II

FlowVR

Après un chapitre d'expérimentations préliminaires, cette partie présente la contribution principale de cette thèse : un modèle de couplage de composants parallèles ainsi que l'implantation associée pour construire des applications interactives. L'originalité repose sur la possibilité d'exprimer des motifs de couplages avancées, permettant de désynchroniser les différentes parties de l'application tout en respectant les contraintes de cohérence exigées par l'utilisateur. Les connexions entre composants ainsi que ces contraintes de cohérence sont définies par un graphe de flux de données, comportant des objets particuliers implantant les opérations de filtrage et de synchronisations nécessaires. Ces considérations étant extérieures aux code des composants, ce modèle permet une bonne réutilisabilité des composants ainsi qu'une construction très modulaire des applications.

Ce travail a été publié lors de la conférence Euro-Par 2004 [9]. Les applications et expérimentations développées ont fait l'objet de publications à IEEE VR 2002 [10], Euro-Par 2003 [14] et IPT/EGVE 2002 [11], 2003 [8], 2004 [5] et 2005 [6].

Travaux préliminaires 5

Dans le cadre de mon stage de maîtrise au laboratoire *LIFO* de l'Université d'Orléans, des premiers travaux ont été effectués pour pouvoir exécuter des applications de réalité virtuelle en environnement multi-projecteurs piloté par grappe de PC utilisant des composants standards. Ceci a abouti à un ensemble d'outils offrant une solution à ce problème. Ce fut l'une des premières disponibles. Plusieurs publications [7, 8, 10, 11, 14] ont contribué à la diffusion de ce travail.

Le résultat de ce travail a servi de base pour le début de cette thèse, avec des premières expérimentations testant le couplage d'une simulation parallèle et interactive à de telles applications.

5.1 Net Juggler : duplication transparente d'applications VR Juggler

Dans le cadre de la réalité virtuelle, une grappe de PC sert tout d'abord à piloter les multiples vidéo-projecteurs nécessaires à la surface d'affichage. Pour cela, chaque PC est relié à un ou deux projecteurs et il faut donc qu'il calcule l'image correspondante. Plusieurs techniques sont possibles (section 4.1 page 28), la plus simple à implanter et la plus performante est la *duplication* de l'application sur chaque machine de rendu. C'est cette méthode que nous avons exploré dans le cadre de *Net Juggler* [10].

5.1.1 Exécution sur grappe

VR Juggler 1.0 (section 4.2.2 page 31), qui était la version disponible au moment de ce travail, ne supportait pas l'exécution d'une application sur une grappe. Net Juggler ajoute donc ce support en dupliquant VR Juggler et l'application sur chaque machine et en utilisant l'abstraction des périphériques d'entrée fournie par VR Juggler pour les synchroniser entre toutes les machines de manière transparente (*datalock*). Les communications ainsi que le lancement de l'application utilisent MPI (section 3.1.1 page 18).

Le système de configuration de VR Juggler est modifié pour ajouter une information de placement de chaque périphérique, tels que les périphériques d'entrée ou les projecteurs. Ensuite chaque machine calcule l'image correspondant au projecteur local. Juste avant d'afficher chaque nouvelle image, une barrière est utilisée afin de synchroniser les différents projecteurs (*swaplock*).

Du fait de l'organisation interne de VR Juggler sous forme de micro-noyau faisant appel à un ensemble de *managers* pour gérer chaque fonctionnalité, les modifications requises par Net Juggler furent assez simple à implanter. En revanche cette approche impose deux limitations importantes :

- Toutes les informations ayant une influence sur l'état de l'application doivent être diffusées à l'ensemble des machines. Les données d'entrée sont gérées par Net Juggler, mais d'autres informations comme l'horloge ou le générateur pseudo-aléatoire peuvent introduire des différences entre machines. Pour les corriger il est possible de les transformer en périphérique d'entrée. Ainsi Net Juggler fourni un périphérique d'horloge globale.
- Du fait de la barrière de *swaplock*, la vitesse de l'application dépend de la machine la plus lente. Si toutes les machines sont identiques et comme les mêmes calculs sont dupliqués sur chaque machine (mise à part la modification de point de vue lié au projecteur), alors les performances sur la grappe seront proches des performances obtenues sur une seule machine (affichant sur un seul projecteur), moins le surcoût des communications réseau (en général très faible).

5.1.2 Résultats

Du fait de la faible quantité d'informations à transmettre entre les machines (généralement quelques matrices et évènements d'appuis sur des boutons), le surcoût réseau est très faible, ce qui fait que les performances obtenues sont proches de celles observées sur une seule machine. L'avantage est que les performances sur la grappe sont faciles à anticiper en testant sur une seule machine, en revanche cela ne permet pas d'accélérer une application trop exigeante, mais uniquement d'augmenter la résolution (le nombre de projecteurs).

La figure 5.1 présente l'application CAVE Quake III Arena [144] exécutée sur une grappe de PC grâce à Net Juggler. Les performances obtenues sur une machine sont de

Figure 5.1 *CAVE Quake III Arena exécuté sur une grappe grâce à Net Juggler et SoftGenLock [8].*

38 images par secondes. Sur 4 machines avec un réseau Myrinet Net Juggler obtient 37 images par secondes. Le surcoût lié à l'exécution sur la grappe est donc assez faible.

Pour le cas des applications coûteuses en calculs, du fait de l'utilisation de MPI par Net Juggler il est relativement facile de l'utiliser aussi pour répartir les calculs de l'application. Ceci permet de développer des applications intégrant des simulations interactives. Une application exemple est décrite dans la section 5.2.1.

5.2 Simulations interactives sous Net Juggler

Intégrer une simulation parallèle, généralement d'un phénomène physique, dans une application de réalité virtuelle peut être utile pour deux objectifs différents. Cela permet d'explorer les résultats de la simulation et ainsi mieux les comprendre, avec parfois la possibilité d'influer sur la simulation de manière interactive. Mais la simulation peut aussi être intégrée à un environnement virtuel de manière à le rendre plus dynamique et plus riche, donc plus immersif. Dans les deux cas les problèmes sont assez similaires, liés au couplage entre la partie simulation et la partie visualisation. Mais l'accent est mis soit sur la précision de la simulation soit sur ses performances et sur la latence de la boucle d'interaction.

5.2.1 Simulation de fluide 2D interactive

Une première approche pour intégrer une simulation parallèle à une application Net Juggler est l'utilisation de l'environnement MPI déjà intégré. C'est l'approche retenue pour l'une des premières applications développées, intégrant une simulation de fluide

Figure 5.2 *Rendu de l'application de simulation de fluide 2D interactive.*

parallélisée interactive [11].

La simulation de fluide est basée sur l'algorithme de résolution des équations de Navier-Stokes proposé par Stam [165, 166]. L'espace est discrétisé par une grille de cellules. Chaque cellule possède un vecteur vitesse du fluide ainsi qu'une valeur de densité caractérisant le fluide présent dans la cellule. A chaque pas de simulation, l'algorithme met à jour ces valeurs. Ce calcul utilise des opérations matricielles simples ainsi qu'un gradient conjugué et une résolution des équations de Poisson.

L'implantation utilise PETSc [20], une librairie mathématique fournissant des opérations matricielles parallélisées par MPI. Toutes les matrices sont réparties sous forme de blocs sur les différentes machines, et les valeurs aux frontières sont communiquées après chaque calcul.

La plupart des implantations de MPI ne supportant pas le *multi-threading*, la simulation est placée dans le même thread que la visualisation. De ce fait, elles sont exécutées de manière synchrone, c'est-à-dire que leur fréquence de rafraîchissement est identique. Cette approche est surtout adaptée au cas de l'intégration d'une simulation de relativement faible échelle dans un environnement virtuel. Une simulation trop coûteuse serait un obstacle à l'interactivité de la visualisation.

Cette simulation de fluide, intégrée dans un environnement virtuel pour former un lac dans une vallée, est visible sur la figure 5.2. L'utilisateur interagit en déplaçant un pointeur qui applique des forces au fluide.

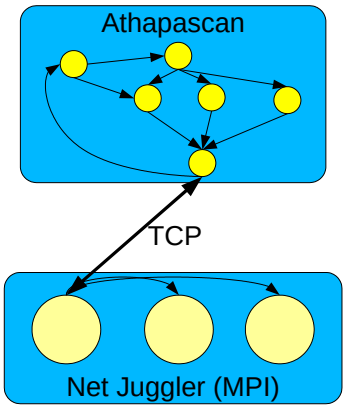


Figure 5.3 *Schéma de couplage entre Net Juggler et Athapascan.*

5.2.2 Simulation de tissus préexistante

Il existe souvent des codes de simulation déjà opérationnels pour des calculs non-interactifs et il semble très intéressant de les réutiliser dans un environnement interactif. Pour tester cette approche, nous avons intégré le travail de Florence Zara sur la simulation de tissus [184, 185] avec une visualisation distribuée par Net Juggler. Cette simulation utilise Athapascan (présenté dans la section 3.1.3 page 20) pour distribuer le calcul. Du fait de l'utilisation de deux paradigmes de parallélisation différents entre la visualisation (répartition statique avec passage de messages MPI) et la simulation (ordonnancement et répartition dynamiques gérés par Athapascan), ces deux parties sont implantées séparément.

Le tissus est discrétisé en un maillage de particules reliées par des ressorts. Pour paralléliser le calcul, ce maillage est partitionné en blocs qui sont ensuite répartis sur les processeurs de manière automatique par Athapascan. Les positions sont mises à jour à chaque pas de simulation par une intégration des équations de Newton.

Le couplage entre la simulation et la visualisation utilise un simple canal TCP, comme indiqué sur la figure 5.3. Ceci implique une centralisation des données vers le noeud qui possède le canal du coté de la simulation, et ensuite une diffusion vers tous les noeuds de visualisation des données reçues. En retour les mouvements de la souris sont envoyés à la simulation et servent à bouger un coin du drap de manière interactive.

Contrairement à l'exemple du fluide, il est possible d'introduire de l'asynchronisme entre la simulation et la visualisation, par exemple en prenant en compte à chaque image uniquement le dernier résultat disponible. Ceci permet de supporter des calculs plus complexes du côté de la simulation.

Cette approche est très simple mais a donné de bons résultats que nous avons présentés lors d'une démonstration interactive pour la conférence Euro-Par 2003 [14]. La figure 5.4 présente une capture d'écran de cette application.

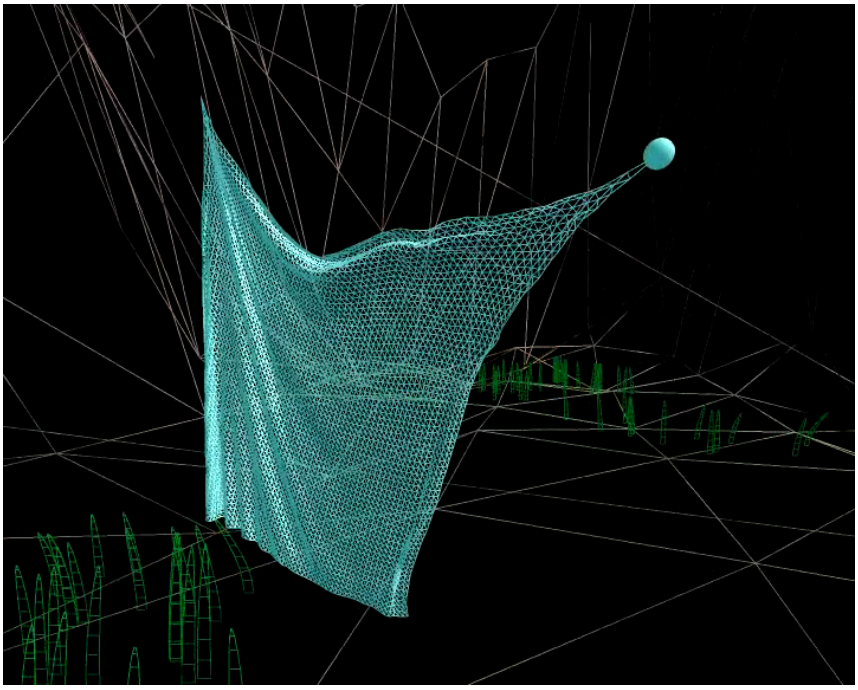


Figure 5.4 *Rendu de l'application de simulation de drap interactive.*

5.3 Bilan

Ce travail a abouti à plusieurs applications couplant une visualisation multi-projecteurs avec plusieurs simulations parallèles. Les performances obtenues sont suffisantes pour une bonne interactivité avec chaque simulation. Les deux applications présentées (simulation de fluide et de tissus) ont même abouti à une seule application intégrant les deux simulations dans le même environnement virtuel. Une vidéo démontrant ce résultat est disponible : `http://www-id.imag.fr/~allardj/these/valley2003.avi`.

Toutefois un certain nombre de problèmes sont apparus. Tout d'abord l'implantation du couplage est spécifique à chaque application. Il doit donc être réécrit à chaque fois qu'un nouvel élément est ajouté. De plus il est difficile d'augmenter considérablement la taille des simulations. En effet, pour le cas du fluide une simulation plus complexe ralentirait d'autant la visualisation. Pour le tissus la liaison vers la visualisation par un canal TCP introduit une centralisation des données qui limite le passage à l'échelle.

Enfin, un autre problème important apparaît lorsqu'on introduit un couplage asynchrone. En effet, le fait de ne plus mettre à jour tous les objets de l'environnement à la même fréquence introduit des décalages entre les objets qui devraient être associés, comme le pointeur de la souris et le coin du drap (figure 5.5). Ce problème est lié aux contraintes de cohérence attendues par l'utilisateur entre les différents objets de l'environ-

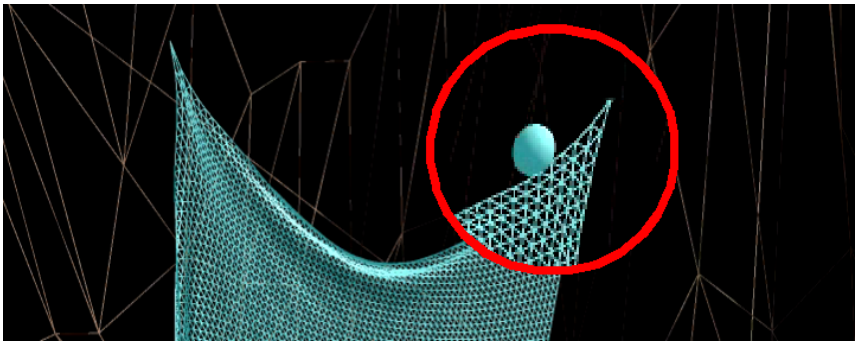


Figure 5.5 *Problèmes de cohérence de la scène.*

nement. Plusieurs approches peuvent être utilisées pour le résoudre, comme par exemple retarder la mise à jour du pointeur de la souris pour être synchronisé avec le pas de temps de la simulation du tissus (au détriment de la latence des mouvements de souris), ou encore afficher un lien entre le pointeur et le coin du drap pour matérialiser un ressort les reliant (et ainsi assouplir la contrainte de liaison entre ces objets). Néanmoins ces modifications sont spécifiques à l'application et ne répondent pas au problème fondamental des contraintes de cohérence dans les systèmes distribués interactifs.

Un modèle d'application distribuée interactive 6

"IF COMPUTERS GET TOO POWERFUL, WE CAN ORGANIZE THEM INTO A COMMITTEE – THAT WILL DO THEM IN."
Bradley's Bromide

6.1 Introduction

Les expériences présentées dans le chapitre 5 permettent de dégager la problématique de notre travail, à savoir le besoin d'un outil générique permettant le couplage de composants parallèles à l'intérieur d'une application distribuée et interactive. Pour permettre une recherche des solutions les plus adaptées et de par la diversité des applications et la complexité du problème, ce modèle doit être simple et facilement modifiable. Les performances à l'exécution sont primordiales, en particulier en ce qui concerne la latence des boucles d'interactions. Enfin, et c'est là une spécificité de ces applications, les contraintes de cohérence de la scène virtuelle doivent être intégrées et modifiables en fonction des souhaits de l'utilisateur afin d'adapter les compromis entre synchronisations et performances.

Ce chapitre présente le modèle adopté en discutant les choix effectués par rapport aux différentes options possibles, avant de le détailler plus formellement.

