 3D graphics*, p. 19–ff., New York, NY, USA, 1995. ACM.

[109] Bishop (G.), Fuchs (H.), McMillan (L.) et Zagier (E. J. S.), « Frameless rendering : double buffering considered harmful », dans *Proceedings of the*

21st Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1994, p. 175–176, 1994.

[110] Smit (F.), van Liere (R.), Beck (S.) et Froehlich (B.), « An Image-Warping Architecture for VR : Low Latency versus Image Quality », dans *IEEE VR 2009*, p. 27–34, Lafayette, March 2009.

[111] Knuth (D. E.), *Art of Computer Programming, Volume 2 : Seminumerical Algorithms (3rd Edition).* Addison-Wesley Professional, 1997.

[112] Lorensen (W. E.) et Cline (H. E.), « Marching cubes : A high resolution 3D surface construction algorithm », dans *Proceedings of ACM SIGGRAPH 87*, vol. 21, p. 163–169, New York, NY, USA, 1987. ACM.

[113] Drebin (R. A.), Carpenter (L.) et Hanrahan (P.), « Volume rendering », dans *Proceedings of ACM SIGGRAPH 88*, vol. 22, p. 65–74, New York, NY, USA, 1988. ACM.

[114] Childs (H.), Brugger (E.), Bonnell (K.) *et al.*, « A Contract Based System For Large Data Visualization », dans *IEEE VIS '05 : Proceedings of the Conference on Visualization '05*, vol. 0, p. 25, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[115] Seiler (L.), Carmean (D.), Sprangle (E.) *et al.*, « Larrabee : A Many-Core x86 Architecture for Visual Computing », dans *Proceedings of ACM SIGGRAPH 2006*, 2006.