Université Joseph Fourier

U.F.R Informatique & Mathématiques Appliquées

Institut National Polytechnique de Grenoble

ENSIMAG

I.M.A.G.

ÉCOLE DOCTORALE
MATHÉMATIQUES ET INFORMATIQUE

MASTER 2 RECHERCHE :
SYSTÈMES ET LOGICIELS

Projet présenté par :

Jean-Denis Lesage

Optimiser la latence et la fréquence des applications parallèles interactives

Effectué au laboratoire ID-imag

Laboratoire
Informatique et
Distribution

Date : Mercredi 21 Juin 2006
9 h 00 – Salle F116

Jury : J. Coutaz
J.-F. Méhaut
R. Echahed
D. Vaufreydaz

Encadrants : B. Raffin
J.-L. Roch

Remerciements

Je tiens à remercier particulièrement Monsieur Bruno Raffin et Monsieur Jean-Louis Roch qui m'ont encadré durant ce stage. Je les remercie pour le temps qu'ils m'ont accordé cette année. Les temps de reflexion passés devant le tableau blanc avec eux m'ont souvent permis d'avancer. Je tiens à les remercier aussi pour tous leurs conseils avisés et précieux.

Je remercie également toute l'équipe qui travaille sur FlowVR et la plateforme GRIMAGE : Thomas, Luciano et Clément. Leurs aides et leurs expériences m'ont permis de mener à bien mes expérimentations. Enfin je remercie tous les membres du laboratoire ID, pour leur accueil durant mon séjour dans le laboratoire.

Table des matières

Chapitre 1

Introduction

1.1 Contexte

Les applications de réalité virtuelle permettent d'immerger un utilisateur dans un monde virtuel. Le coeur de ces applications réside dans la boucle d'itération. Cette boucle est formée de trois étapes :
– Entrée : capture des actions de l'utilisateur
– Calculs : mise à jour de l'environnement virtuel
– Sorties : rendu de l'environnement virtuel à l'utilisateur
Cette faculté d'immersion permet à ces applications de faciliter par exemple la formation en permettant aux personnes de rejouer des scénarii. Ces applications sont aussi utilisées dans le milieu industriel pour faciliter la conception de nouveaux produits en permettant de visualiser les prototypes(aéronautique, automobile). Le domaine de la culture profite aussi de l'évolution de la réalité virtuelle pour faire resurgir des vestiges anciens en permettant de les visiter et de les étudier virtuellement.

Divers matériels ont été élaborés pour augmenter l'impression d'immersion de l'utilisateur. Des outils adaptés à différentes échelles ont été créés. On peut citer par exemple les casques de vision (*Head Mounted Display*). Mais aussi les CAVE[12] qui sont des cubes dont les parois sont des écrans. Dans les CAVE, l'utilisateur a une vue à 360° dans le monde virtuel (figure 1.1(b)). Le rendu et la synchronisation entre les différents écrans sont réalisés grâce à une machine dédiée, le plus souvent des Onyx de SGI.

Les murs d'images ont été développés notamment pour répondre aux besoins des applications de visualisation scientifique qui recherchaient un espace d'affichage avec une très forte luminosité et une très grande résolution. Ces environnements sont le plus souvent composés de vidéoprojecteurs ou d'écrans LCD. Le calcul des données à afficher est souvent réalisé en amont de la visualisation. Dans ces environnements, on va rechercher à distribuer le rendu sur plusieurs machines. On remplace alors une machine dédiée par un cluster où chaque machine effectue le rendu d'une zone du mur d'images. Le *Princeton Scalable Wall*[17] est un exemple de mur d'images formé par 8 projecteurs proposant une résolution de 4096x1536 pixels et permettant de faire de la visualisation scientifique.

Le nombre de machines composant les clusters étant de plus en plus important, il est maintenant possible d'envisager des applications interactives plus ambitieuses tirant partie des multiples entrées/sorties et des ressources de calculs des grappes de PC ou grilles. Il devient alors possible de construire de véritables environnements interactifs où l'utilisateur est plongé dans un monde virtuel. Il peut agir dans cet univers virtuel grâce à des interactions complexes. On peut ajouter par exemple dans les applications des simulations d'écoulement de fluide. Les calculs de simulation sont parallélisés sur une grappe pour atteindre des fréquences permettant à l'utilisateur d'interagir avec l'écoulement[10].

L'apparition des cartes graphiques modernes à la fin des années 90, a permis la création de clusters de PC dédiés aux applications de réalité virtuelle. De nombreuses évolutions au niveau matériel et logiciel[25] permettent de monter de telles architectures en facilitant la synchronisation dans les environnements multi-projecteurs, ou la parallélisation des opérations de rendu. Des librairies comme Chromium[15] ou VR Juggler[8] permettent de faire du rendu en parallèle. Avec Chromium par exemple, une machine calcule seule une scène et les directives OpenGL pour son rendu sont ensuite distribuées sur plusieurs machines qui calculent et affichent de la scène. Les outils comme FlowVR[1], ou OpenMask[4] permettent grâce à une programmation modulaire de développer des applications de réalité virtuelle pour de grands environnements. Ces intergiciels, en plus de proposer des solutions de rendu parallèle, proposent des mécanismes pour paralléliser les calculs nécessaires pour réaliser une application de réalité virtuelle complexe et tirer partie de la puissance de calcul d'un cluster ou d'une grille.

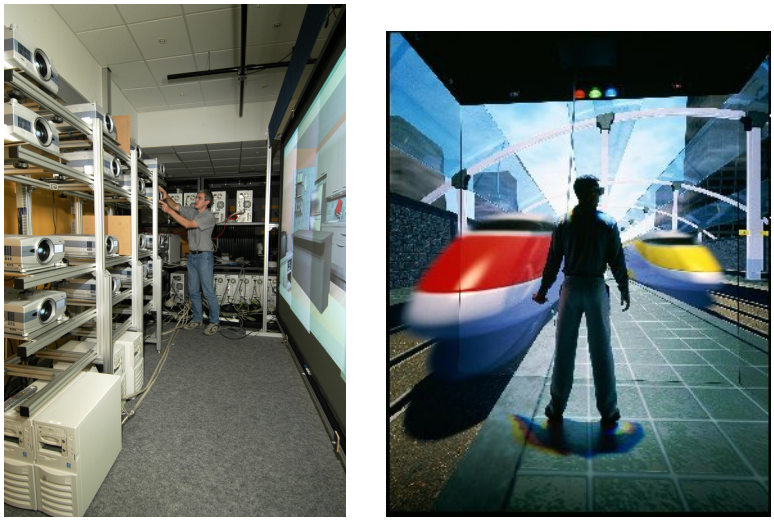

FIG. 1.1 – (a) La plateforme GRIMAGE ; (b) A l'intérieur d'un CAVE

Avec ces évolutions, les applications peuvent comporter plusieurs centaines de modules ou composants (figure 1.2) qui doivent être placés sur des dizaines de processeurs. Le choix d'un placement des modules sur les machines devient alors un problème difficile, alors que le choix du placement a une influence importante sur les performances de l'application. Dans le cas des applications interactives, où les performances ont un impact important sur le sentiment d'immersion de l'utilisateur, ce choix peut se révéler critique. Par exemple, l'OptIPuter[27] relie des machines entre plusieurs universités américaines et

propose un mur d'images d'une résolution de 100 Megapixels. GRIMAGE (figure 1.1(a)) est un exemple d'environnement interactif utilisant un cluster pour paralléliser les calculs. GRIMAGE est composée de 27 machines bi-processeurs équipées de cartes graphiques récentes pilotant un mur d'image formé par 16 vidéo-projecteurs et interagissant grâce à 5 caméras avec l'utilisateur. Potentiellement, GRIMAGE peut être reliée à la grille Grid5000, permettant de profiter de plus de 1000 processeurs supplémentaires pour effectuer des calculs pour une application de réalité virtuelle. Les applications pouvant tirer parti de la puissance offerte par ses deux environnements deviennent tellement complexes qu'il n'est plus envisageable de réaliser le placement manuellement.