6.2 Le modèle choisi

La base de notre approche est un découpage de l'application en *modules* communiquant entre eux. En fonction de la granularité de ce découpage, chaque *tâche* de l'application (entrées, simulations, visualisation, ...) peut être constituée d'un ou plusieurs

modules. Une tâche peut être parallélisée et réutilise fréquemment un code ou un algorithme préexistant. Le modèle doit donc faciliter leur intégration au reste de l'application, en particulier en limitant les modifications nécessaires.

Le rôle principal du modèle doit être de gérer les flux de données entre les modules en respectant les contraintes de performance et de cohérence de l'application.

Pour respecter les contraintes de performance il faut pouvoir répartir et utiliser efficacement les ressources disponibles (réseau, calculs, mémoire). Pour supporter de données volumineuses et diminuer la latence il faut par exemple réduire au maximum les recopies des données. Pour imposer les contraintes de cohérence il faut introduire certains mécanismes de synchronisation entre les flux de données.

Les caractéristiques de performances (débit, latence) et de synchronisation (cohérence) de l'application sont très liées. Par exemple, en parallélisant une tâche via un découpage en pipeline le débit est amélioré mais la latence peut être augmentée étant donné le surcoût des communications entre chaque étape. De même, une contrainte forte de synchronisation entre plusieurs machines peut entraîner des mises en attentes et donc une perte de débit. Il n'est en général pas possible d'utiliser une même approche pour toutes les applications et notre modèle doit donc permettre de spécifier des techniques différentes en fonction de l'application, des ressources disponibles et des attentes de l'utilisateur.

Pour pouvoir tester les différentes approches et construire rapidement des applications expérimentales, il doit être possible de changer facilement les techniques employées. En particulier certaines tâches peuvent supporter plusieurs modes de parallélisation (répartition des données, répartition des itérations, pipeline). Le modèle doit permettre de changer ce mode pour une tâche de l'application sans que cela n'affecte les autres tâches.

6.2.1 Granularité du découpage

Le niveau auquel s'effectue le découpage de l'application en modules est primordial pour déterminer les fonctionnalités de l'application. On peut distinguer plusieurs niveaux et leurs conséquences sur l'expression d'une boucle de simulation distribuée :

Opération de base : le calcul est exprimé sous forme d'une suite d'opérations matricielles, traitements sur les données, etc. Ces opérations sont parallélisées et réparties sur les machines.

Boucle de calcul (*thread*) : le calcul est exprimé sous forme d'un ensemble de boucles de calcul, chacune affectée à une machine.

Tâche parallèle : la tâche complète est vue comme un seul composant occupant un groupe de machines.

Un découpage de très bas niveau introduit de très nombreux éléments à gérer, ce qui peut permettre d'exploiter pleinement le parallélisme mais entraîne un surcoût important et une contrainte forte pour l'implantation de l'application qui doit être faite en fonction de ce découpage. Des composants de plus haut niveau rendent l'implantation de chaque

composant responsable de sa parallélisation, ce qui nécessite d'utiliser d'autres outils et méthodes pour le faire, mais permet de choisir celui qui convient le mieux.

Du fait de la prédominance des boucles dans une application interactive, cela semble être une base de découpage intéressante. Chaque composant, que nous appellerons *module*, est un calcul itératif exécuté sur une machine donnée. Plusieurs modules peuvent former une tâche parallèle en utilisant des communications internes. Bien que ce regroupement en tâches parallèles est utile comme concept de haut niveau, du point de vue du modèle il est souhaitable de ne pas le faire apparaître, pour traiter de manière similaire un ensemble de modules, qu'il soit lié par des communications internes ou non.

6.2.2 Connaissance des modules sur le reste de l'application

Une fois l'application découpée en modules, ils doivent être combinés suivant les fonctionnalités requises. Cela se traduit par des échanges de données et/ou de commandes entre ces modules. Ces échanges peuvent être fait de manière totalement transparente aux modules, c'est-à-dire qu'ils n'ont aucune connaissance de la provenance ni la destination des données qu'ils utilisent. Il peuvent alternativement êtres conscients des modules avec lesquels ils sont reliés, ce qui peut leur permettre d'adapter au mieux leur comportement, mais introduit une dépendance plus forte entre module. Cette connaissance du reste de l'application peut être uniquement locale (i.e. uniquement les modules directement reliés) ou alors globale, ou chaque module a conscience de tout le reste de l'application et peut donc directement contacter n'importe quel autre module. Cette décision a un impact fort sur l'interdépendance des modules et les fonctionnalités qui leurs sont offertes pour interagir avec l'application.

Pour favoriser la modularité du système ainsi que la simplicité de sa programmation, il semble désirable que les modules n'aient aucune connaissance des autres modules de l'application.

6.2.3 Configuration de l'application

Pour mettre en place l'application il est nécessaire de spécifier les modules à instancier ainsi que les connexions qui les relient. Cette spécification peut être effectuée par différents intervenants :

- le récepteur : chaque module indique explicitement où il va récupérer les informations qu'il utilise ;
- l'émetteur : de manière opposée le module qui produit une donnée peut spécifier où elle doit être transmise ;
- un autre module : un module particulier peut spécifier les connexions entre les autres modules de l'application.

De plus, les connexions entre les modules peuvent être soit statiques, c'est-à-dire spécifiées lors du démarrage de l'application puis inchangées, soit dynamiques, c'est-à-dire modifiables à n'importe quel moment.

Etant donné que les modules de l'application n'ont pas connaissance des autres modules, ils ne peuvent établir directement les connexions. Un module particulier appelé *contrôleur* est donc utilisé pour créer l'application en lançant les autres modules. Du fait qu'il a connaissance de tous les modules qu'il a créé, il peut ensuite spécifier les connexions entre-eux.

6.2.4 Synchronisation et cohérence

Le niveau de synchronisation utilisé dans une application distribuée a d'importantes conséquences sur l'extensibilité de l'application ainsi que la cohérence du résultat obtenu. Par exemple, un modèle de type BSP [178] permet d'obtenir un résultat déterministe mais au prix d'opérations de synchronisations globales qui peuvent introduire un surcoût important. En particulier pour les applications interactives, il est important que les performances d'une partie de l'application n'affectent pas outre mesure les autres parties. Cela nécessite donc de découpler les différents modules en autorisant des fréquences d'activation différentes, tout en respectant les contraintes demandées par l'application. La première conséquence se situe au niveau des flux de données entre modules. Il est nécessaire de modifier ces flux, par exemple en sélectionnant une partie des données (échantillonnage) ou en interpolant les données. Cette opération peut être effectuée à plusieurs niveaux :

- le récepteur : chaque module peut spécifier la méthode de traitement de ses données, voir directement l'implanter s'il a accès à l'historique des messages ;
- un autre composant : de la même façon que les connexions, les méthodes de filtrage peuvent être spécifiées par un composant particulier ;
- déduite de contraintes : l'utilisateur peut spécifier ses exigences sous formes de contraintes de cohérence que le système aura la responsabilité d'implanter.

La fréquence d'activation de chaque module peut être soit implicite via filtrage des données, c'est-à-dire que chaque module est activé quand les filtres dont il dépend envoient des nouvelles données, soit explicite via un algorithme d'ordonnancement particulier, et dans ce cas le filtrage des données devra agir en fonction de cet algorithme.

Dans le but de réutiliser un même module dans une application interactive ou non, il faut que celui-ci soit indépendant des mécanismes de synchronisation avec les autres modules. De ce fait, nous introduisons de nouveaux objets appelés *filtres* et *synchroniseurs*, placés sur les connexions entre modules pour implanter ces mécanismes. L'ordonnancement des modules ainsi que les politiques de couplage associées sont entièrement spécifiés par les mécanismes de filtrage des données d'entrée de chaque module. Ainsi,

chaque module attend au début de chaque itération un nouveau message sur chacun de ses ports d'entrées. Cette attente est implicitement contrôlée par les filtres et synchroniseurs présents.

6.2.5 Placement et allocation des ressources

Une fois la structure de l'application en place, il est nécessaire de l'instancier en fonction des ressources disponibles. Ce placement peut être spécifié directement à la construction de l'application, ou bien déduit à partir du placement de certains modules (périphériques, données). Il peut être optimisé en se basant sur un modèle de la plateforme paramétré, par exemple, par des mesures de performance. Enfin, ce placement peut être statique ou dynamique, ce qui implique dans ce dernier cas de disposer d'opérations de migration des modules de l'application.

Bien que ces fonctionnalités soient intéressantes, elles ne sont pas primordiales en environnement homogène de type grappe. Dans un premier temps la spécification de l'application est donc entièrement statique. Il est impossible de modifier les connexions ou les modules après le démarrage de l'application. Une extension permettant plus de souplesse pourra être envisagée par la suite, par exemple en exploitant des environnements autorisant la migration transparente de processus de type Kerrighed [120].

6.3 Exemple d'application

Avant la présentation formelle du modèle FlowVR, nous allons l'illustrer en étudiant le cas de l'application de simulation de fluide parallèle interactive présentée à la section 5.2.1 (page 39).

La structure de cette application (figure 6.1(b)) comporte trois modules :

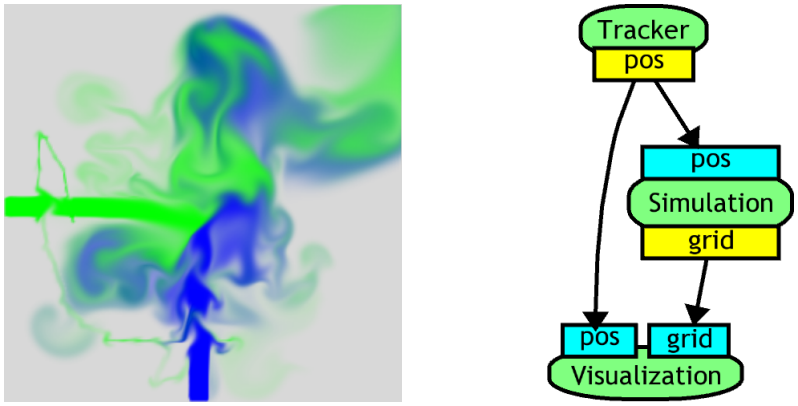
Tracker : récupère la position de la souris.

Simulation : mets à jour la densité du fluide en fonction des mouvements de la souris.

Visualization : affiche le curseur de la souris et le résultat de la simulation.

Pour représenter graphiquement les applications, nous utiliserons dans la suite les conventions suivantes :

- les modules sont représentés par des ovales de couleur verte ;
- les données d'entrée sont représentées par des rectangles bleus au dessus du module associé ;
- les données de sortie sont représentées par des rectangles jaunes au dessous du module ;
- les connexions sont représentées par des flèches noires.



(a) (b)

Figure 6.1 *Structure de l'application exemple.*

6.4 Des modules réutilisables

6.4.1 Définition d'un module

Un module est un programme (dans le sens processus ou thread) s'exécutant sur une machine donnée. Chaque module de l'application est différencié par un *identifiant unique*. Pour échanger des informations avec les autres modules, il déclare une liste de *ports d'entrée* et de *ports de sortie*. Chaque port est défini par un nom et optionnellement un type de donnée.

Une tâche parallèle donne lieu à plusieurs processus ou threads de calculs. Chacun d'entre-eux se traduit par un module différent dans l'application. Ces modules possèdent leurs propres ports. Dans la plupart des cas ces ports seront identiques sur tous les modules, mais cela peut ne pas être le cas. Par exemple, une donnée d'entrée peut n'être requise que sur le premier module, alors qu'un résultat peut être produit sur chaque module, mais de façon découpée. Dans le cas de l'exemple, la simulation de fluide est parallélisée avec MPI, chaque processus donnant lieu à un module avec un port d'entrée *pos* et un port de sortie *grid*. Le résultat calculé par chaque module correspond au bloc de la grille 2D. Si cette simulation est découpé en 4 processus, cela correspond du point de vue du reste de l'application à 4 modules (figure 6.2).

6.4.2 Données échangées par les modules

Les ports d'entrée et de sortie des modules communiquent des données sous forme de *messages*. Un message est un bloc de données associé à un certain nombre d'informations sémantiques appelées *estampilles*. Ces estampilles sont définies par un nom et un type de donnée (parmi *int, float, string, array*). Tous les messages contiennent au moins les

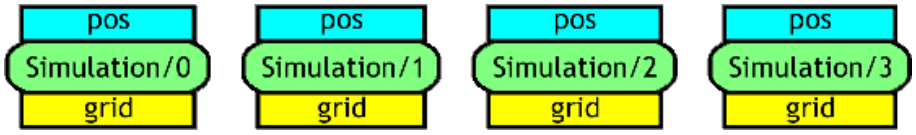


Figure 6.2 *Modules de simulation parallélisée sur 4 machines.*

estampilles suivantes :

source de type *string* : l'identifiant de la source du message (nom du module et du port ayant produit ce message).

it de type *int* : le numéro d'itération du module source lors de l'envoi du message.

num de type *int* : le numéro du message dans le flux de données.

Ces estampilles permettent d'identifier certaines propriétés du messages, sans avoir à l'analyser. Elles sont utilisées pour implanter les opérations de routage et de filtrage sur les messages. Par exemple, **source** permet de déterminer à qui envoyer le message en regardant dans le graphe de l'application les connexions partant de la source. **Num** permet de s'assurer que les messages sont reçus dans le bon ordre. **It** peut être utiliser pour retrouver les dépendances, c'est-à-dire les données utilisées pour le calcul d'un message donné. En effet, les modules consommant un message sur chacune de leur entrée à chaque itération, grâce au numéro d'itération it d'un message on peut retrouver les données utilisées en recherchant dans les flux de messages en entrée du module le message dont le numéro num correspond au it cherché. Ce type de calcul sera utile par la suite pour gérer la cohérence entre flux de données.

En plus de ces estampilles prédéfinies l'utilisateur peut ajouter d'autres informations utiles comme les dimensions des données si celle-ci sont sous forme de vecteurs ou matrices, ou encore la boite englobante d'un modèle 3D. Ces estampilles seront lisibles par tous les objets de l'application FlowVR et permettront d'effectuer des communications complexes adaptées à l'application (frustum culling par exemple). Cependant les données elles-mêmes seront toujours considérées comme des données binaires opaques, permettant ainsi aux modules d'utiliser le format le plus approprié sans affecter le système.

6.4.3 Opérations utilisées par les modules

Comme défini dans la section 6.2.2 (page 47), les modules n'ont qu'une vision locale de l'application. Ainsi les opérations qu'ils ont à leur disposition se limitent au minimum nécessaire pour remplir leur rôle. Cela permet d'avoir un modèle simple du fonctionnement des modules, ce qui est important car ce modèle fait la liaison entre l'étape de création des modules et celle de la construction de l'application les utilisant.

Les cinq premières opérations sont les plus importantes et sont indispensables. Les deux restantes permettent d'optimiser les performances du module.

6.4.3.1 *init()*

Avant de pouvoir travailler, un module doit s'initialiser via l'opération ***init()***. Son rôle est notamment de spécifier l'identifiant du module ainsi que la liste des ses ports d'entrée et ses ports de sortie. Cette opération n'est pas bloquante. En particulier elle n'attend pas que le reste de l'application soit initialisé. Il n'est donc pas possible de lire des informations sur les ports avant le premier appel à ***wait()***.

6.4.3.2 *wait()*

Les modules reposant sur un modèle itératif, l'opération ***wait()*** est utilisée pour démarrer une nouvelle itération. Sa sémantique est la suivante :

---

pour tout les ports d'entrée **faire**
  si ce port est connecté **alors**
    Attendre un nouveau message sur ce port

Cette opération bloque donc le module jusqu'à ce qu'il puisse effectuer l'itération suivante, ce qui est déterminé par les messages en entrée. La disponibilité de ces messages est liée au filtrage effectué en dehors du module. La nature des contraintes résolues par ce filtrage n'est pas visible pour le module. C'est cette séparation qui permet de rendre le module indépendant des méthodes de couplage choisies pour une application donnée. A titre d'exemple, ces contraintes peuvent être une fréquence d'activation ou une barrière entre plusieurs modules (section 6.6.1.1 page 60).

En cas d'erreur ou de fin de l'application, l'opération ***wait()*** renvoie une valeur nulle.