A notre connaissance, les intergiciels permettant de construire des applications interactives distribuées ne proposent pas d'outils d'aide au placement des modules sur les machines. Il devient nécessaire de développer ces outils. L'objectif de cette étude est de proposer des solutions pour construire un outil d'aide au placement qui optimise les performances de l'application. Comme critères de performance, nous allons nous intéresser ici à deux paramètres importants sur la sensation d'immersion dans un monde virtuel, la fréquence et la latence.

La fréquence du rendu d'une application mesure la fluidité de l'application. La latence est le temps écoulé entre une action de l'utilisateur et sa répercussion dans le monde virtuel. Pour le visuel, la fréquence cible est de l'ordre de 30-60Hz et la latence est inférieure à 75-150ms[16]. Pour l'haptique la fréquence doit être de l'ordre de 1000 Hz.

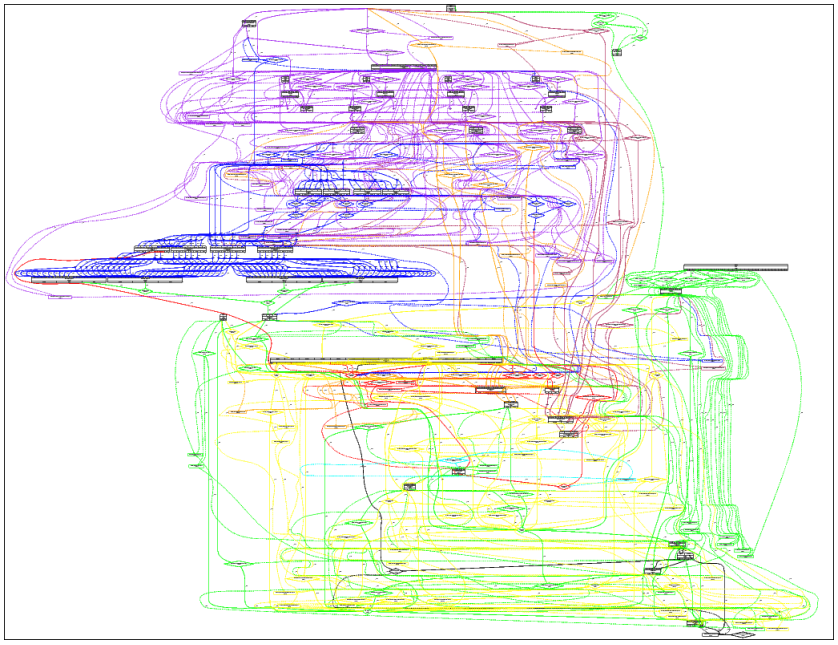

FIG. 1.2 – Graphe d'une application interactive distribuée développée grâce à FlowVR. Chaque sommet correspond à un module qui doit être placé sur une machine.

1.2 Etat de l'art

1.2.1 Visualisation scientifique et réalité virtuelle

Les logiciels de visualisation scientifique comme IRIS Explorer ou OpenDX utilisent un modèle de programmation à flot de données définissant un graphe orienté[19]. Les applications sont découpées en modules qui représentent les sommets du graphe. Ces modules s'échangent des messages ou des flux, ce qui définit des relations de précédence entre les modules. Il est possible grâce à ces logiciels de visualiser des jeux de données complexes qui peuvent représenter par exemple des phénomènes météorologiques, des données médicales ou bien des simulations de fluide. Ces logiciels tirent profil de la représentation par graphe pour passer à l'échelle et permettre l'utilisation du parallélisme pour accélérer les calculs[6]. En effet, il est possible d'implémenter facilement plusieurs types de parallélisme. Le parallélisme de tâches consiste à effectuer des opérations indépendantes simultanément. Le pipeline permet d'accélérer la fréquence des calculs en séparant les étapes d'un calcul complexe sur différentes machines. Le parallélisme de données consiste à diviser un grand jeu de données en zone et d'affecter le calcul de chaque zone à un processeur différent.

La notion de fréquence est assez importante dans ce domaine car l'utilisateur souhaite obtenir une visualisation fluide des données. Les données visualisées sont la plupart du temps calculées indépendamment de la visualisation. Contrairement à la réalité virtuelle, l'impact des contraintes sur la latence pour la visualisation scientifique est donc moins important car il n'est plus possible lors de la visualisation d'interagir avec la simulation.

FIG. 1.3 – Exemple de parallélisme de données avec VTK. Les deux zones de couleur représentent la distribution du travail entre deux processeurs

Des intergiciels comme Stampede permettent de faciliter la parallélisation des applications utilisant des flux. Grâce à la vision modulaire, il est alors possible de proposer à

l'utilisateur des instructions haut niveau[26] pour construire des applications interactives distribuées.

La synchronisation entre les modules peut varier selon les implémentations. Dans FlowVR[1, 7], les modules ont la même fréquence par défaut, celle du module le plus lent. Il est cependant possible de désynchroniser un groupe de modules très lents par rapport au reste de l'application. Dans ce cas, le système aura une fréquence plus élevée. En contrepartie, les messages échangés entre les deux parties ne seront pas produits et consommés durant la même itération. L'utilisation de l'asynchronisme doit alors être faite avec prudence car elle risque d'introduire des incohérences temporelles lors de l'exécution.

Dans OpenMASK, les modules fonctionnent tous à des fréquences différentes. Dans la pratique, l'ordonnanceur du système se ramène à une fréquence commune pour tous les modules. Dans le vocabulaire d'OpenMask, cette fréquence est appelée celle du pas de simulation[20]. Le rapport entre la fréquence d'un module OpenMASK est la fréquence du pas de simulation est un nombre entier. Cela facilite la création de filtres permettant de limiter les phénomènes d'incohérence qui peuvent apparaître avec les différences de fréquence entre les modules. Dans OpenMASK, ces filtres sont appelés des *polateurs*. Ils délivrent des messages à la fréquence du module destinataire en extrapolant ou en interpolant les messages générés par le module expéditeur. Les problèmes de cohérence et l'étude d'extrapolateurs et d'interpolateurs se retrouvent aussi dans les simulateurs ou les jeux vidéo en réseau avec des techniques de dead-reckoning[5]. Le dead-reckoning est une méthode permettant de prévoir la position des objets dans un jeu multijoueur ou une simulation. Chaque participant envoyant des données à des fréquences différentes, les joueurs doivent extrapoler à partir des données reçues la position de tous ses adversaires. Cela permet de garantir à chaque joueur une bonne fréquence tout en limitant les effets d'incohérences introduits par l'asynchronisme.