6.4.3.3 *get()*

L'opération ***get()*** s'effectue sur un port d'entrée et permet de lire le message courant reçu sur ce port. Elle ne peut être appelée qu'après le premier ***wait()***. C'est une opération non bloquante qui se contente de récupérer le message reçu lors du ***wait()*** précédant. En particulier, entre deux ***wait()*** le résultat de ***get()*** sur un port donné ne changera pas. Si le module n'appelle pas ***get()*** sur un port d'entrée l'opération ***wait()*** attendra tout de même un message sur ce port.

Il est possible qu'un port d'entrée ne soit pas connecté, dans ce cas ***get()*** renverra un message invalide. Le module doit savoir gérer ce cas. Il peut utiliser une donnée par défaut, ou produire une erreur et terminer son exécution.

6.4.3.4 *put()*

Pour transmettre ses résultats le module utilise l'opération ***put()*** en spécifiant le port de sortie ainsi que le message à envoyer. C'est une opération non bloquante. Lors d'une itération il n'est possible d'envoyer qu'un seul message sur chaque port de sortie. Si ***put()*** n'est pas appelé aucun message n'est envoyé. Cette contrainte d'un seul message par itération permet de gérer les contraintes sur la taille des buffers de communication lors de chaque ***wait()***.

Dans le cas ou le port de sortie n'est pas connecté, le message est silencieusement supprimé. Pour éviter de faire des calculs inutiles dans ce cas, il est possible de le détecter via l'opération ***isConnected()*** décrite dans la section 6.4.3.7.

6.4.3.5 *close()*

Symétriquement à ***init()***, l'opération ***close()*** permet de déclarer la fin de l'exécution du module. Aucune autre opération ne peut être utilisée après l'appel à ***close()***.

6.4.3.6 *alloc()*

Pour éviter les recopies inutiles de données, il est souhaitable d'utiliser un système de mémoire partagée qui permet, en particulier pour le cas ou plusieurs modules se trouvent sur une même machine, de ne transmettre que le pointeur sur ces données. L'inconvénient de cette approche est qu'il n'est pas possible d'utiliser l'allocateur de mémoire standard mais il faut fournir une opération permettant de faire ces allocations dans la mémoire partagée. C'est le rôle de l'opération ***alloc()***, qui alloue un bloc de taille donnée et renvoie le pointeur vers ce bloc.

Comme une même donnée peut être accédée par plusieurs modules simultanément, il est nécessaire d'utiliser une synchronisation de ces accès. Nous utilisons ici une politique très simple : le module qui alloue un bloc via ***alloc()*** peut le modifier librement jusqu'à ce qu'il exécute ***put()***. A partir de ce moment aucune modification n'est permise sur ce bloc. Les données reçues par ***get()*** ne peuvent être que lues. Il est par contre permit d'envoyer via ***put()*** des données en lecture seule, comme dans le cas d'un envoi d'une même donnée sur plusieurs ports ou de la retransmission d'une des données d'entrée.

6.4.3.7 *isConnected()*

Il arrive fréquemment qu'un calcul produise plusieurs résultats. Dans l'exemple de la simulation de fluide il y a une grille de densité du fluide ainsi qu'une grille des vecteurs vitesse. Pour rendre chacun de ces résultats disponibles au reste de l'application, le module déclare plusieurs ports de sortie. En fonction de l'application seuls certains ports seront réellement utilisés. Cela ne pose pas de problème étant donné que l'opération ***put()*** supprime les messages envoyés sur des ports non connectés, mais cela introduit

un certain nombre d'opérations inutiles pour construire ces messages, voir même pour en calculer les données. Pour économiser ces opérations le module peut utiliser l'opération ***isConnected()*** qui détermine si un port donné est connecté dans l'application. Cette information n'est disponible qu'après le premier appel à ***wait()***.

6.4.4 Exemple

Dans notre application d'exemple, les trois modules utilisent les opérations décrites précédemment suivant les algorithmes présentés ci-dessous.

Tracker

init("Tracker"***)***
tant que ***wait()*** **faire**
  lecture de la position de la souris
  put(pos***)***
close()

Simulation

MPI_Init()
init("Simulation/"+rank***)***
tant que ***wait()*** **faire**
  get(pos***)***
  mise à jour de l'état de la grille
  put(grid***)***
close()

Visualization

init("Visualization"***)***
tant que ***wait()*** **faire**
  get(pos***)***
  get(grid***)***
  affichage d'une nouvelle image
close()

6.5 Graphe de flux de données

Une fois que les modules de l'application sont définis, ils sont assemblés en connectant leurs ports d'entrée et de sortie. Ces connexions forment le graphe de flux de données de l'application.

6.5.1 Connections simples

Une connexion de base relie un port d'entrée à un port de sortie et transmet les messages en respectant l'ordre d'émission (mode *FIFO*). Bien qu'un port d'entrée ne peut être relié qu'à un seul port de sortie, à l'inverse un port de sortie peut être connecté à plusieurs port d'entrée, les données sont dans ce cas transmises à tous les ports connectés.

En reprenant notre exemple de simulation de fluide interactive (section 6.3 page 49), pour construire une première version simple non parallélisée il suffit de connecter le port de sortie *pos* du module *Tracker* aux port d'entrée correspondant des deux modules *Simulation* et *Visualization*, et de connecter le résultat de la simulation (port *grid* du module *Simulation*) au port *grid* du module *Visualization* (figure 6.1(b) page 50).

Le mode de communication *FIFO* permet de garantir une cohérence forte pour l'application. En effet les modules sont exécutés à la même fréquence et leurs données d'entrée correspondent à la même itération. Dans notre exemple, la position du traqueur reçue par la visualisation est la même que celle utilisée par la simulation. Toutefois l'inconvénient majeur de ce mode d'exécution est qu'il fixe les performances de toute l'application à la fréquence du module le plus lent.

6.5.2 Filtrage des données

Dans le cas de modules parallélisés des schémas de communications collectives sont nécessaires. Ces schémas reposent sur des constructions plus complexes que de simples assemblages de connexions point-à-point. Pour répondre à ces contraintes nous introduisons de nouveaux objets dans le graphe de flux de données.

La plupart des schémas de couplage de codes parallèles peuvent s'exprimer à l'aide d'opérations de filtrage sur les données transmises. Par exemple pour paralléliser un module il est parfois nécessaire de découper les données, ou bien de répartir chaque messages vers différentes machines (parallélisme inter-itérations). Ensuite les résultats doivent être recombinés en concaténant les données ou en réordonnant les messages.

Appliquer des filtres sur les données transmises peut être utile pour d'autres utilisations telles que convertir le type des données, compresser/décompresser, ou encore ne sélectionner qu'une partie des données, comm le *frustum culling* (section 12.3 page 113) par exemple.

Ces opérations dépendent des modules impliqués dans la communication. Comme la conception d'un module doit être indépendante des autres modules avec lesquels il

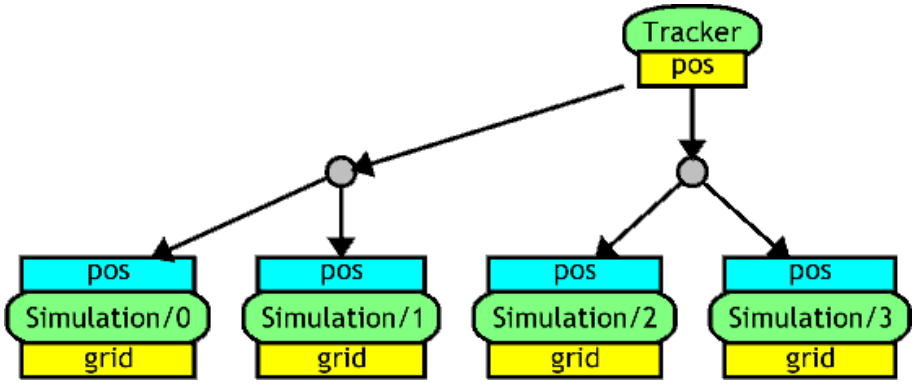


Figure 6.3 *Arbre binaire de broadcast utilisant des noeuds de routages (représentés par des ronds).*

communique ce filtrage doit être implanté par une autre entité. Nous définissons donc les *filtres*, qui sont des objets implantant chacun une opération de filtrage de base. Comme les modules, ils possèdent des ports d'entrée et de sortie. Toutefois ils ne sont pas basés sur un modèle itératif reposant sur son propre fil de calcul mais plutôt sur une structure de type évènements. Un filtre est activé à chaque fois qu'un nouveau message est disponible sur l'un de ses ports d'entrée. Il peut aussi utiliser d'autres évènements tels qu'une horloge périodique. Pour pouvoir implanter des opérations telles que réordonner des messages ou interpoler/combiner plusieurs données un filtre a accès à l'historique des messages reçus sur chacun de ses ports.

L'implantation d'un filtre dépend souvent du type de donnée transmise. Il est donc important que l'utilisateur puisse implanter ses propres filtres. Comme les filtres ne sont pas contenu dans leur propre programme comme le sont les modules, cette implantation est très dépendante de l'implantation du modèle lui même. Ceci est toutefois inévitable pour assurer un surcoût minimal. Chaque filtre implante une opération de base et une série de ces opérations peut parfois être nécessaire lors d'une communication. Leur surcoût doit donc être le plus faible possible.

6.5.3 Communications collectives

Comme expliqué ci-dessus, les filtres permettent d'implanter des schémas de communications entre modules parallèles. Dans cette section quelques uns des ces schémas classiques sont présentés à l'aide de l'exemple d'application de simulation de fluide. Tous ces schémas se retrouvent dans les librairies de passage de messages comme MPI. Nous présentons ici leur transposition au modèle *data-flow* de FlowVR.

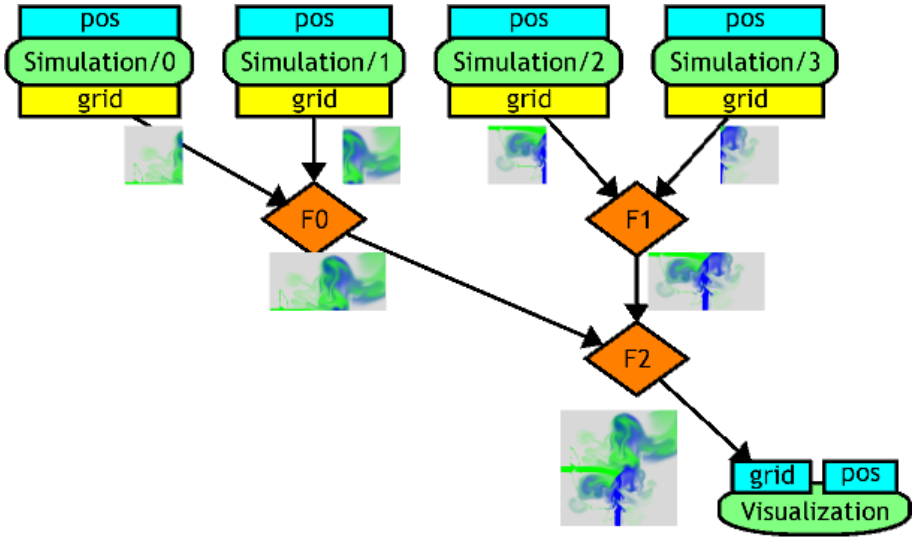


Figure 6.4 *Filtres de fusion du résultat de la simulation.*

6.5.3.1 Arbre de broadcast et découpage (*Scatter*)

Un broadcast *plat*, c'est-à-dire où la source des données les envoie directement et séquentiellement à chaque destinataire, a un coût linéaire en nombre de machines. Pour réduire ce coût il est possible d'utiliser des communications en arbre, ou la source envoie les données à quelques machines, qui les retransmettent à d'autres machines, récursivement jusqu'à ce que tous les destinataires obtiennent une copie des données. Cette méthode réduit le coût du broadcast en $O(\log n)$.

Pour implanter ce schéma dans le graphe de flux de données il faut pouvoir spécifier via quelles machines les communications sont routées. Pour cela on utilise un type de filtre particulier appelé *noeud de routage*. C'est un filtre qui ne modifie en rien les données mais permet de spécifier un chemin de routage dans le graphe. La figure 6.3 montre le schéma utilisant ces noeuds de routage pour faire un arbre de broadcast binaire entre le module *Tracker* et les modules *Simulation*.

Si tous les destinataires n'ont pas besoin de toutes les données mais seulement d'une partie on utilise une opération de découpage ou *Scatter* à la place d'un broadcast. Cela ce traduit par le même schéma de communication à la seule différence que les noeuds de routages sont remplacés par des filtres plus évolués qui découpent les données en 2 morceaux (ou n pour un arbre n-aire).

6.5.3.2 Fusion de données (*Gather*)

Quand on utilise un parallélisme basé sur les données, le calcul est réparti sur plusieurs machines qui produisent chacune une partie du résultat. Si un autre calcul nécessite le résultat complet il est nécessaire de fusionner les données partielles via l'opération appelée

Gather. Cette opération dépend du type de données transmises. Dans le cas d'un vecteur de données souvent il suffit de concaténer les sous-vecteurs. Si les données sont des forces à appliquer à une liste d'objets alors il faut additionner chaque valeur. Dans notre exemple les données sont des densités sur une grille 2D. En utilisant un filtre qui fusionne deux sous-grilles adjacentes en fonction de leur position relative on peut fusionner récursivement toutes les données (figure 6.4). Pour connaître la disposition relative des blocs à fusionner le filtre utilise les estampilles contenues dans chaque message (section 6.4.2 page 50).

6.5.3.3 Combinaison de schémas de base

Les communications collectives en $N \times M$, c'est-à-dire ou N modules produisent des données qui doivent être transmises à M autre modules, sont un problème récurrent en couplage de codes parallèles (section 3.2.2.2 page 24). Par le jeu des filtres et connexions, il est possible d'implanter des schémas très optimisés. Une approche simple consiste à combiner plusieurs schémas comme ceux présentés en les ajoutant dans le graphe de l'application. Un système de scripts (section 7.3 page 71) permet à l'utilisateur de spécifier le schéma adapté.

Si on reprend notre application d'exemple en considérant le cas ou le rendu est distribué sur plusieurs machines, alors une telle communication est nécessaire entre les modules de simulation et ceux de visualisation. Pour l'implanter on peut simplement ajouter un arbre de broadcast après l'opération de fusion. La figure 6.5 présente le résultat pour le cas ou on utilise 8 modules de simulations et 2 modules de rendu.

6.6 Asynchronisme contrôlé : ordonnancement par les données

Les réseaux de modules et filtres tels que présentés dans la section précédente permettent de spécifier des applications complexes impliquant plusieurs composants parallèles. Toutes les communications étant en mode *FIFO*, tous les composants de l'application s'exécutent de manière synchrone, ce qui garantit une très forte cohérence mais limite les performances en terme de fréquence de rafraîchissement, latence et passage à l'échelle de l'application. Dans cette section nous allons introduire un nouveau type d'objets pour spécifier d'autres politiques de synchronisation.