1.2.2 Temps réel dur

Dans les systèmes temps réel, on retrouve des contraintes de performances qui s'apparentent à celles rencontrées en réalité virtuelle. Dans l'approche temps réel, l'utilisateur propose des contraintes temporelles sur l'exécution d'une application et veut savoir s'il existe un ordonnancement permettant de respecter ces contraintes. Les applications de temps réel dur nécessitent des hypothèses fortes. Par exemple, il est nécessaire d'utiliser des noyaux temps réel permettant maîtriser l'ordonnancement des processus ou bien utiliser du matériel dédié aux systèmes critiques.

Dans ce domaine, on retrouve la notion de latence et de fréquence, car elles peuvent s'exprimer sous forme de contraintes temps réel. Il existe un algorithme qui permet de trouver un ordonnancement optimal pour une machine monoprocesseur en énonçant le problème avec des contraintes de latence[13]. Dans le cas d'une plateforme hétérogène, tous les ordonnancements possibles entre processus sont construits puis la sélection d'un ordonnancement respectant les contraintes peut-être réalisée grâce à une heuristique glouton[29].

Le logiciel SynDEx[28] propose un environnement pour la conception de systèmes temps

réel embarqués sur des architectures hétérogènes distribuées. L'utilisateur décrit son application par un graphe orienté sans cycle (DAG). Il doit définir chaque module par une fonction calculant les valeurs de sortie du module par rapport aux valeurs d'entrée. L'architecture est aussi décrite par un graphe. Dans ce graphe, les noeuds représentent les processeurs et les mémoires. Les arêtes représentent les liaisons physiques. Après avoir défini l'application et l'architecture, l'utilisateur peut imposer des contraintes sur la latence et la fréquence. Si le logiciel trouve un ordonnancement satisfaisant toutes les contraintes, il est possible alors de générer pour chaque processeur un exécutable décrivant toutes les séquences de calculs et de communications.

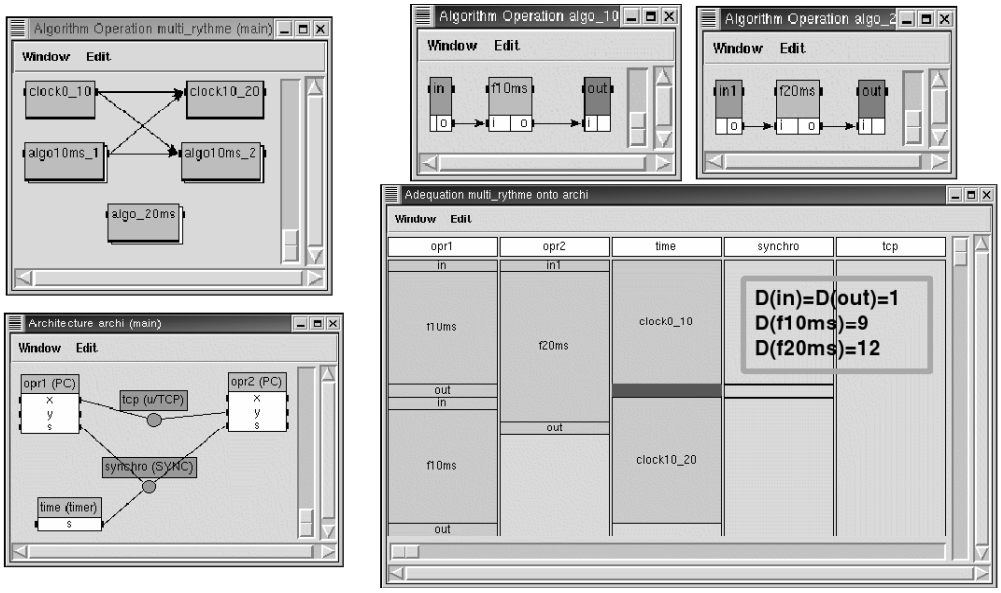

FIG. 1.4 – Capture d'écran du logiciel SynDEx présentant les différents graphes qui décrivent l'application et l'architecture

Les grilles de calculs utilisées pour la réalité virtuelle ont rarement les propriétés du matériel dédié aux systèmes critiques et ne permettent pas de faire les hypothèses nécessaires pour appliquer les méthodes de temps réel. Dans les applications interactives distribuées, les processus sont préemptables. Dans le temps réel, cette hypothèse n'existe pas. Par conséquent, nous ne cherchons pas un ordonnancement précis donnant les dates de début de chaque tâche mais plutôt un placement des tâches sur les processeurs. En effet, les environnements interactifs fonctionnent rarement avec des noyaux temps réel, et nous avons peu de connaissances sur l'ordonnanceur du système.

De plus, les applications de temps réel sont composées de modules dont les performances sont difficiles à prévoir. Certains algorithmes utilisés dans ces applications peuvent avoir des performances qui varient durant l'exécution. Les interactions, la qualité des images, l'utilisation du réseau sont des paramètres qui peuvent faire varier les propriétés du système. Il devient alors très complexe de modéliser une application de réalité virtuelle en utilisant le formalisme du temps réel.

Ces outils de temps réel dur peuvent devenir inadaptés pour la construction d'applica-

tion de réalité virtuelle. Ces applications sont rarement des systèmes critiques, il n'est pas nécessaire d'avoir une vision aussi précise de l'ordonnancement des processus ou des caractéristiques du système. Nous allons donc utiliser un modèle plus simple qui permettra de modéliser les applications à grande échelle.

1.3 Contribution

Le problème de placement est important si on souhaite envisager des applications de réalité virtuelle à grande échelle. Dans cette étude, nous proposons donc une modélisation de ces applications pour formuler les problèmes d'optimisation de la latence et de la fréquence. Nous avons fait la preuve que le problème d'optimisation des deux paramètres est multi-critères dans le cas général. Nous proposons des méthodes pour le résoudre. Nous avons expérimenté la pertinence du modèle et l'efficacité de nos méthodes de résolution sur une application de réalité virtuelle fonctionnant sur la grille GRIMAGE. Nous avons étudié la perspective de proposer un placement adaptif pour résoudre notre problème.