6.6.1 Synchronisation entre modules

L'application étant basée sur un graphe de flux de données, il est nécessaire de modifier ce flux pour modifier la fréquence d'activation des différents composants. Par exemple

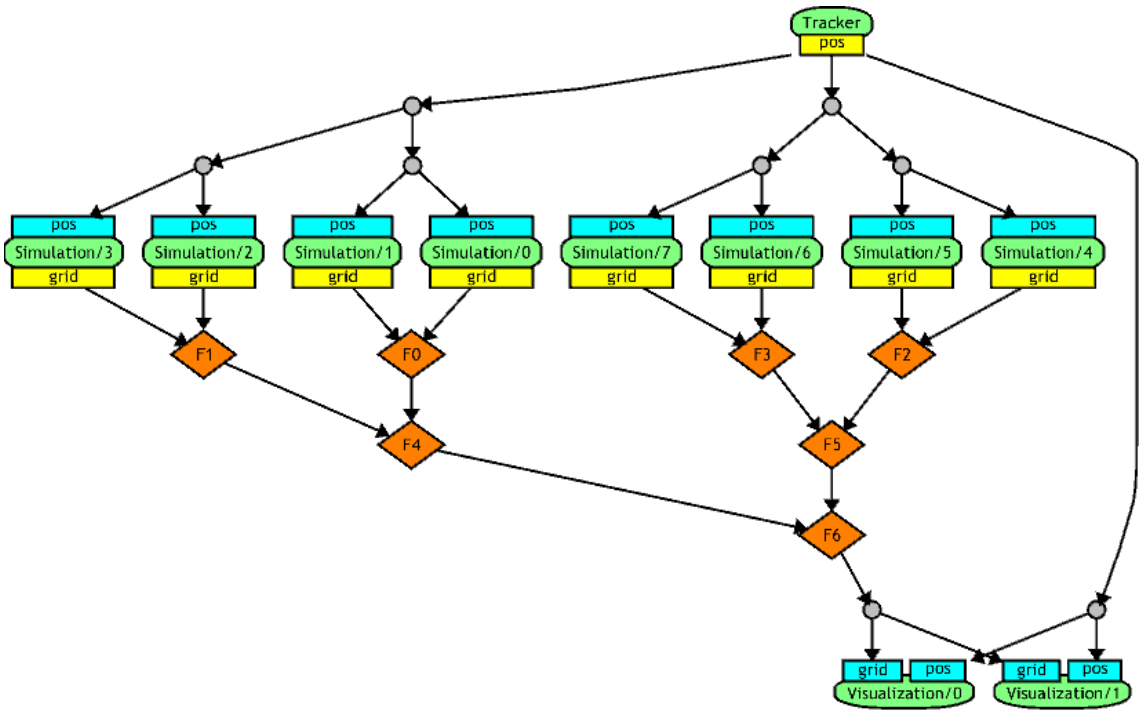


Figure 6.5 *Application complète de simulation de fluide parallèle et interactive.*

il est possible de filtrer les messages en en supprimant un sur deux et ainsi activer une partie de l'application deux fois moins fréquemment que le reste. Un schéma plus courant pour les applications interactives est l'ordonnancement de chaque partie à la fréquence maximale que le permettent les ressources qui lui sont allouées. Dans ce cas, les données communiquées entre les différentes parties ne doivent pas introduire d'attente comme dans le cas *FIFO*. Un mécanisme simple est d'utiliser un mode de communication de ré-échantillonnage *glouton* : les modules utilisent la dernière donnée disponible en jetant les autres données et potentiellement en réutilisant la même donnée qu'à l'itération précédente si aucune nouvelle donnée n'est parvenue. Ce type de filtrage peut introduire des incohérences dans l'application : les données utilisées par un module ne sont plus garanties comme étant calculées à la même itération. Cela veut dire par exemple que le module de rendu peut visualiser des parties de la scène correspondant à des temps de simulations différents. Dans certain cas cette incohérence n'est pas gênante pour l'utilisateur mais dans d'autres circonstances cela peut ne pas être acceptable et il faut alors implanter une politique de couplage différente.

Le choix de la politique de couplage dépend de nombreux facteurs extérieurs. Comme le filtrage des données, cette opération ne doit donc pas être implantée par les modules de l'application. C'est important pour qu'un même module (la simulation de fluide parallèle par exemple) puisse être exploité dans des types d'applications différents (interactions

temps-réel versus simulation classique en batch-processing). Contrairement aux filtres qui ont une influence localisée sur l'application (au niveau des communications reliant deux composants), les politiques de couplage ont des répercussions plus globales. De plus il y a un important compromis entre la centralisation des décisions liées à cette politique, la cohérence de l'application, et l'extensibilité du système. Une politique gérée par un composant centralisé peut garantir une forte cohérence entre toutes les communications mais pénalise le passage à l'échelle de l'application, alors qu'une politique décentralisée sera plus performante mais offrira moins de cohérence. De plus, les prises de décisions de ces politiques peuvent suivre un schéma de distribution différent du filtrage des données en résultant. Pour ces différentes raisons nous avons choisi d'utiliser un nouveau type d'objets appelé *synchroniseur*.

6.6.1.1 Les synchroniseurs

Un *synchroniseur* est un objet similaire aux *filtres* (section 6.5.2 page 55), à la différence qu'il est responsable de la prise des décisions liés à une certaine politique de couplage. Pour prendre ces décisions un synchroniseur reçoit les estampilles (section 6.4.2 page 50) indiquant les messages disponibles ainsi que l'état des différents modules. Le résultat de ses décisions est transmis sous forme d'ordres qui sont ensuite envoyés à des filtres implantant le filtrage des données en conséquence. Cette séparation entre la prise de décision et le filtrage des données est importante pour pouvoir spécifier un schéma de distribution différent pour ces deux opérations (i.e. avoir une politique de couplage centralisée sans pour autant avoir à centraliser les données elles-mêmes). Elle permet aussi de changer l'implantation d'une partie sans affecter l'autre. Par exemple une politique *glouton* implique la destruction de certain messages. Ceci peut ne pas être acceptable dans le cas ou les données du message sont exprimées de manière incrémentale, c'est-à-dire par rapport au message qui les précèdent (évènements, déplacement de souris, ...). Dans ce cas le filtre ne doit pas détruire les messages mais au contraire les fusionner (concaténation des évènements, sommation des déplacements). L'implantation du filtre doit alors être modifiée mais pas celle du synchroniseur. De même il est possible de modifier le synchroniseur pour implanter une politique légèrement différente (comme ne jamais réutiliser le même message et ainsi éviter des calculs inutiles si c'est la seule dépendance de donnée) sans affecter le filtrage des données lui-même.

6.6.1.2 Les ports d'activations

Les seules informations disponibles aux synchroniseurs sont les estampilles des messages envoyés par les modules (numéro d'itération, éventuellement dimensions, bounding box, ...). Pour être informé de l'état des modules, et en particulier être signalé quand le module a fini son itération et donc a besoin d'une nouvelle donnée pour l'itération suivante, les synchroniseurs peuvent utiliser les messages de ports particuliers appelés ports

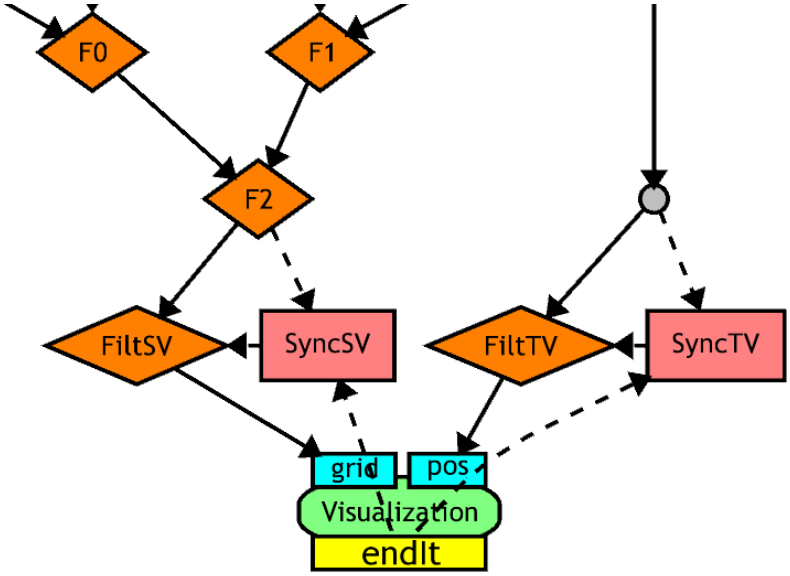


Figure 6.6 *Couples filtres (losanges oranges) + synchroniseurs (rectangles roses) découplant la visualisation des autres modules (couplage asynchrone). Les pointillés représentent les connexions ou transitent des estampilles uniquement.*

d'*activation*. Ces ports sont prédéfinis pour chaque module. Le port de sortie *endIt* signale la fin d'une itération, c'est-à-dire qu'un message est envoyé à chaque fois que le module termine une itération (i.e. appelle la méthode *wait*). Ce port permet donc de savoir quand une nouvelle donnée doit être envoyée. De manière symétrique le port d'entrée *beginIt* permet de faire attendre le module avant de commencer la prochaine itération. Ce port est particulièrement important pour pouvoir bloquer un module (par exemple pour imposer une fréquence maximum ou pour empêcher la saturation des buffers en aval) quand celui-ci n'a pas d'autre port d'entrée qui pourrait le permettre.

6.6.2 Exemple d'utilisation des filtres et synchroniseurs

Dans notre application exemple nous pouvons appliquer une politique glouton pour le module de visualisation de manière à rafraîchir l'image le plus souvent possible (et ainsi permettre de changer le point de vue de manière fluide par exemple) et à mettre à jour la position du curseur le plus rapidement possible pour diminuer la latence perçue par l'utilisateur et ainsi augmenter le confort d'utilisation. Dans ce cas nous pouvons utiliser les couples filtres + synchroniseurs présentés sur la figure 6.6.

Dans le cas d'une tâche distribuée comme la simulation il est souvent nécessaire de garder une cohérence forte entre les différents modules. Dans ce cas il est possible d'utiliser un synchroniseur centralisé qui utilise l'état de tous les modules pour décider de la donnée à utiliser (figure 6.7).

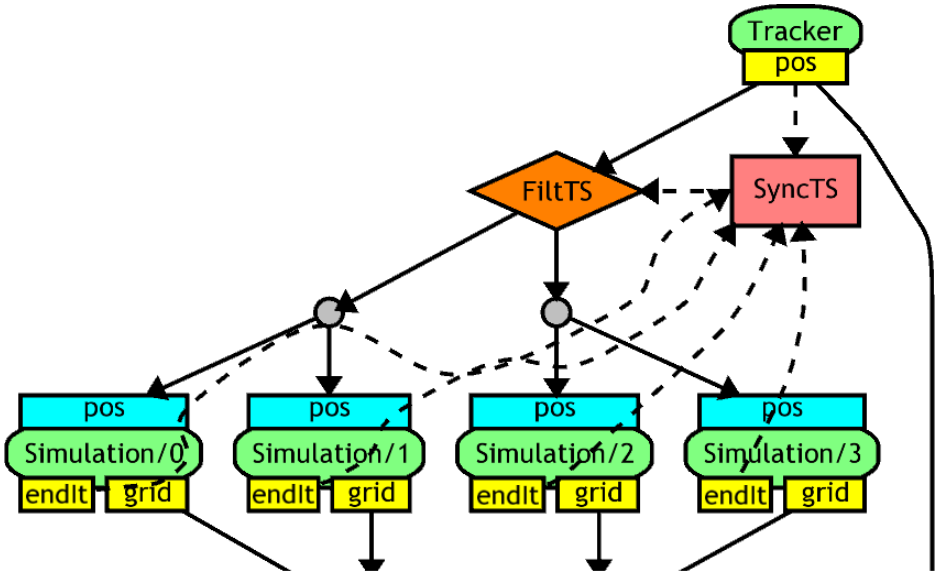


Figure 6.7 *Schéma de couplage asynchrone cohérent : un synchroniseur glouton est utilisé pour filtrer les données d'un ensemble de modules.*

En ajoutant ces schémas de synchronisation chaque partie de notre application peut maintenant être exécutée aussi vite que ses ressources le permettent. Dans le cas ou certaines de ces ressources sont partagées, comme le réseau par exemple, cela peut entraîner une consommation inutile des ressources. Par exemple, le traqueur peu envoyer des nouvelles données très fréquemment (souvent plusieurs milliers de fois par secondes), ce qui ne sert à rien si les autres modules ne dépassent jamais les 100 itérations par seconde. Il est possible de restreindre le module *Tracker* à une fréquence plus faible. Pour cela on ajoute au graphe de l'application un synchroniseur relié aux ports d'activation du module *Tracker* permettant de le mettre en attente s'il dépasse la fréquence fixée. Ce synchroniseur transmet les messages de fin d'itération vers le port de début de l'itération suivant en les mettant en attente s'ils sont trop rapprochés, implantant ainsi la contrainte de fréquence souhaitée. La figure 6.8 présente le schéma correspondant dans le graphe de l'application.

6.6.3 Exemples d'algorithmes de filtres et synchroniseurs

Afin de présenter de manière plus précise le fonctionnement des synchroniseurs et leur couplage avec les filtres, cette section contient les algorithmes utilisés pour le synchroniseur glouton ainsi que le filtre associé de sélection des messages. Les ordres transmis entre le synchroniseur et le filtre sont les numéros d'itération des messages à sélectionner. L'algorithme du synchroniseur de limitation de fréquence est aussi présenté.

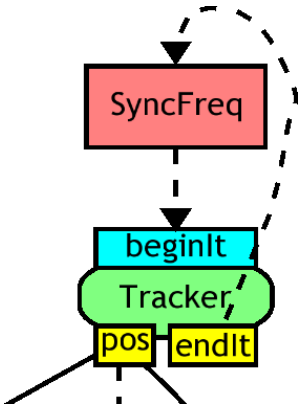


Figure 6.8 *Synchroniseur permettant de contraindre la fréquence du module Tracker.*

Synchroniseur de fréquence maximum

Ce synchroniseur comporte un paramètre *Freq* exprimé en Hertz contrôlant la fréquence d'activation maximale du module connecté. Il est utilisé dans le schéma de la figure 6.8.

Entrée : *endIt* : signaux de fin d'itération du module
Sortie : *beginIt* : signaux de début d'itération du module
Quand début **:**
    put*(beginIt)*
    démarrer le timer pour $1/Freq$ secondes
Quand nouveau message sur *endIt* **:**
    si le timer est expiré **alors**
        supprimer un message sur *endIt*
        put*(beginIt)*
        démarrer le timer pour $1/Freq$ secondes
Quand le timer expire **:**
    si il existe un message sur *endIt* **alors**
        supprimer un message sur *endIt*
        put*(beginIt)*
        démarrer le timer pour $1/Freq$ secondes

Synchroniseur glouton

Le synchroniseur glouton est extrêmement simple. Il se contente d'ordonner l'utilisation du message le plus récent à chaque fois qu'une nouvelle donnée est requise.

Entrée : *in* : estampilles des messages disponibles
Entrée : *endIt* : signaux de fin d'itération du module de destination
Sortie : *order* : numéros des messages à sélectionner
Quand nouveau message sur *endIt* **:**
    n = numéro du message le plus récent sur *in*
    put*(order,n)*
    supprimer tous les messages plus anciens que *n* dans *in*

Filtre de sélection

Pour appliquer les décisions d'un synchroniseur, le filtre de sélection gère la liste des messages reçus et applique les ordres du synchroniseur. Son fonctionnement est plus complexe que le synchroniseur glouton, du fait que l'application d'un ordre reçu peut être mise en attente si le message de donnée correspond n'a pas encore été reçu.

Entrée : *in* : messages disponibles
Entrée : *order* : numéros des messages à sélectionner
Sortie : *out* : messages sélectionnés
Quand nouveau message sur *order* **:**
    n = numéro contenu dans le premier message sur *order*
    supprimer tous les messages plus anciens que *n* dans *in*
    si le premier message sur *in* correspond au numéro *n* **alors**
        put*(out*,premier message sur *in)*
        supprimer le premier message sur *order*
Quand nouveau message sur *in* **:**
    n = numéro contenu dans le premier message sur *order*
    si le nouveau message sur *in* correspond au numéro *n* **alors**
        put*(out*,nouveau message sur *in)*
        supprimer le premier message sur *order*

Implantation 7

Pour faciliter une implantation étape par étape, FlowVR est structuré en un ensemble de couches empilées, permettant de fournir un niveau d'abstraction et de fonctionnalité variable en fonction des besoins. Un ensemble d'outils relativement simples et indépendants implante chaque niveau, ce qui permet d'expérimenter facilement certaines modifications sans affecter le reste du mécanisme. Ce chapitre présente ces outils en partant du niveau le plus bas, constituant les bases du système, jusqu'aux couches supérieures, beaucoup plus souples.

7.1 Environnement d'exécution

Pour favoriser la modularité du système nous avons choisi un environnement d'exécution où chaque module est exécuté dans son propre processus. Cela permet de compiler et lancer chaque module indépendamment et ainsi d'éviter les problèmes de conflits entre les dépendances de chaque module (librairies, outils de compilation, etc). Cela simplifie aussi l'intégration de codes existants, dont les scripts de compilation et de lancement ne doivent être modifiés que très légèrement.

7.1.1 Communications inter-processus

La séparation des modules implique l'utilisation de primitives de synchronisation et communication inter-processus, ou *IPC* (*Inter-Processus Communication*). Plusieurs mécanismes existent sous UNIX tels que les *pipes*, les *semaphores*, les *message queues*, les *sockets* ainsi que les *shared memory areas*. Dans le cadre de FlowVR il est nécessaire de

transmettre des flux de données importants avec le minimum de surcoût possible. Pour cela le plus adapté semble les segments de mémoires partagées (*shared memory areas*). Ceux-ci permettent d'envoyer des données à potentiellement plusieurs modules sans avoir à faire de recopie. Comme ces données sont partagées il est important de gérer explicitement la synchronisation entre les processus pour éviter les corruptions et signaler l'arrivée de nouvelles données. L'API *POSIX Thread*, ou *pthread*, permet de créer des *locks* et des signaux pour cela. Sous Linux depuis l'arrivée du noyau 2.6 une nouvelle implantation [52] de cette API permet d'utiliser ces primitives entre processus différents et avec de très bonnes performances (les *locks* par exemple n'ont recours à un appel système qu'en cas de contention). FlowVR repose donc sur un segment de mémoire partagée pour les communications inter-processus ainsi que les primitives *pthread* pour la synchronisation. Quand ces primitives ne sont pas disponibles une autre implantation peut utiliser des opérations atomiques et des boucles d'attentes actives pour pouvoir tout de même développer des applications FlowVR sur les plateformes ne fournissant pas les *locks* inter-processus (comme le noyau Linux 2.4), mais avec des performances plus faibles.

7.1.2 Démon

Une application FlowVR ne consiste pas uniquement en des communications de données entre processus d'une même machine, mais nécessite aussi des communications avec des modules distants, ainsi que des opérations de filtrage des données (section 6.5 page 55). Un processus particulier appelé *démon* est utilisé pour implanter ces opérations, comme indiqué sur la figure 7.1. Un démon est exécuté sur chaque machine. Il communique via la mémoire partagée avec les modules locaux et envoie via le réseau les messages destinés aux machines distantes.

Le démon implante aussi les filtres et les synchroniseurs utilisés dans le graphe de l'application. Comme ceux-ci dépendent de l'application et peuvent être modifiés en fonction des besoins, ils sont implantés via un mécanisme de plugins. Quand une application nécessite un certain filtre, le démon charge dynamiquement la librairie contenant son implantation. Cette implantation dérive d'une classe abstraite définissant les méthodes utilisées par le démon pour exécuter le filtre (initialisation, réception et envoi de messages). La plupart des autres éléments du démon (connexions réseaux, contrôle des modules) sont aussi implantés sous forme de plugins. Ils permettent ainsi de fournir facilement des implantations alternatives. Par exemple le réseau utilise actuellement le protocole TCP mais une autre implantation autour de MPI a aussi été effectuée.

Le coeur du démon repose sur deux structures de données principales :

– La *table des objets*, répertoriant tous les objets instanciés (tels que les filtres) en fonction de leur identifiant.
– La *table de routage*, indiquant quelles sont les *actions* à effectuer sur un message (i.e. les filtres à activer) en fonction de la provenance du message (l'estampille *source* du message).

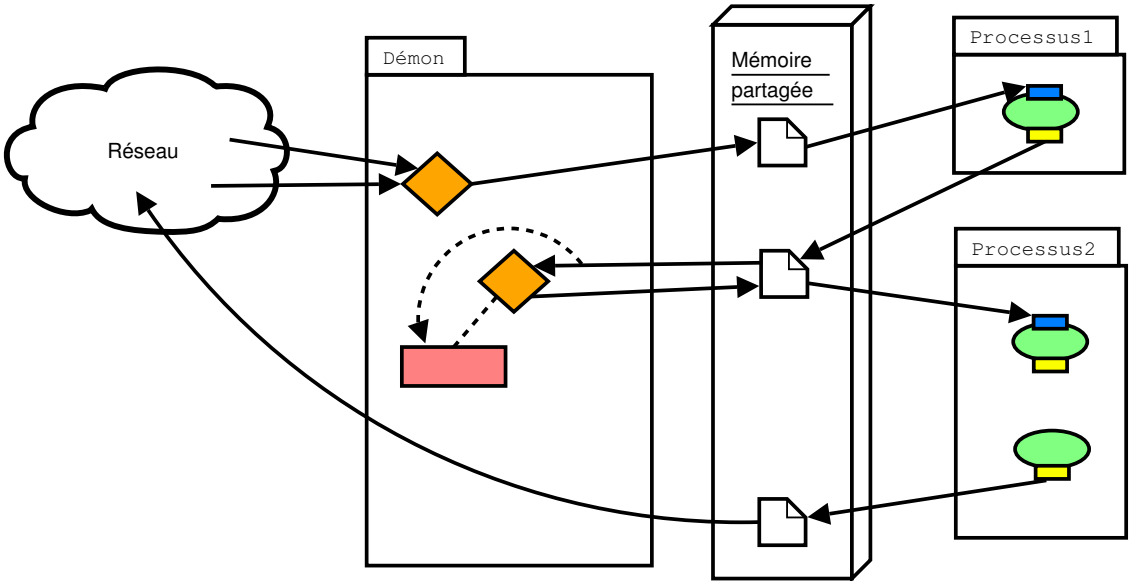


Figure 7.1 *Implantation d'une application FlowVR via un démon par machine.*

En interne le démon est architecturé autour d'un mécanisme de routage de messages via plusieurs threads liés aux modules locaux, connexions réseaux, et évènements périodiques. Chaque thread gère une queue de messages en cours de traitement. Ces messages sont traités un par un en utilisant les actions indiquées dans la *table de routage*. Ces actions peuvent :

– générer de nouveaux messages qui seront alors placés dans la queue de messages à traiter ;
– envoyer un signal à d'autres threads de traitement (modules, connexions réseaux).

En revanche une action ne doit pas effectuer un traitement long ou attendre longtemps un signal car cela ralentirait la boucle de traitement des messages du thread appelant. Pour les cas où c'est nécessaire l'action doit faire appel à un autre thread. Par exemple, quand un message doit être envoyé sur le réseau cet envoi n'est pas fait directement par l'action sur ce message mais il est plutôt transmis au thread spécifique d'envoi des données sur la connexion réseau correspondante.

La figure 7.2 montre le schéma de fonctionnement des threads du démon. La fonction principale *main* de l'exécutable crée un premier thread *Commander* qui gère les commandes de contrôle de l'application (décrite dans la section 7.1.3 page 69). Si le mode réseau est activé, un deuxième thread *NetTCP* est créé pour attendre les connexions TCP. Dès qu'un autre démon se connecte, un thread *RecvThread* est créé pour recevoir les messages transmis. Quand le *Commander* reçoit une commande indiquant un nouveau module, il crée alors un thread *Regulator* qui va gérer les messages produits par ce module. La plupart des filtres et synchroniseurs sont de simples objets qui sont ajoutés dans la table des objets, mais certains peuvent aussi créer leur propre thread, qui viennent dans

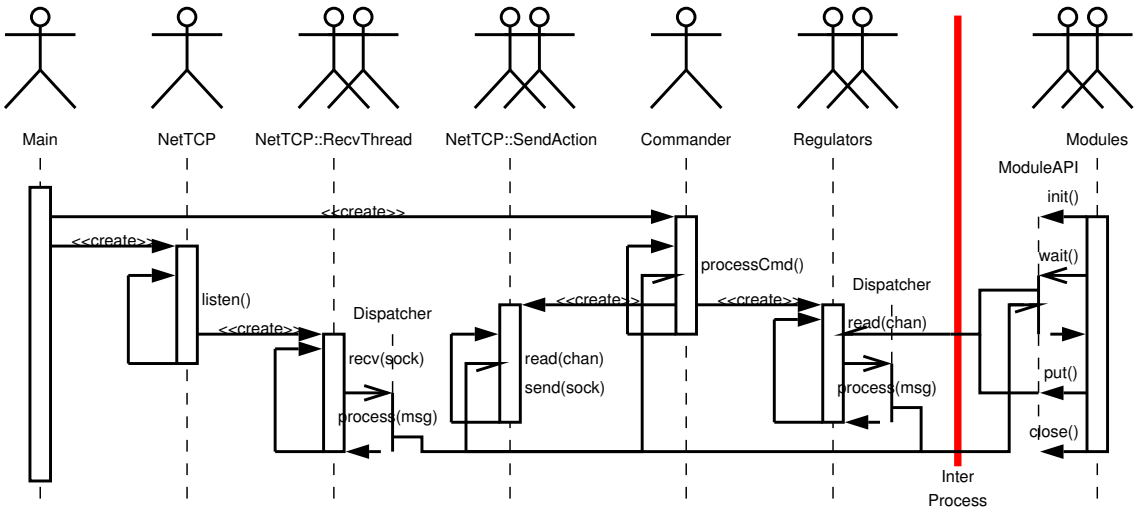


Figure 7.2 *Threads utilisés à l'intérieur du démon et des modules ainsi que les communications entre ces threads.*

ce cas s'ajouter à ce schéma.

Tous les threads qui peuvent produire des messages, comme par exemple les threads *Regulator* et *RecvThread*, utilisent une instance de la classe *Dispatcher* pour gérer la boucle de traitement des messages via la méthode *process(msg)*. C'est une classe abstraite dont l'implantation est faite par un plugin. L'implantation actuelle exécute toutes les actions dans le thread ayant produit le message, mais d'autres politiques pourront être étudiées par la suite. Ces actions peuvent au besoin réveiller le bon thread (module, envoi réseau, commander) en utilisant le signal associé.

Du fait de la séparation entre le processus du démon et les modules, à chaque itération un changement de contexte vers le thread *Regulator* est nécessaire pour implanter les filtres et les synchroniseurs associés à ce module. Dans le cas ou toutes les données sont disponibles localement, comme dans le cas de la figure 6.6 où un couple de filtre et synchroniseur greedy est utilisé sur chaque port d'entrée, les actions correspondantes sont exécutées dans la boucle de traitement des messages du thread *Regulator* et donc le module sera réveillé immédiatement, ce qui fait au total deux changements de contextes. Si en revanche un message doit être reçu par le réseau, l'action de réveil du module sera exécutée directement depuis le thread de réception des messages de la connexion réseau correspondante. Les messages envoyés par le module via la méthode *put* passent eux aussi par le thread *Regulator*, et vont ensuite réveiller un thread d'envoi réseau ou un autre module local.

7.1.3 Configuration de l'application : langage de commandes

Pour spécifier les objets à mettre en place ainsi que les routes (i.e. association source–action) de la table de routage il est nécessaire d'envoyer au démon des commandes les décrivant. Nous utilisons pour cela un protocole de commandes à base de XML. Les commandes suivantes sont disponibles :

addobject : ajoute un nouvel objet en spécifiant son identifiant, la classe à instancier, ainsi que d'éventuels paramètres.

delobject : supprime un objet en spécifiant son identifiant.

addroute : ajoute une route dans la table de routage en spécifiant son identifiant, la source du message à traiter, ainsi que l'action à effectuer. Cette action contient l'identifiant de l'objet destinataire ainsi qu'un paramètre (nom du port dans le cas d'un filtre, nom de la machine distante dans le cas d'une transmission réseau).

delroute : supprime une route en spécifiant son identifiant.

action : invoque une action spécifique d'un objet en spécifiant son identifiant ainsi que ses paramètres. Ceci permet de déclencher des actions telles que le démarrage et l'arrêt de l'application.

group : permet d'invoquer une commande sur un groupe d'objets. Utile pour mettre en pause et redémarrer tous les objets de l'application par exemple.

Toutes ces commandes utilisent un système d'identifiants pour spécifier les objets ou routes auxquelles elles s'appliquent. FlowVR utilise un système similaire aux systèmes de fichiers classiques. Les objets existants dans le démon indépendamment de l'application ont un identifiant *absolu*, c'est-à-dire commençant par `"/"`, tel que `"/NET"` pour le réseau. Les objets appartenant à une application sont exprimés *relativement* à l'identifiant de l'application, apparenté à un répertoire courant. Par exemple, une commande `addobject "Module1"` dans le cadre de l'application `"/A1"` créera un objet portant l'identifiant `"/A1/Module1"`. Cela permet d'exécuter plusieurs applications simultanément de manière indépendante, tout en autorisant le partage de certains objets en utilisant leurs identifiants absolus.

Un objet particulier à l'intérieur du démon appelé *Commander* est responsable de l'implantation de ces commandes. La façon dont ces commandes sont générées et envoyées aux démons est décrite dans les sections suivantes.

7.2 Mécanisme de contrôle d'applications

Démarrer et contrôler une application distribuée constituée de nombreux composants n'est pas trivial, surtout considérant que les modules FlowVR doivent pouvoir utiliser n'importe quelle librairie de communication, et donc souvent le mécanisme de lancement associé. Pour cela FlowVR repose sur un type de module particulier appelé *contrôleur*.

Un contrôleur est un module particulier qui est responsable du lancement et du contrôle du reste de l'application (i.e. démarrage, arrêt, éventuellement pause, monitoring). Pour cela il déclare un port de sortie lui permettant d'envoyer des commandes XML telles que décrites dans la section 7.1.3. Ces commandes sont transmises une par une et sont associées à une estampille *dest* indiquant à quelle machine elles sont destinées. Le démon local envoie alors chaque commande au *Commander* du démon correspondant et récupère la réponse qui est retransmise au contrôleur via un port d'entrée.

7.2.1 Lancement des modules

Les commandes XML envoyées aux démons permettent de contrôler le réseau FlowVR mais elles ne lancent pas les modules eux-mêmes étant donné que ce sont des programmes indépendants qui peuvent utiliser leur propre mécanisme de lancement. De plus, le lancement à distance de ces programmes pose d'importants problèmes de sécurité. Ainsi le contrôleur lance les modules en utilisant les outils appropriés, comme `mpirun` ou `ssh` par exemple. Pour faire la liaison avec l'application FlowVR les variables d'environnement suivantes sont utilisées :

FLOWVR_DAEMON : l'identifiant du démon à utiliser, c'est-à-dire le numéro du segment de mémoire partagée à ouvrir ;

FLOWVR_PARENT : l'identifiant du module contrôleur de l'application, qui sert de préfixe pour les identifiants du module ;

FLOWVR_MODNAME : l'identifiant du module ou groupe de modules à lancer.

Ces variables sont initialisées par le contrôleur avant d'exécuter chaque script lançant un groupe de module de l'application.

Dans l'implantation actuelle de FlowVR un programme assez simple appelé *flowvr-telnet* implante ce rôle de contrôleur de manière similaire au programme *telnet*. Il lit les commandes XML écrites sur son entrée standard et les fait suivre au démon. En plus des commandes décrites à la section précédente des informations supplémentaires sont nécessaires. Pour spécifier quelle machine est concernée par chaque commande un tag XML `dest` est utilisé. Pour spécifier les scripts de lancement des modules de l'application un tag `run` est utilisé. Un tag `wait` est employé pour demander d'attendre le résultat des commandes précédentes. Cela permet par exemple d'attendre que tous les objets soient créés avant de créer les connexions entre eux. Enfin les tags `pause` et `start` permettent de stopper et redémarrer l'application.

7.2.2 Construction du graphe de flux de données

Une application FlowVR peut nécessiter plusieurs milliers de commandes à transmettre. Spécifier ces commandes une par une directement serait difficile voir impossible pour les applications de grande taille. C'est pour cela qu'un ensemble d'outils permet

de le faire en utilisant des descriptions de plus haut niveau de l'application. Le premier d'entre eux est *flowvr-network*, un outil permettant de générer les commandes de lancement des éléments d'une application à partir d'une description XML de ces éléments. Cette description est appelée *graphe instancié* et décrit de manière complète l'application FlowVR telle qu'elle est instanciée sur des machines données. Elle contient la liste des objets (modules et filtres) de l'application, spécifiant leur identifiant, placement, ainsi que leurs ports. Elle contient aussi la liste des connexions entre ces objets, spécifiant les identifiants et ports de la source et la destination, ainsi que le type de connexion (envoi des messages complets ou juste de leurs estampilles).

Bien que le graphe instancié reste une description assez bas niveau elle contient plus d'informations que le langage de commandes. Par exemple, la description des ports des objets ainsi que des connexions permet de vérifier leur validité (i.e. les ports connectés existent bien, le type de connexion est compatible, ...). Changer certains paramètres tel que le placement des modules n'affectent que les données des modules concernés, alors qu'au niveau des commandes ces changements affectent aussi les connexions associées. Cependant d'autres changements tels que la modification du nombre de modules instanciés requièrent des changements plus profonds dans le graphe de l'application.