Dans le chapitre 2, nous proposerons une modélisation des applications de réalité virtuelle distribuée. Cela nous permettra d'énoncer les deux problèmes d'optimisation de la fréquence et de la latence. Nous montrerons dans le chapitre 3 que l'optimisation des deux critères est dans le cas général un problème multi-critères. Nous étudierons l'influence de plusieurs hypothèses sur l'existence d'un placement optimal pour les deux critères. Nous allons ensuite proposer au chapitre 4, des méthodes de résolution du problème multi-critères qui donneront un placement qui respecte des contraintes sur la latence et la fréquence. Nous proposerons notamment une méthode utilisant une exploration des solutions par un algorithme de Branch & Bound. Dans le chapitre 5, nous allons vérifier expérimentalement si notre modèle correspond à la réalité. Cela permettra de calibrer le modèle et de tester l'algorithme de placement par Branch & Bound sur une véritable application. Les expérimentations confirment que le problème est multi-critères dans le cas général. En perspective, nous proposerons dans le chapitre 6 une solution qui permettrait d'améliorer le placement en ajoutant du parallélisme. Ce placement serait obtenu par vol de tâches et permettrait d'améliorer les performances de l'application.

Chapitre 2

Formulation des problèmes

Dans ce chapitre, nous allons formuler les deux problèmes d'optimisation de la fréquence et de la latence. Nous étudierons ensuite l'impact des communications sur la formulation des problèmes. Nous verrons ensuite comment cette formulation peut s'adapter aux applications asynchrones.

2.1 Modélisation d'une application de réalité virtuelle distribuée

Une application de réalité virtuelle est composée de modules échangeant des données. Un module est une boucle qui produit à chaque itération des données en consommant les données en entrée du module.

En se basant sur les modèles classiques avec précédence, une itération d'une application distribuée interactive est modélisée par un graphe orienté sans cycle(DAG) où les noeuds sont les modules et les arêtes représentent les communications entre les modules.

L'architecture est composée de machines équipées de processeurs hétérogènes (P_i) de vitesse ν_i (nombre d'opérations / seconde) reliées entre elles par un réseau. La vitesse des processeurs peut dépendre de certains paramètres comme la taille des données à traiter par exemple. Nous considérons ici une valeur moyenne de la vitesse des processeurs. Les modules ont un coût μ_j^i représentant le nombre d'opérations du module i sur le processeur j. Certains modules sont contraints sur un processeur. Par exemple, le module faisant l'acquisition vidéo doit être sur la même machine que la caméra.

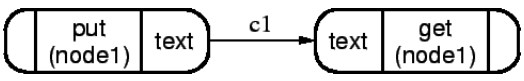

FIG. 2.1 – Le DAG d'une application simple avec deux modules et une transition

2.2 Maximiser la fréquence

La fréquence f est le nombre moyen d'itérations de l'application par seconde. Pour maximiser la fréquence, il suffit de minimiser le temps moyen d'une itération.

La fréquence du système est imposée par le module le plus lent. Sur un même processeur, les modules ont tous la même fréquence.

Soit t_i le temps d'itération moyen des modules placés sur le processeur i, on a $t_i = \frac{\sum_{j \in i}(\mu_i^j)}{\nu_i}$. Il faut donc essayer de minimiser le temps moyen d'itération le plus long.

$f_{freq} : \Phi \longrightarrow \mathbb{R}^+$ est la fonction d'évaluation du temps moyen d'itération d'un placement.

$$\forall \varphi \in \Phi,\ f_{freq}(\varphi) = \max_i(t_i) = \max_i \left(\frac{\sum_{j \in i} \left(\mu_i^j \right)}{\nu_i} \right)$$

Le problème consiste à trouver $\overline{\varphi}$ tel que $f_{freq}(\overline{\varphi}) = \min_{\varphi \in \Phi} f_{freq}(\varphi)$

Dans le cas où deux modules sur des machines distantes échangent des messages, le système doit envoyer les messages sur le réseau. Cet envoi entraîne un surcoût d'opérations (recopie du message sur la carte réseau et envoi des paquets). Nous considérons que les applications permettent de recouvrir totalement ses communications par des calculs. Par conséquent, les communications n'ont pas d'influence sur la fréquence des applications. Néanmoins, comme nous allons le voir, les communications ont un impact important sur la latence de l'application.

2.3 Minimiser la latence

2.3.1 Communications à coût nul

Pour ne pas prendre en compte les communications, nous supposons dans un premier temps que le réseau a un débit infini et une latence nulle.

La latence L est le *makespan* d'une itération. C'est la date de fin de la tâche qui finit le plus tard. Pour cette itération, on pose T^j, la date de fin de la tâche j. On considère que l'itération démarre à l'instant t = 0. La latence est le maximum des T^j. $f_{lat} : \Phi \longrightarrow \mathbb{R}^+$ est la fonction d'évaluation de la latence d'un placement :

$$\forall \varphi \in \Phi, f_{lat}(\varphi) = \max_j(T^j)$$

Le problème consiste à minimiser la latence donc trouver $\overline{\varphi}$ tel que $f_{lat}(\overline{\varphi}) = \min_{\varphi \in \Phi} f_{lat}(\varphi)$.

Si nous plaçons plusieurs tâches sur un même processeur, nous ne pouvons pas connaître l'ordre d'ordonnancement des tâches sur le processeur. Dans ce cas, le calcul des T^j n'est pas faisable. Un calcul du temps de calcul d^j d'une tâche j nécessiterai d'étudier avec précision l'ordonnancement des tâches sur le processeur. On peut néanmoins encadrer d^j avec deux bornes d^j_{min} et d^j_{max}. Si on calcule le plus long chemin du graphe de l'application en attribuant comme poids aux modules respectivement d^j_{min} et d^j_{max}, nous obtiendrons deux valeurs L_{min} et L_{max} qui encadreront la latence L.

Pour chaque tâche j, nous calculons d^j_{min} et d^j_{max} deux bornes telles que le temps de calcul d^j de la tâche j est comprise entre d^j_{min} et d^j_{max}

Si la tâche j est sur le processeur i, le cas où son itération sera la plus rapide, se produira quand elle sera seule sur le processeur i.

$$d^j_{min} = \frac{\mu^j_i}{\nu_i} \Rightarrow d^j_{min} \leq d^j$$

Si plusieurs tâches sont placées sur le même processeur, le pire cas est quand toutes les tâches s'exécutent en même temps. On suppose le processeur équitable. Il va donc attribuer les ressources proportionnellement au travail à effectuer. Les tâches ont toutes la même fréquence.

A l'instant où une tâche j se termine, comme le processeur ordonnance de façon équitable toutes les tâches, le travail total exécuté pour toutes les tâches sera $\sum_{k \in i} \min(\mu^k_i, \mu^j_i)$(figure 2.2) . Connaissant la vitesse du processeur, on peut donc calculer la durée de la tâche j : d^j_{max}.

$$d^j_{max} = \frac{\sum_{k \in i} \min(\mu^k_i, \mu^j_i)}{\nu_i}$$

d^j_{max} est une approximation très pessimiste du temps de calcul du module j. En effet, dans ce modèle, on ne prend pas en compte les dépendances entre les tâches. S'il y a une relation de précédence entre deux tâches, celles-ci ne peuvent pas s'exécuter en même temps. Dans ce cas, d^j_{max} sera alors une évaluation très pessimiste. Cette remarque permet d'expliquer les écarts mesurés dans la partie expérimentations (chapitre 5.2.1 à la page 36).

On peut obtenir une majoration beaucoup plus fine en prenant en compte les dépendances entre les tâches. Les relations de précédences entre les tâches nous permettent de savoir si deux modules peuvent s'exécuter simultanément. Nous n'avons pas eu le temps d'expérimenter cette borne, cela pourra faire l'objet d'une future amélioration des algorithmes programmés lors du stage.

L_{min} s'obtient en calculant un plus long chemin dans le graphe où les modules ont un coût d_{min}. L_{max} s'obtient de la même manière en associant le coût d_{max} aux modules.

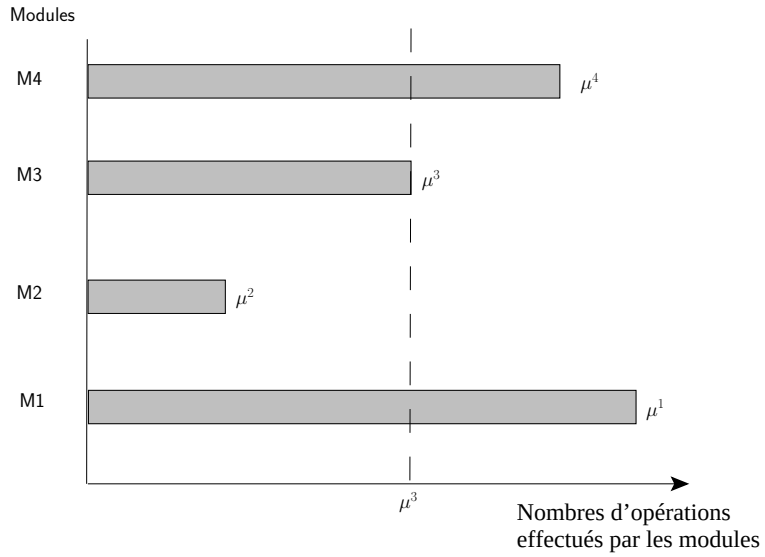

FIG. 2.2 – Calcul du travail total du processeur quand la tâche M3 se termine

2.3.2 Prise en compte des communications

On considère que les communications ont un coût non nul. Pour calculer la latence, nous devons maintenant prendre en compte ces communications. Nous allons définir un paramètre β qui représentera le débit de transfert des messages sur le réseau et prendra en compte l'ensemble des opérations supplémentaires pour envoyer un paquet sur le réseau. On attribue alors dans les arrêtes du DAG un poids (noté $c_{j\to k}$) pour représenter la latence causée par les communications.

Comme pour l'ordonnancement des processus sur les machines, nous n'avons pas de connaissance de l'ordre d'envoi des paquets sur le réseau. Si deux processus envoient un message en même temps à travers une interface réseau, nous n'avons pas connaissance de la séquence d'envoi des paquets par la carte réseau. Par conséquent, nous allons encadrer $c_{j\to k}$ entre deux valeur $c_{j\to k;min}$ et $c_{j\to k;max}$.

On note β (octets / seconde), le débit du réseau. L'envoi le plus rapide est réalisé quand il y a un qu'un seul message en attente dans la carte réseau. Par conséquent, son temps d'envoi sera $\frac{N_{j\to k}}{\beta}$.

$$c_{j\to k;min} = \begin{cases} 0 & \text{si j et k sont sur le même processeur} \\ \frac{N_{j\to k}}{\beta} & \text{sinon} \end{cases}$$

A l'inverse, le cas le pire se présente quand plusieurs messages sont en concurrence lors de l'envoi des messages. Dans ce cas, on considère que l'interface réseau envoi les données de façon équitable et entremêle les messages. La méthode de calcul de $c_{j\to k;max}$ est alors similaire à celui de d^j_{max}. Quand le message de taille N_1 est envoyé, l'interface réseau de la machine i a alors envoyé pour l'ensemble des messages M en attente sur la

carte réseau $\sum_{N_k \in M} \min(N_1, N_k)$ octets. Par conséquent, on en déduit :

$$c_{j\to k;max} = \begin{cases} 0 & \text{si j et k sont sur le même processeur} \\ \frac{\sum_{N_p \in M} \min(N_{j\to k}, N_p)}{\beta} & \text{sinon} \end{cases}$$

Si on affecte maintenant au module soit le coût d_{min} soit d_{max} calculés précédemment (chapitre 2.3.1) et aux arrêtes soit le coût $c_{j\to k;min}$ soit le coût $c_{j\to k;max}$, on peut calculer L_{min} et L_{max} grâce un calcul du plus long chemin. Nous obtenons donc un encadrement de la latence L.

De même que la version proposée de d^j_{max} est très pessimiste car nous ne prenons pas en compte les dépendances entre les tâches, la borne $c_{j\to k;max}$ est elle aussi pessimiste. Une prochaine implémentation des fonctions d'évaluation en considérant les précédences, permettra d'obtenir une approximation plus fine de la borne supérieure de la latence L_{max}.

2.4 Désynchronisation d'une partie de l'application

Il est assez fréquent, qu'une partie de l'application soit désynchronisée par rapport à une autre (figure 2.3) pour optimiser les performances de latence et de fréquence. Cette technique est utilisé par exemple dans les jeux vidéos en réseau grâce aux techniques de dead-reckoning[5]. La désynchronisation de tous les joueurs permet d'obtenir une fréquence donnant l'impression de jouer en temps réel. Les techniques de dead-reckoning permettent de limiter les incohérences pouvant être introduites par cette asynchronisme.

Pour une visualisation de fluide par exemple, les calculs de simulation de l'écoulement étant complexes, ils se réaliseront à basse fréquence. Il est alors nécessaire de désynchroniser le rendu pour qu'il se réalise au minimum à 25 images par secondes. Cela permet à l'utilisateur d'interagir avec le monde virtuel sans être gêné par l'écoulement du fluide. En effet, il ne faut pas que cet écoulement impose sa fréquence au reste de l'application.

Le choix des points d'asynchronisme entre les modules est difficilement automatisable. En effet, cet asynchronisme va créer de l'incohérence. Il faut donc choisir de séparer l'application pour minimiser l'incohérence tout en maximisant les performances. Pour réaliser un algorithme qui fixe les points d'asynchronisme, il faudrait trouver une fonction qui puisse évaluer l'incohérence d'une scène. Mais pour l'instant décider si une scène est incohérente est encore très subjectif et réalisable uniquement grâce à l'intervention d'un opérateur humain. Aujourd'hui, c'est le développeur de l'application qui fixe les frontières entre les sous-parties de son application.

Si on connaît les barrières d'asynchronisme, le problème de placement sur ces applications peut alors être résolu simplement. Il est possible d'isoler chaque partie de l'application sur des ensembles de machines distinctes. Nous avons alors à résoudre un problème

de placement pour chaque partie de l'application. Dans ce cas, nos méthodes de résolution peuvent s'appliquer indépendamment pour chaque partie de l'application.