7.3 Passage à l'échelle : dépliage de graphes

Comme indiqué à la section précédente, le graphe de l'application dépend des paramètres des modules utilisés. Par exemple si on change le nombre de modules utilisés par la simulation de fluide (figure 6.5 page 59), il faut refaire le graphe en modifiant les filtres *Merge2D* qui permettent de rassembler les résultats de chaque processus. Cependant la structure du graphe reste la même : les données d'entrée utilisent un arbre de broadcast pour être envoyées aux n modules de la simulation, et les résultats sont fusionnés 2 à 2 par un arbre inversé de filtres *Merge2D*, quel que soit le nombre exact n de modules utilisés.

En utilisant une représentation de la structure du graphe ainsi que des outils permettant de paramétrer les modules et d'appliquer cette structure pour générer le graphe instancié alors l'utilisateur n'aura qu'à changer les paramètres pour adapter l'application à l'environnement qu'il souhaite et la lancer automatiquement. Nous avons envisagé deux types de représentations de la structure du graphe :

- une représentation *déclarative*, à base de règles élémentaires de transformation des données (i.e. comment transformer deux sous-grilles 2D en une seule grille 2D via le filtre *Merge2D*, ou comment utiliser un filtre pour convertir des données du format *float* en *double*) et d'un moteur de résolution qui cherche à connecter les ports des modules en combinant ces règles ;
- une représentation *procédurale*, à base de procédures spécifiant les structures classiques (broadcast, gather, ...) à partir des éléments de base (connexion FIFO, filtres, noeuds de routage) et permettant à l'utilisateur de définir ses propres procédures.

La première solution peut être très puissante mais difficile à maîtriser, surtout dans le cadre de la conception des premières applications FlowVR. Nous avons donc choisi la deuxième solution, plus simple à implanter. Après avoir acquis de l'expérience avec cette solution il sera toujours possible de tenter des alternatives via la première solution.

FlowVR fournit donc un module Perl permettant de modifier le graphe XML d'une application en utilisant des procédures prédéfinies ou implantées par l'utilisateur. Par exemple notre application de simulation de fluide en mode *FIFO* telle que présentée sur la figure 6.5 peut être générée par le script suivant :

```
addRecursiveBroadcast('Tracker','pos','Simulation','pos');
addRecursiveBroadcast('Tracker','pos','Visualization','pos');
addRoutingNode('R','pc1');
addRecursiveGather('Simulation','grid','R','',Merge2D);
addRecursiveBroadcast('R','','Visualization','grid');
```

La plupart des procédures utilisées prennent en paramètre le préfixe et le port des modules sources et destinations. Ils sont reliés en utilisant le schéma de communication demandé en fonction du nombre d'instances de ces modules. Le script déplie en quelque sorte le graphe de l'application, c'est-à-dire qu'il créé le graphe complet à partir d'une spécification repliée de la structure de celui-ci.

Pour générer la description des modules instanciés utilisée en entrée des scripts de dépliage ainsi que les commandes de lancement nous utilisons une description XML des modules disponibles. Chaque description de module contient la commande de lancement ainsi que la description des instances de modules créées (identifiant, placement et liste des ports), en fonction de la liste des machines que ce module doit utiliser ainsi qu'un certain nombre d'autres paramètres dépendant du module (par exemple nom du périphérique du tracker, taille de la grille à simuler, configuration des vidéo-projecteurs). Ensuite une application est spécifiée en listant les modules à utiliser ainsi que leurs paramètres. Un outil appelé *flowvr-module* prend ces informations en entrée et génère les commandes de lancements de tous les modules de l'application ainsi que la description des modules instanciés. Cette description est donnée en entrée des scripts de dépliage pour générer le graphe instancié de l'application, qui est ensuite transmis en entrée de *flowvr-network* pour générer les commandes XML utilisées par *flowvr-telnet* pour configurer l'application finale. Cette chaîne de traitement est résumée sur la figure 7.3.

En utilisant les structures conditionnelles et de bouclage du langage procédural il est possible de faire des scripts génériques qui sont réutilisables dans de nombreuses applications. Par exemple, dans la suite nous présenterons l'utilisation de FlowVR dans des applications impliquant des chaînes de traitement sur des flux vidéos pour de la reconstruction 3D temps réel et des interactions dans un monde virtuel. Ces applications demandent des centaines de modules et des milliers de connexions et filtres. Ils sont décrit via une dizaine

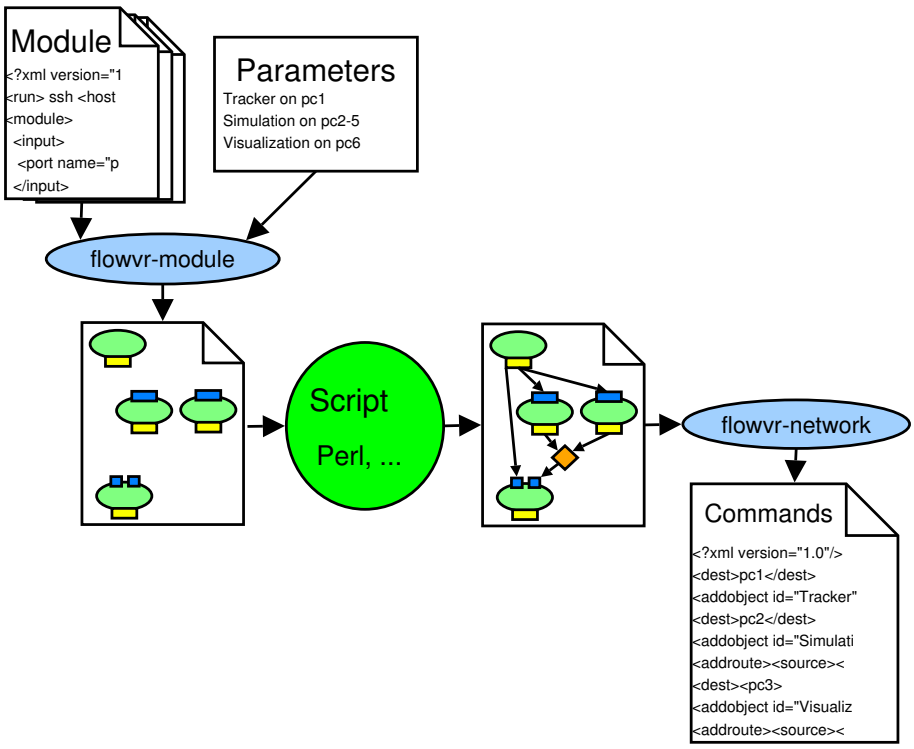


Figure 7.3 *Chaîne d'outils permettant de générer les commandes de lancement d'une application FlowVR en changeant les paramètres des modules instanciés.*

de scripts Perl d'environ 100 lignes chacun et utilisant des schémas différents en fonction des conditions. Par exemple on peut soit utiliser des flux vidéos *"live"* soit les lire à partir de fichiers pré-enregistrés, les schémas de synchronisation étant alors différents. La figure 7.4 présente le graphe instancié d'une telle application. Les objets et les connexions ajoutés par chacun des scripts sont représentés par des couleurs différentes. Un graphe d'une telle complexité ne peut pas être généré manuellement.

7.4 Performances

FlowVR est avant tout un environnement destiné à développer de grandes applications interactives. En ce sens, les résultats les plus importants concernent l'efficacité du modèle et de l'environnement de développement pour la construction et l'exécution de telles applications. Cependant la performance obtenue lors de l'exécution des applications est aussi un critère important, nous allons donc présenter quelques tests effectués sur des applications simples.

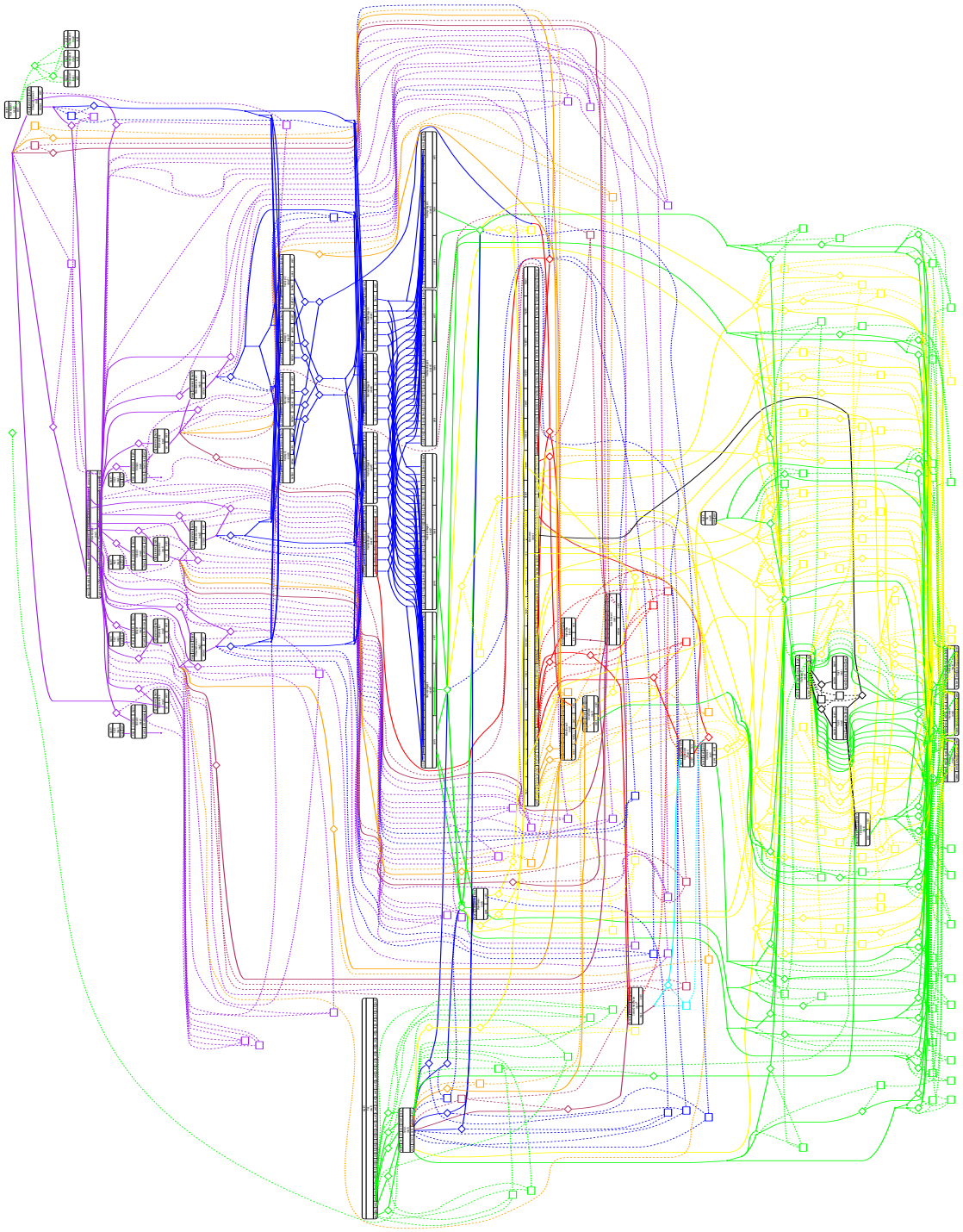


Figure 7.4 *Application FlowVR utilisant de nombreux modules et plusieurs scripts de dépliage, qui totalisent environ 1000 lignes de Perl.*

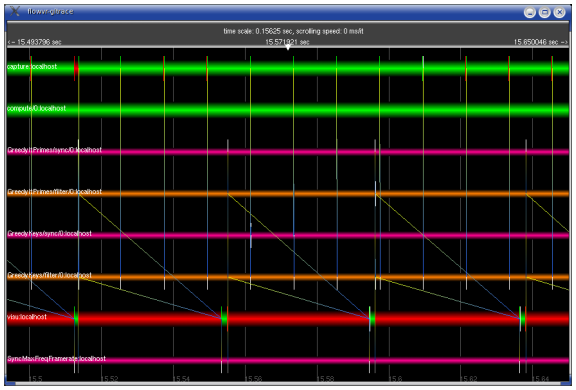


Figure 7.5 *Visualisation de la trace d'une application FlowVR distribuée.*

Les résultats présentés sont toutefois très partiels, du fait que cette première implantation de FlowVR était surtout destinée à tester les concepts du modèle et non à obtenir les performances optimales.

7.4.1 Mesure et analyse de trace

Mesurer les performances d'une application distribuée n'est pas toujours facile. La vitesse de chaque module peut facilement être calculée localement sur chaque machine. En revanche, obtenir des données de latence est plus compliqué si l'opération concernée est répartie sur plusieurs machines. Une latence peut être définie comme la différence de temps entre le moment où une donnée est produite (mouvements de l'utilisateur) et le moment où le résultat correspondant est disponible (l'image de l'environnement virtuel est affichée). Entre ces deux temps le calcul peut passer par un certain nombre de modules distribués sur plusieurs machines (la simulation de fluide par exemple). Mesurer cette latence nécessite donc de tracer les évènements le long du chemin de traitement et comparer les horloges des machines impliquées.

Pour effectuer ces mesures un système de traces a été ajouté à FlowVR, dans le cadre du stage de Julien Garand. Une machine sert d'horloge de référence. Avant et après l'exécution de l'application une série d'allers-retours réseau est effectuée entre cette machine et toutes les autres. Ces *ping* permettent de faire la correspondance entre les cycles CPU de chaque machine et l'horloge de référence. Lors de l'exécution un ensemble d'évènements est stocké, estampillé par l'horloge CPU de la machine locale. Ensuite, en utilisant la description du réseau de l'application et les correspondances des horloges il est possible d'analyser ces données, pour par exemple les visualiser comme sur la figure 7.5.

Plateforme	Fréquence maximale (Hz)	Temps de changement de contexte (microseconde)
Intel Xeon 2.66 GHz	69400	9.11
AMD Opteron 2.0 GHz	130500	4.52
Intel Pentium M 1.7 GHz	88800	5.63
Intel Itanium II 900 MHz	39680	13.93

TAB. 7.1 *Surcoût engendré par les changements de contexte.*

7.4.2 Signaux inter-processus

Les modules étant instanciés dans des processus distincts, toutes les communications entraînent des changements de contexte entre ces processus. Pour quantifier le surcoût associé, nous avons mesuré le nombre d'itérations maximal effectué par seconde pour un module n'effectuant aucun calcul. Ceci permet de déduire le temps nécessaire pour chaque changement de contexte entre les modules et le démon (table 7.1).

Le temps de changement de contexte est très variable en fonction de la plateforme utilisée. Toutefois il ne dépasse jamais les 15 microsecondes, ce qui est négligeable pour les modules fonctionnant aux fréquences classiques d'interaction (de l'ordre de 100 Hz). Ces performances permettent d'atteindre plus de 10000 Hz, ce qui est suffisant pour les modules nécessitant les fréquences les plus élevées, comme pour le contrôle d'un retour haptique.

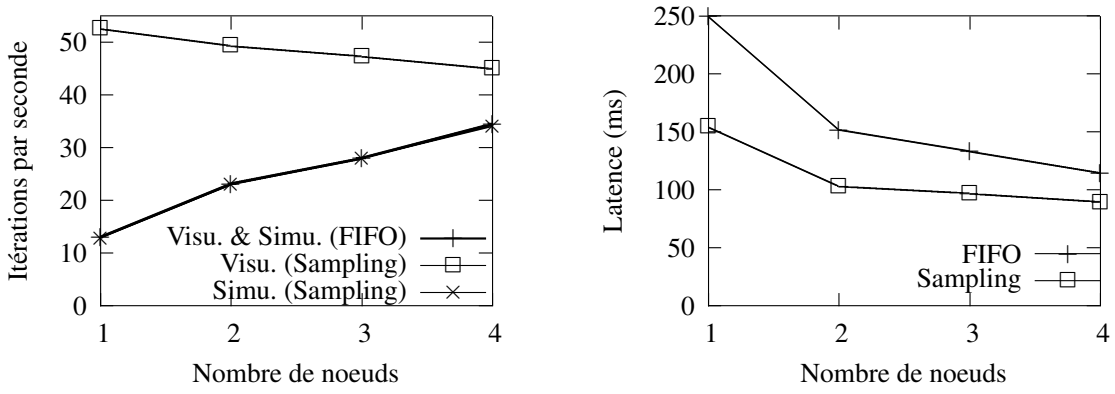
Comme indiqué dans la section 7.1.1 (page 65), le noyau Linux 2.4 ne supporte pas les primitives de synchronisation inter-processus. Dans ce cas, FlowVR utilise une implantation basée sur des attentes actives qui entraînent un surcoût important dès que plusieurs modules sont lancés sur une même machine. Ainsi, une application comportant deux modules tourne en moyenne 30 % moins vite sur un noyau 2.4 que sur un 2.6, et cette différence s'accroît considérablement pour plus de modules. Cette implantation permet donc de développer et tester de petites applications, mais une utilisation plus poussée de FlowVR sous Linux requiert le noyau 2.6.