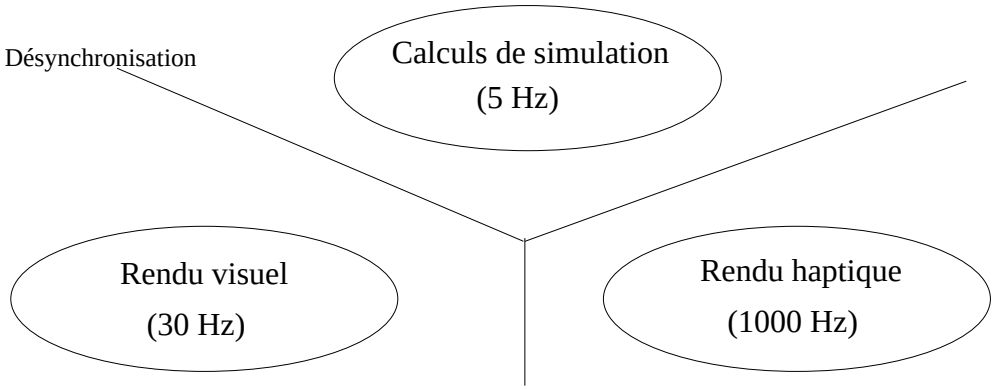

FIG. 2.3 – Un exemple d'application nécessitant de l'asynchronisme

2.5 Résumé

Nous avons dans ce chapitre énoncé deux problèmes d'optimisation. Pour la fréquence, nous avons écrit une fonction d'évaluation du temps moyen d'itération à minimiser.

Pour la latence, il est difficile d'obtenir une fonction d'évaluation exacte. Nous avons donc trouvé un encadrement de la latence qui nous permet de formuler le problème d'optimisation.

Nous avons après raffiné le modèle en prenant en compte des communications entre les modules dans les fonctions d'évaluation.

Cette formulation peut s'adapter au cas des applications asynchrones en considérant les différentes parties de l'application.

Chapitre 3

Optimisation de la latence et de la fréquence : problème multi-critères

Nous avons énoncé deux problèmes optimisant les deux critères de la latence et de la fréquence. Dans ce chapitre, nous allons montrer dans un premier temps que si nous avons une infinité de machines identiques et sans prendre en compte les communications, nous pouvons trouver un placement optimisant les deux critères. Dans les autres cas, nous allons prouver que l'on ne peut pas optimiser les deux critères séparement. Notre problème est donc un problème d'optimisation multi-critères.

Pour prouver ce résultat, nous allons présenter des exemples où il est impossible d'optimiser les deux critères simultanément. Le tableau 3.1 synthétise les résultats de ce chapitre.

Nombre de machines	Machines identiques	Débit du réseau	Existence d'un placement optimisant la fréquence et la latence
∞	oui	∞	oui
fini	oui	∞	contre-exemple
∞	non	∞	contre-exemple
∞	oui	fini	contre-exemple

TAB. 3.1 – Influence des hypothèses sur le problème multi-critères

3.1 Nombre infini de machines identiques et sans communication

Avec ces hypothèses, on peut toujours trouver un placement qui optimise la latence et la fréquence simultanément.

Théorème 1. *Soit un nombre infini de machines identiques (le nombre de machines est supérieur à celui des modules) et des coûts de communications nuls, alors tout placement où il y a au plus un module par machine est optimal pour la fréquence et la latence.*

Preuve. Prenons un placement quelconque, φ

Soit μ^{max} ($\mu^{max} = \max_j(\mu^j)$) le coût du module nécessitant le plus grand nombre d'instructions par itération. Il existe deux cas possible pour le placement de ce module :

- Le module le plus coûteux est seul sur une machine i, dans ce cas, le temps d'itération moyen t_i sur cette machine est $\frac{\mu^{max}}{\nu}$. Ce temps correspond au temps d'itération moyen de l'application pour φ :
$$f_{freq}(\varphi) = \max t_i \geq \frac{\mu^{max}}{\nu}$$
- Le module le plus coûteux n'est pas seul sur une machine i , dans ce cas, le temps d'itération moyen des modules de cette machine est $\frac{\sum_{j\in i}\mu^j}{\nu} > \frac{\mu^{max}}{\nu}$. Donc le temps d'itération moyen de l'application pour φ est supérieur à $\frac{\mu^{max}}{\nu}$:
$$f_{freq}(\varphi) = \max t_i > \frac{\mu^{max}}{\nu}$$

$\frac{\mu^{max}}{\nu}$ est une minoration de l'évaluation de f_{freq} sur l'ensemble des placements possibles.

Considérons maintenant un placement φ' où il y a au plus un module par machine. Le temps d'itération du module j t^j est égal à $\frac{\mu^j}{\nu}$. Les machines sont toutes identiques de vitesse ν, on a donc
$$f_{freq}(\varphi') = \max_j t^j = \frac{\max_j \mu^j}{\nu} = \frac{\mu^{max}}{\nu}$$

La borne inférieure de la valeur de f_{freq} est atteinte pour tout placement où il y a au plus un module par machine. Les placements où il y a au plus un module par machine optimisent donc la fréquence.

Intéressons nous maintenant à la latence. On note W_∞ le chemin critique d'une itération. W_∞ est le poids du chemin le plus long du DAG représentant une itération quand on affecte le coût μ^j comme poids à chaque module. Ce chemin correspond à la plus grande suite de calculs que l'on est obligé de réaliser séquentiellement. Par conséquent, comme on a des machines identiques de vitesse ν, ce chemin peut être calculé au minimum en $\frac{W_\infty}{\nu}$ unités de temps.

Par conséquent, quelque soit le placement φ, on sait que $L \geq \frac{W_\infty}{\nu}$. On a donc trouvé un minorant de la latence.

Soit un placement φ' tel qu'il y a au plus un module par machine. S'il n'y a qu'un seul module par machine, on sait que la latence $L = L_{min}$ (voir chapitre 2.3.1). L_{min} est le plus long chemin dans le DAG d'une itération en affectant $\frac{\mu^j}{\nu}$ comme poids aux modules. D'où $L = L_{min} = \frac{W_\infty}{\nu}$.

La minoration $\frac{W_\infty}{v}$ de la latence est atteinte par tout placement où il y a au plus un module par machine. Ces placements optimisent donc la latence.

On en déduit donc que tout placement où il y a au plus un module par machine est optimal pour la fréquence et la latence.

□

3.2 Problème multi-critères

Pour le cas précédent, on connaît des placements qui optimisent les critères de fréquence et de latence simultanément. Mais ce résultat est obtenu sous des hypothèses fortes non réalistes. Nous allons étudier maintenant des cas plus réalistes en considérant trois hypothèses :
– le nombre fini de machines
– des machines différentes
– des communications à coût non nul
Nous allons voir qu'il suffit qu'une de ces trois hypothèses soient vérifiée pour trouver des applications où il n'existe pas de placements optimisant la fréquence et la latence en même temps. On pourra en conclure que dans la cas général, le problème est multi-critères.

Théorème 2. *Si une des hypothèses suivantes est vérifiée :*
– *le nombre de machine est fini*
– *les machines sont différentes*
– *les communications ont un coût non nul*
alors il existe des applications pour lesquelles il est impossible d'optimiser la latence et la fréquence en même temps.

Pour prouver ce résultat, nous allons prouver 3 lemmes qui correspondent chacun à une des hypothèses supplémentaires.

3.2.1 Nombre fini de machines

Lemme 1. *Si on a moins de machines que de modules, il existe des application où il n'existe pas de placements optimisant la latence et la fréquence en même temps.*

Preuve. Considérons l'application représentée par la figure 3.1.

En enlevant les placements symétriques, il y a trois types de placement possibles. La fréquence et la latence de chacun de ces placements sont calculées dans le tableau 3.2. On remarque qu'il existe un placement optimisant la latence (figure 3.3) et un autre placement optimisant seulement la fréquence (figure 3.2). □

Modules	Transitions	Processeurs
m1 : $\mu^1 = 2$	**m1** vers **m2**	**p1** : $\nu_1 = \nu = 1$
m2 : $\mu^2 = 1$	**m2** vers **m3**	**p2** : $\nu_2 = \nu = 1$
m3 : $\mu^3 = 1$		

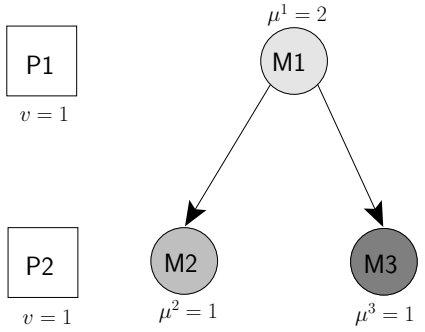

FIG. 3.1 – Application du contre-exemple

Placement des modules	{M1, M2, M3}{}	{M1, M2}{M3} (figure 3.3)	{M1}{M2, M3} (figure 3.2)
1/f (unité de temps)	4	3	2
L (unité de temps)	4	3	4

TAB. 3.2 – Enumération de tous les placements possibles

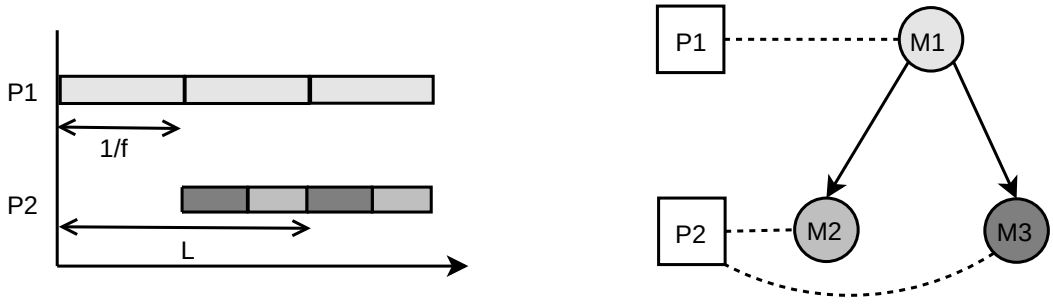

FIG. 3.2 – Chronogramme et placement optimisant la fréquence

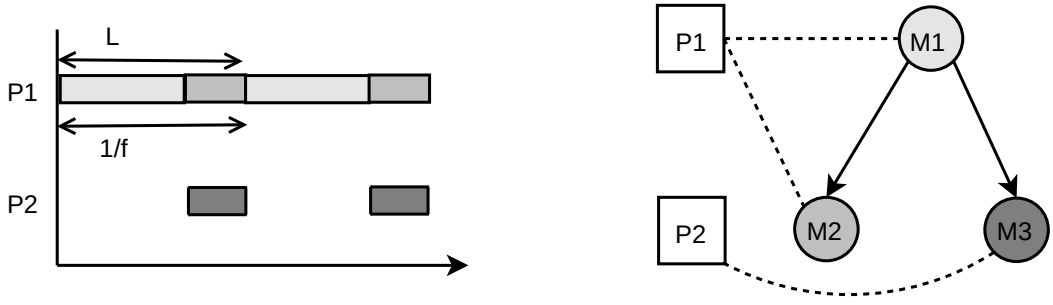

FIG. 3.3 – Chronogramme et placement optimisant la latence

3.2.2 Machines avec des vitesses différentes

Lemme 2. *Dans le cas, où les machines sont de vitesses différentes, il existe des applications où il n'existe pas de placements optimisant la latence et la fréquence en même temps.*

Preuve. Considérons l'application représentée par la figure 3.4) :

Modules	Transitions	Processeurs
m1 : $\mu_1^1 = \mu_2^1 = 60$	**m1** vers **m2**	**p1** : $\nu_1 = 5$
m2 : $\mu_1^2 = \mu_2^2 = 24$		**p2** : $\nu_2 = 12$

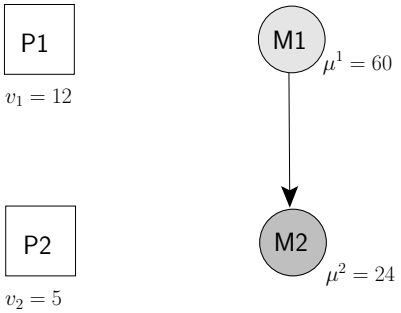

FIG. 3.4 – Application du contre-exemple

Il y a 4 placements possibles énumérés dans le tableau 3.3. On remarque qu'il n'existe pas de placements optimisant la fréquence et la latence en même temps. Le placement optimisant seulement la fréquence est représenté sur la figure 3.5. Le placement optimisant la latence est représenté sur la figure 3.6. □

Placement des modules	{M1, M2}{} (figure 3.6)	{M1}{M2} (figure 3.5)	{M2}{M1}	{}{M1, M2}
1/f (unité de temps)	7	5	12	16,8
L (unité de temps)	7	9,8	14	16,8