7.4.3 Couplage parallèle Simulation / Visualisation

Une mesure importante concerne les performances du couplage entre une simulation et visualisation parallèles, comme notre application d'exemple (figure 6.1 page 50), en fonction de la politique de synchronisation utilisée. Deux politiques ont été testées :

FIFO : Les données de la simulation sont utilisées une par une, tous les modules s'exécutant donc à la même vitesse (figure 6.4).

Sampling : La visualisation utilise les dernières données disponibles et tourne aussi vite que possible (figure 6.6). Dans le cas où la visualisation est distribuée sur plusieurs



(a) Fréquence

(b) Latence

Figure 7.6 *Performance du couplage simulation / visualisation avec un réseau d'échantillonnage cohérent et un réseau FIFO.*

machines cet échantillonnage est effectué de manière cohérente (les mêmes données sont utilisées sur toutes les machines).

La figure 7.6 présente les résultats pour un nombre de machines affectées à la visualisation et à la simulation variant de 1 à 4. Les détails de cette expérimentation ont été publiés à Euro-Par 2004 [9].

On remarque que la politique *Sampling* permet d'exécuter la visualisation plus rapidement, ce qui était attendu. En ce qui concerne la latence, on observe que la politique *Sampling* est aussi avantageuse, du fait du choix de la dernière donnée disponible, alors que la politique *FIFO* utilise toujours la donnée bufferisée la plus ancienne. Dans les deux cas les performances augmentent avec le nombre de machine utilisée, mis à part pour la visualisation dans le cas *Sampling*, qui est ralentie au fur et à mesure que la simulation est plus rapide, du fait du coût d'envoi des données sur la carte graphique.

Expérimentations 8

Pour valider l'architecture proposée, cette section présente son utilisation au travers du développement de plusieurs applications.

8.1 La plateforme GrImage

La plupart des applications développées au cours de ma thèse l'ont été sur la plateforme GrImage (figure 8.1). C'est une plateforme regroupant plusieurs caméras firewire (jusqu'à une dizaine) et un mur de 16 vidéo-projecteurs, tout ceci piloté par 11 Bi-Xeon 2.66 GHz et 16 Bi-Opteron 2.0 GHz équipés de cartes graphiques Geforce 6800 Ultra. Ces machines sont reliées par un double réseau Ethernet Gigabit. Un lien 10 Gigabits relie cette grappe à deux autres grappes du laboratoire, l'une avec 100 Bi-Itanium 2 et l'autre regroupant 48 Bi-Xeon 2.4 GHz. Ces grappes font aussi parties du projet Grid 5000, dont l'objectif est de rassembler 5000 processeurs répartis sur plusieurs sites en France.

Le but de la plateforme GrImage est de fédérer les travaux des équipes de recherche en parallélisme, vision, graphisme et animation autour de projets communs.

8.2 Applications graphiques simples

Il arrive fréquemment qu'une application graphique existante relativement simple, souvent programmée en OpenGL, doive être adaptée pour fonctionner sur une plateforme parallèle comme un mur d'image. En considérant que cette application n'utilise pas beaucoup de données en entrée, ni des calculs trop complexes, seule la partie rendu doit être

Figure 8.1 *Matériels utilisés sur la plateforme GrImage.*

modifiée. Deux problèmes principaux se posent :

- le rendu parallèle, i.e. fournir les données de la scène à chaque carte graphique ;
- la configuration des vidéo-projecteurs (matrices de positionnement, optionnellement masques de recouvrement).

Plusieurs solutions sont possibles :

- exécuter l'application originale sur une machine et utiliser Chromium pour distribuer les commandes OpenGL ;
- porter l'application sous Net / VR Juggler qui la répliquera sur chaque machine ;
- ajouter des ports de communications FlowVR pour synchroniser les données internes.

Chacune de ces solutions a des avantages particuliers. Si Chromium est disponible sur la plateforme, alors la première solution est la plus rapide à implanter, mais risque d'avoir un surcoût important si la scène est très dynamique. Porter l'application sous Net Juggler permet de ne pas être dépendant de ce facteur mais demande parfois des modifications profondes dans le code de l'application. Enfin la dernière solution impose de réajuster pour chaque application le code nécessaire à la synchronisation des données et à l'ajustement des projecteurs. Cependant, ce code est assez simple et tient en quelques dizaines de lignes.

Plusieurs applications ont été adaptées au mur d'image en utilisant Net Juggler. Plus récemment l'utilisation de ports FlowVR a été employée par exemple pour adapter Ani-

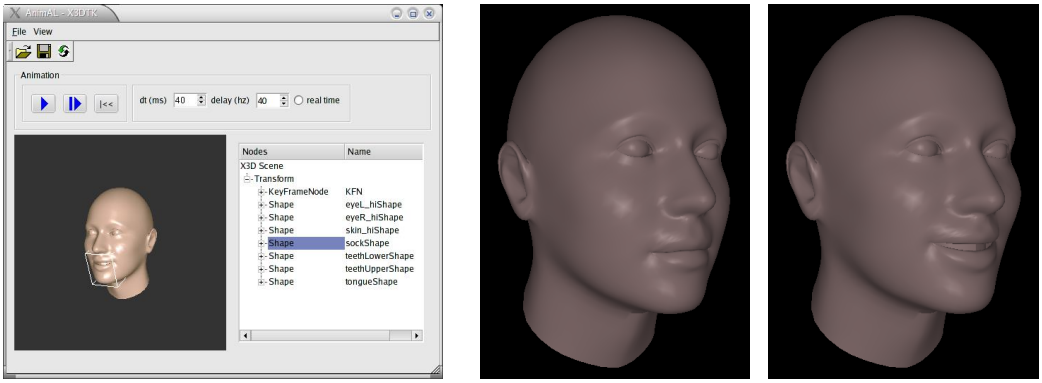


Figure 8.2 *Animation faciale simple utilisant AnimAL.*

mAL [57] (figure 8.2), le moteur d'animation développé dans le cadre du projet EVASION du laboratoire GRAVIR de l'INRIA Rhône-Alpes.

Quel que soit la méthode choisie, une fois l'application intégrée dans l'architecture FlowVR il est possible de l'étendre facilement en la couplant avec d'autres composants. En ajoutant quelques ports d'entrée l'application peut recevoir des données produites par les modules de reconstruction multi-caméras par exemple. Que l'application soit répliquée ou bien instanciée sur une seule machine via Chromium n'affecte alors que le réseau d'interconnexion FlowVR. Enfin, pour intégrer cette nouvelle application à une autre application plus conséquente, il est possible d'ajouter des ports de sortie communiquant les données calculées, qui seront alors intégrées à la visualisation du reste de l'application, court-circuitant l'ancien affichage OpenGL local. L'ancienne application intègre donc l'architecture en tant que nouveau module appartenant à la tâche *Calculs* dans ce cas (chapitre 2 page 7).

8.3 Reconstruction 3D temps réel

La première application développée exploitant toutes les ressources de la plateforme GrImage fut la parallélisation d'une méthode de reconstruction 3D. L'objectif est de calculer en temps réel l'enveloppe visuelle (volume) 3D d'un objet à partir des images obtenues par plusieurs caméras le filmant. Nous avons utilisé un algorithme de reconstruction surfacique basé sur une extraction de silhouettes (images noir et blanc séparant les pixels correspondant à l'objet à reconstruire du fond) développé par Edmond Boyer et Jean-Sébastien Franco [61]. Clément Ménier a implanté une parallélisation de la phase de reconstruction en utilisant MPI [62] que nous avons intégrée grâce à FlowVR aux étapes d'acquisition et de traitement des images ainsi qu'une visualisation utilisant Net Juggler. L'application complète est présentée sur la figure 8.3. Ce travail a été publié lors d'IPT 2004 [5].

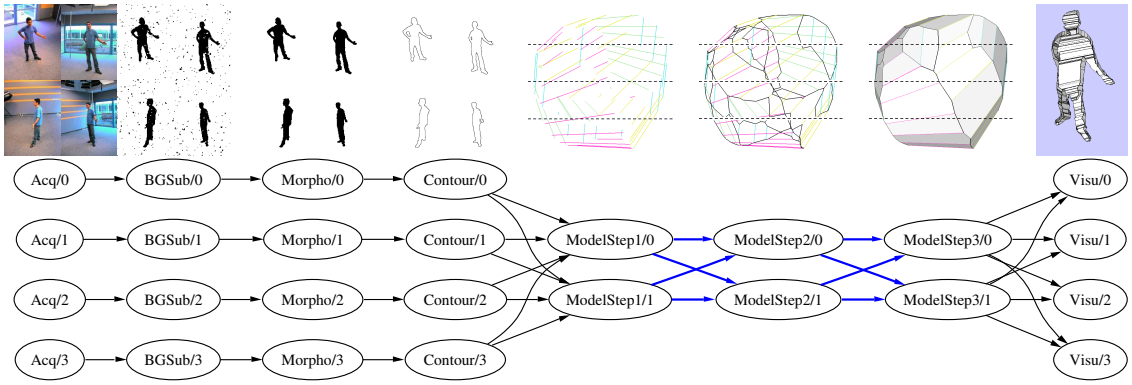


Figure 8.3 *Graphe de flux de données de la reconstruction parallèle. Les liaisons en bleu entre les modules Model sont implantées avec MPI, le reste utilise FlowVR.*

La première phase de cette application concerne le traitement des images des caméras. Ce traitement est effectué en local sur chaque machine contrôlant une caméra, ce qui évite de transférer les flux d'images sur le réseau. Les différentes étapes de ce traitement (acquisition, soustraction de fond, morpho-maths, vectorisation des contours) sont implantées par des modules FlowVR distincts. Malgré la taille importante des données échangées, le surcoût de ce découpage est faible (quelques microsecondes) du fait de l'utilisation d'une mémoire partagée. Cette modularité a permis de faire évoluer chacune des étapes de manière indépendante. Ainsi la plupart des modules ont été réimplantés ou optimisés par différentes personnes au cours de l'évolution de l'application. Par exemple, le module d'acquisition est implanté en utilisant la librairie *Blinky* [51] pour contrôler les caméras Firewire de GrImage. Une version alternative utilise l'API *Video4Linux* pour accéder à des caméras USB, de manière transparente pour les autres modules. De même, le module de vectorisation de contour a été étendu pour permettre de calculer une vectorisation approximative [46], et ainsi contrôler la précision de la reconstruction pour maîtriser la durée des calculs en fonction des besoins de l'application.

La deuxième phase concerne l'algorithme de reconstruction lui-même. Sa parallélisation est basée sur un pipeline en 3 étapes et un découpage en tranches de l'espace 3D. Les communications liées à cette parallélisation sont effectuées par MPI. Les différents processus MPI sont vus comme un ensemble de modules dans l'application FlowVR. Les contours issus des différentes caméras doivent être diffusés vers une partie de ces modules. Le résultat de cette phase, le modèle 3D reconstruit, est produit de manière découpée par les modules correspondant à la dernière étape. Ce modèle est ensuite diffusé vers les modules de visualisation, instanciés sur chaque machine contrôlant un vidéo-projecteur du mur d'image.

Avec 6 caméras capturant des images 640×480 à 30 images/seconde, en dédiant 8 processeurs à la phase de reconstruction celle-ci est effectuée en temps réel (entre 30 et 20 fois par secondes, en fonction de la complexité du modèle reconstruit) avec une latence

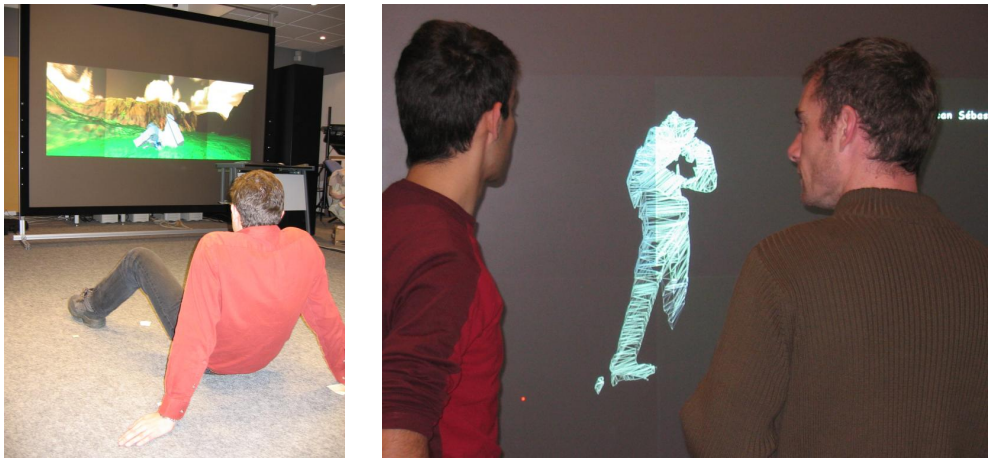


Figure 8.4 *Reconstruction 3D temps réel sur GrImage. Le mur d'image permet de visualiser les détails du modèle. La précision obtenue peut atteindre 0,5 cm.*

globale inférieure à 150 microsecondes. Ce modèle est ensuite visualisé sur le mur d'image en utilisant une application Net Juggler développée antérieurement (section 5.2 page 39), modifiée en ajoutant les ports nécessaires pour recevoir les nouvelles données. Le résultat est visible sur la figure 8.4.

La reconstruction étant réutilisée pour plusieurs applications, elle a été améliorée au cours du temps. En particulier, les communications MPI entre les phases de reconstruction ont été remplacées par des connexions FlowVR. Ceci permet d'utiliser les informations intermédiaires (comme les points générés par la phase 1) pour effectuer des calculs sans attendre la fin de la reconstruction complète (il est par exemple possible d'exprimer la boite englobante de l'objet reconstruit. Le code interne des différentes étapes a été allégé, du fait qu'il ne doit plus gérer les communicateurs MPI liés aux communications collectives. De plus, les performances ont été améliorées du fait du meilleur contrôle de la bufferisation des messages par FlowVR. Cette deuxième version a été présentée lors d'IPT 2005[6].

8.4 Interactions

Une fois les données 3D de l'utilisateur disponibles en temps réel, il est possible de concevoir de nouvelles interactions avec les objets virtuels.

Figure 8.5 *Module d'interaction Carving permettant de sculpter virtuellement un cube qui tourne à la manière d'un potier.*

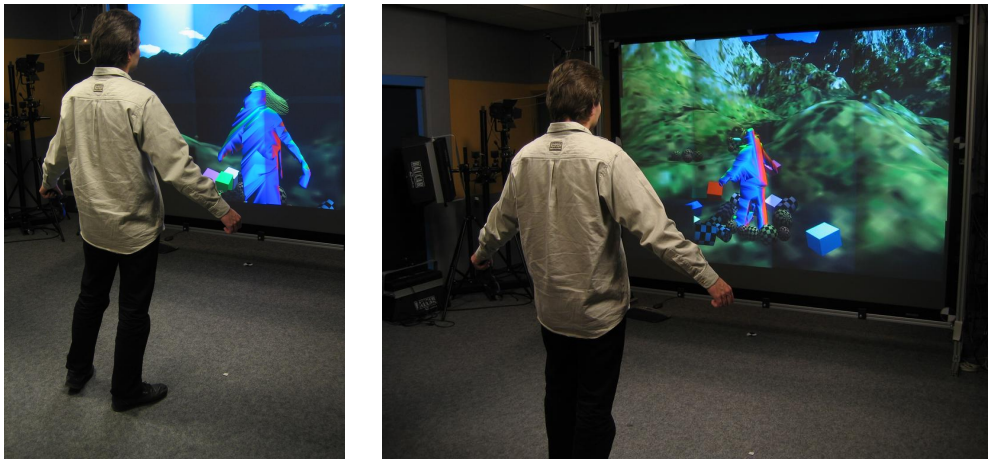
8.4.1 Sculpture

Le premier module développé permet de sculpter un octree virtuel (figure 8.5). Ce module génère un octree qui est modifié dès qu'il entre en contact avec une partie du corps de l'utilisateur. Selon le mode d'interaction il est possible de supprimer de la matière, d'en ajouter ou d'en changer la couleur. L'algorithme utilisé projette récursivement chaque cube dans les images des caméras. Le cube peut alors couvrir des zones pleines sur toutes les images, ou vide sur l'une des images, ou des cas intermédiaires. L'octree est alors modifié en fonction du mode d'interaction.

Cette application a démontré une utilisation nouvelle des informations obtenues sur tout le corps de l'utilisateur. Le fait de pouvoir utiliser tout son corps était très bénéfique. Par exemple modeler une forme de vase est très facile en utilisant le coude. Une vidéo présentant cet exemple est disponible : `http://www-id.imag.fr/~allardj/these/carve.avi`.
La configuration mur d'image est mal adaptée pour cette application du fait du décalage entre la zone d'interaction et l'affichage (environ 2 mètres) qui était donc peu immersif. Il est alors nécessaire de déplacer le point de vue virtuel pour mieux voir les actions effectuées, ce qui désoriente l'utilisateur.

8.4.2 Simulation de cheveux

Les autres interactions que nous avons testées concernent le couplage entre la reconstruction 3D et des simulations physiques. Le premier couplage, assez simple, fut de localiser et suivre la position de la tête et de la relier à une simulation de cheveux pour faire une sorte de perruque virtuelle (figure 8.6(a)). La simulation de cheveux était implantée au préalable dans le cadre de la thèse de Florence Bertails [23]. Le module de calcul de la position de la tête utilise une technique très simple basée sur l'utilisation du point le



(a) (b)

Figure 8.6 *Couplage de la reconstruction de l'utilisateur avec des simulations physiques : (a) cheveux virtuels ; (b) objets rigides.*

plus haut du modèle reconstruit. Pour ajouter de la précision une reconstruction voxelique autour de ce point est utilisée pour approximer un centre de masse beaucoup plus stable.

Ce travail a permis de mettre en lumière les problèmes de couplage multi-fréquences. En effet, la reconstruction, et donc le suivi, tourne approximativement à 30 Hz alors que la simulation nécessite 100 Hz. En utilisant un simple échantillonnage, la position en entrée de la simulation sera constante pendant quelques itérations puis va "sauter" à la position suivante. Ces sauts rendent la simulation de cheveux instable. Nous avons donc utilisé des filtres d'interpolation pour adoucir ces mouvements. Plusieurs solutions sont possibles, chacune offrant des caractéristiques de continuité et latence différentes. Une méthode qui est apparue comme le meilleur compromis dans ce cas est un filtrage par amortissement, calculant à chaque itération une position qui est une combinaison linéaire entre la dernière position calculée par le suivi et la position utilisée lors de l'itération précédente. Le coefficient utilisé pour combiner les deux valeurs permet d'adapter le filtre en fonction de la latence et de la stabilité voulue. Ce filtre peut aussi être utilisé en plus en sortie du module de suivi pour lisser les erreurs, dans le cas ou la reconstruction n'est pas stable (mauvaise soustraction de fond).

Cette application fut présentée dans le cadre de la Fête de la Science 2004 sur le campus de Grenoble. Environ 250 personnes ont pu vivre l'expérience d'être reconstruit dans un monde virtuel et jouer à la fausse blonde. Pour cet évènement nous avons utilisé une configuration impliquant deux sites distants. En effet, seule 7 machines étaient présentes sur le campus pour piloter 4 caméras et 2 vidéo-projecteurs. Tous les calculs de reconstruction et de simulation étaient effectués par 8 machines de la grappe GrImage, au travers d'un lien Gigabit de 13 kilomètres du projet *VTHD*. Du fait de sa rapidité ce lien n'introduisait pas une grande latence additionnelle. En une journée de

préparation et deux jours de démonstration cette application a tout de même utilisé environ deux Tera-octets de bande passante. Une vidéo de cet évènement est disponible : http://www-id.imag.fr/~allardj/these/sef2004.avi.

8.4.3 Collisions

Un deuxième couplage plus complexe concerne la gestion des collisions entre le modèle reconstruit et les objets rigides de l'environnement virtuel (figure 8.6(b)). Ce type d'interaction permet d'obtenir un sentiment de présence dans l'environnement virtuel, du fait que le corps de l'utilisateur ne passe plus au travers des objets mais a une influence réaliste sur eux. Nous utilisons un module calculant les collisions entre objets en testant les triangles de chaque objet avec des champs de distance signée des autres objets [77, 35]. Pour calculer ce champ de distance pour le modèle reconstruit, nous utilisons une reconstruction voxelique suivit d'un module de calcul de distance signée basé sur l'algorithme présenté dans [155].

Les collisions obtenues entre l'utilisateur et les objets virtuelles sont relativement réalistes, toutefois il manque l'information de vitesse de déplacement de l'utilisateur, ce qui fait qu'il n'est pas possible de donner une impulsion aux objets, comme taper dans un ballon. En revanche, quand ce sont les objets qui se déplacent la collision est réaliste. Une vidéo présentant cette application est disponible : http://www-id.imag.fr/~allardj/these/collision.avi.

Ce dernier couplage, ainsi que des couplages intégrant plusieurs simulations simultanément, sera développé dans la partie IV (page 133).

8.5 Contrôle et paramétrage

Contrôler tous les paramètres des applications les plus complexes devient vite problématique. Utiliser des raccourcis clavier ou les boutons d'un Pad n'est pas facile à maîtriser car ces actions changent en fonction de l'application. Pour résoudre ce problème nous avons développé une interface 2D assez simple offrant un certain nombre d'éléments de contrôle reliés à l'activation de certains modules, des paramètres des simulations physiques ou de filtrage des données (figure 8.7). Chaque élément est lié à un port de sortie qui est ensuite connecté vers le bon module ou filtre dans l'application. Un autre module permet de recevoir d'autres évènements liés aux boutons du Pad qui peuvent aussi être connectés à certains paramètres utilisés fréquemment. L'influence de ce contrôle sur l'exécution de l'application est visible sur la figure 8.8. Ce type de paramétrage pendant l'exécution est important pour tester l'utilité de chaque fonctionnalité séparément.

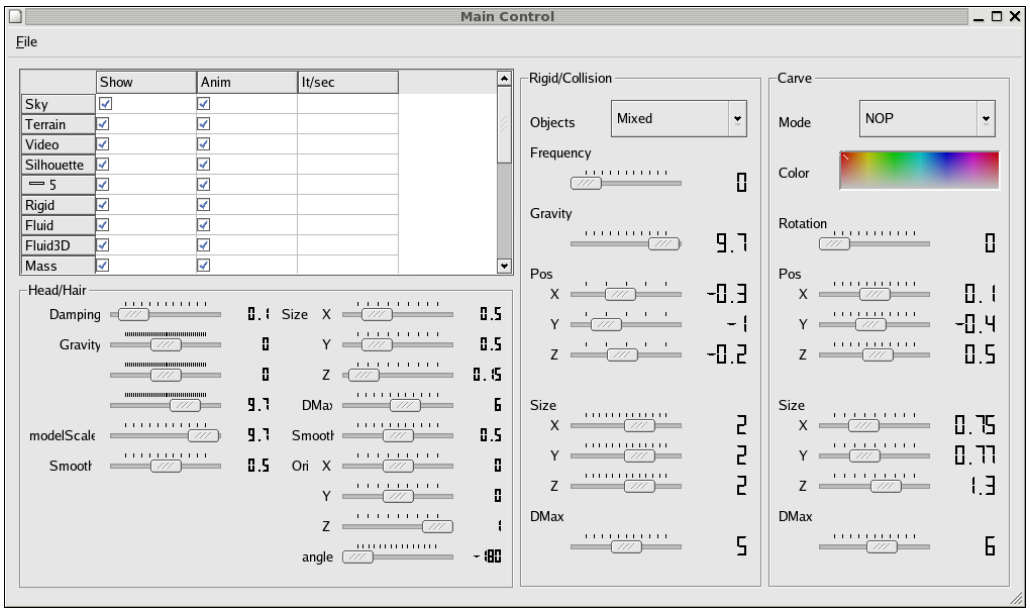


Figure 8.7 *Interface utilisateur 2D de contrôle de l'application.*

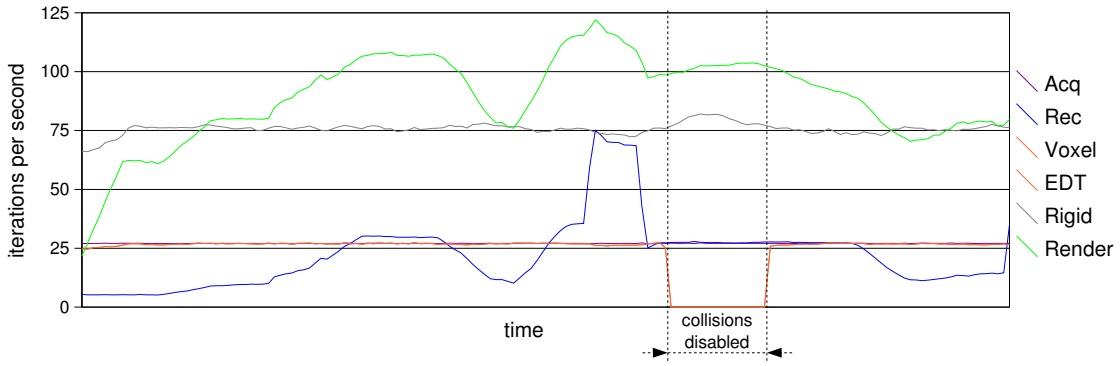


Figure 8.8 *Fréquence d'activation de chaque partie de l'application de collisions. En utilisant l'interface de contrôles certaines parties peuvent être désactivées.*

Bilan 9

Nous avons présenté FlowVR, un middleware dédié au couplage de codes parallèles interactifs. Il repose sur un modèle simple, représentant une application sous forme d'un graphe de flux de données reliant des *modules* et utilisant des *filtres* pour implanter des communications collectives et des opérations sur les données ; et des *synchroniseurs* pour appliquer des politiques de synchronisation entre les différentes parties de l'application.

La construction de l'application se fait en deux étapes distinctes. Les modules sont programmés le plus indépendemment possible du reste de l'application. Ils proviennent souvent de codes existants. Une API de programmation très simple leur permet de déclarer des *ports* de communication. Ils utilisent ensuite un modèle itératif. Entre chaque itération ils appellent la méthode *wait()*. C'est une opération bloquante dont la sémantique est de lire un nouveau message sur tous les ports d'entrée du module. Ensuite le module peut lire le message de chacun des ports d'entrée et utilise la méthode *put()* pour envoyer au plus un message sur chaque port de sortie de façon non bloquante.

Ces modules sont ensuite connectés entre-eux pour former l'application complète. Les connexions sont point à point mais peuvent passer par des noeuds de routages et des filtres pour exprimer les différents types de communications collectives et au besoin les conversions ou répartitions de données.

Un même module peut être utilisé pour un calcul hors-ligne en temps dilaté comme dans une application interactive. La différence se retrouve souvent dans les politiques de synchronisation entre les différentes parties de l'application. Dans un calcul hors-ligne les connexions sont souvent *FIFO*, chaque module travaillant de manière synchronisé sans perte de donnée. Dans une application interactive en revanche on cherche à obtenir le maximum de performance, même si une partie comme une simulation coûteuse ne peut pas suivre le même rythme. Dans ce cas on utilise des mécanismes d'échantillonnage des

données pour désynchroniser chaque partie. Ses mécanismes sont adaptés en fonction de la nature des données et des exigences de cohérence de l'utilisateur.

FlowVR utilise un système de spécification de ce graphe d'application utilisant des descriptions XML générées via un ensemble de scripts. Ces graphes peuvent être construits morceau par morceau et offrent une représentation visuelle de l'application.

En découpant les applications en tâches génériques nous avons construit des applications de plus en plus ambitieuses en réutilisant au maximum des composants d'anciennes applications ou d'autres codes existants. Par exemple la simulation de fluide 2D parallèle, qui fut développée en tant qu'exemple d'application Net Juggler il y a quatre ans, a été convertie en module FlowVR et utilisée depuis dans plusieurs applications sans retoucher au code. Ceci réponds bien à la problématique initiale, à savoir le développement d'un outil permettant le couplage de composants parallèles à l'intérieur d'une application interactive.

Grâce à des concepts repris des méthodes de couplage par composants et adaptés pour l'interaction, FlowVR permet de combiner de nombreux travaux existants issus des domaines comme le calcul parallèle et la réalité virtuelle. Ainsi les applications les plus récentes intègrent des modules de traitement d'images et de vision 3D ; des simulations parallèles de fluide, de tissus, de collisions, de cheveux utilisant MPI, Athapascan, AnimAL ; une extraction de données à visualiser parallélisée sous VTK (section 13.1 page 121) ; et enfin un moteur d'affichage OpenGL distribué par Net Juggler ou VR Juggler 2.

D'autres équipes ont développé des applications utilisant FlowVR. Dans le cadre du projet RNTL Geobench, l'équipe de Sabine Coquillart à couplé avec FlowVR un moteur de rendu haptique avec une visualisation sous Amira pour un workbench, utilisant une grappe de PC. Le LIFO en collaboration avec le BRGM travaille sur du rendu de grands terrains ou couches géologiques utilisant plusieurs machines pour extraire les données et les visualiser grâce à différents algorithmes de niveaux de détails dynamiques [71, 72].

Le modèle et l'implantation de FlowVR semble bien convenir à ces applications. Plusieurs améliorations sont toutefois apparues comme nécessaires. Les performances de l'implantation actuelle pourraient être améliorées en optimisant certaines parties. En particulier, l'implantation réseau utilise la couche TCP qui entraîne plusieurs recopies. Cette première implantation pourrait être améliorée en utilisant des travaux plus évolués comme PadicoTM [49] (section 3.1.4 page 21). Le système de spécification des applications par des fichiers XML et des scripts Perl est parfois difficile à maîtriser, surtout pour créer de petites applications. Plusieurs stages en cours essayent de définir une interface permettant de construire les applications de manière plus visuelle en cachant l'ensemble des outils utilisés.

Ensuite pour de applications de grande envergure l'allocation des ressources reste un problème difficile. Le placement des modules sur les machines et le routage des communications reste à la charge de l'utilisateur. Pour réduire cette charge il serait nécessaire

de modéliser les performances du système, peut-être en utilisant les mesures du mécanisme de traces, pour déterminer un placement et une politique d'activation des modules efficace. Dans le cas général ce problème est NP-Complet. Toutefois en ce restreignant à des cas simples (connections FIFO uniquement par exemple), il est possible d'utiliser des techniques comme l'*ordonnancement cyclique* [44], en les modifiant pour intégrer le critère de latence. Le stage de DEA de Christophe Sadoine co-encadré par Denis Tristram va dans ce sens.

Enfin FlowVR est un outil de couplage d'assez bas niveau. Des surcouches spécialisées pour certains types d'applications sont souvent très utiles pour en faciliter le développement. Dans les dernières versions de FlowVR des types de données sont fournis permettant de gérer facilement les vecteurs et les matrices de valeurs en gérant automatiquement les estampilles pour spécifier les dimensions et le type des données. Les couplages à l'intérieur des premières applications, telles que présentées dans ce chapitre, sont réalisés manuellement. Quand un nouveau composant doit être visualisé ou une nouvelle simulation doit interagir avec le reste de l'environnement il faut souvent modifier certaines parties existantes en ajoutant des ports pour communiquer les données concernées. Un système de visualisation et d'interaction plus générique permettrai de faciliter ces ajouts. Les prochaines parties présenteront nos travaux en ce sens.