TAB. 3.3 – Enumération de tous les placements possibles

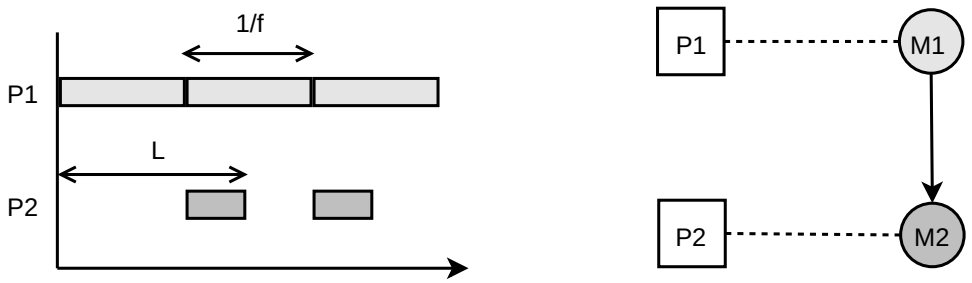

FIG. 3.5 – Chronogramme et placement optimisant la fréquence

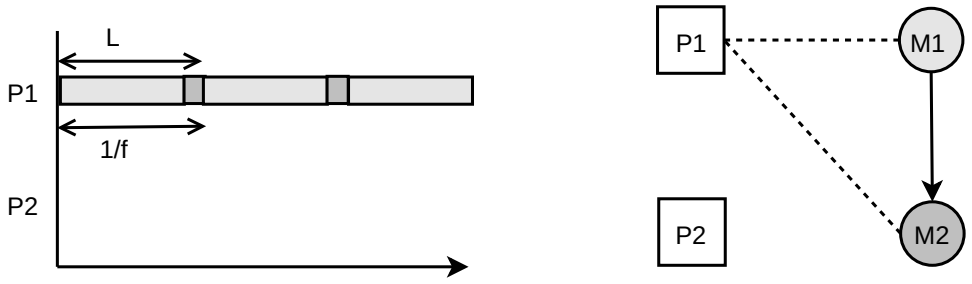

FIG. 3.6 – Chronogramme et placement optimisant la latence

3.2.3 Prise en compte des communications

Lemme 3. *Dans le cadre de machines identiques, il existe des applications où il n'existe pas de placements optimisant la latence et la fréquence en même temps.*

Preuve. Considérons l'application représentée par la figure 3.7

Modules	Transitions	Processeurs	Réseau
m1 : $\mu_1^1 = \mu_2^1 = 5$	**m1** vers **m2**	**p1** : $\nu_1 = 10$	$\beta = 2$
m2 : $\mu_1^2 = \mu_2^2 = 5$		**p2** : $\nu_2 = 10$	

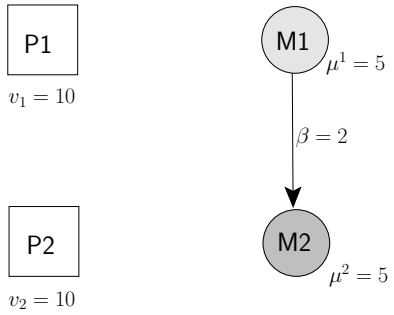

FIG. 3.7 – Application du contre-exemple

Il y a 4 placements possibles énumérés dans le tableau 3.4. Cet exemple n'admet pas de placement optimisant à la fois la latence et la fréquence. Les figures 3.8 et 3.9 présentent respectivement un des placements optimisant la fréquence et un des placements optimisant la latence. □

Placement des modules	{M1, M2}{} (figure 3.9)	{M1}{M2} (figure 3.8)	{M2}{M1}	{}{M1, M2}
1/f (unité de temps)	1	0.5	0.5	1
L (unité de temps)	1	1.5	1.5	1

TAB. 3.4 – Enumération de tous les placements possibles

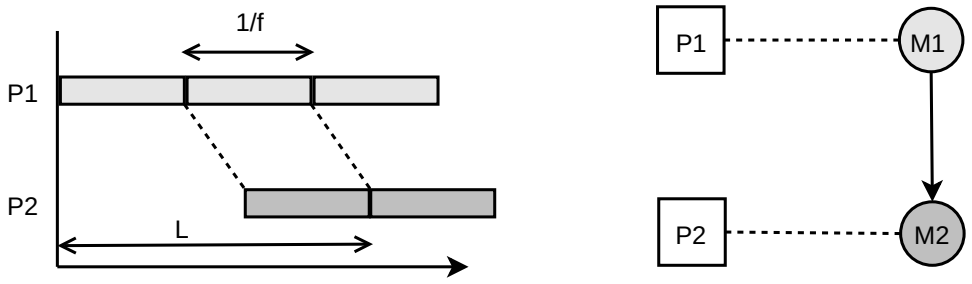

FIG. 3.8 – Chronogramme et placement optimisant la fréquence

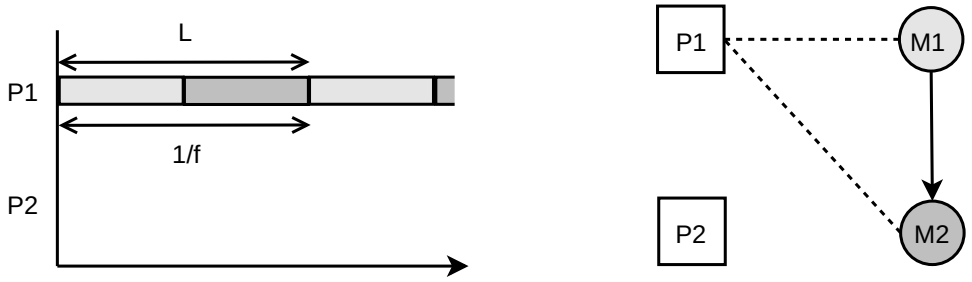

FIG. 3.9 – Chronogramme et placement optimisant la latence

3.3 Résumé

Dans cette partie, nous avons étudié l'influence de plusieurs hypothèses sur l'existence d'un placement optimisant la latence et la fréquence en même temps.

Dans le cadre de notre étude, le nombre de machines est plus petit que le nombre de modules, les machines sont hétérogènes et le réseau a un débit fini. Notre problème d'optimisation de la latence et de la fréquence est donc un problème multi-critères.

Chapitre 4

Résolution

Nous avons vu qu'il n'existe pas de placement optimisant les deux critères simultanément. Dans ce chapitre, nous allons nous intéresser à différentes méthodes de résolution de ce problème multi-critères.

Dans un premier temps, nous regarderons les méthodes proposant un placement sous-optimal en temps raisonnable. Puis grâce à la relaxation du programme linéaire, nous pourrons calculer une garantie sur les critères. Le Branch & Bound permettra d'obtenir une solution garantissant les critères.

4.1 Recherche d'un placement sous-optimal

:

Les problèmes de placements sont en général NP-complet. Il est nécessaire d'explorer l'ensemble des solutions pour obtenir une solution exacte du problème. Certaines instances de problème de placement peuvent aussi être résolues en temps polynomial. Par exemple, il est possible de minimiser la charge totale de travail et de communications pour deux processeurs. Lo a étendu ce résultat en donnant un algorithme de placement polynomial mais pouvant échouer[18].

Nous voulons résoudre les problèmes de minimisation de la latence et maximisation de la fréquence dans le cadre de machines différentes avec communications. Dans ce cadre, il existe des algorithmes pouvant donner un placement sous-optimal en un temps raisonnable. Pour résoudre ces placements sur des plateformes hétérogènes, on peut explorer les solutions grâces à différentes heuristiques[11] (glouton, algorithmes génétiques, recuit simulé, méthode tabou ou A*). Le problème de placement peut également être résolu grâce à une approche "Diviser pour régner"[24]. Par exemple, SCOTCH est un logiciel de placement utilisant cette approche[23].

4.2 Relaxation du programme linéaire

Nous allons écrire les problèmes d'optimisation sans prendre en compte les communications sous forme de programme linéaire à nombres entiers (PLE). Ces PLE peuvent être relaxés pour obtenir des approximations.

On définit la variable X_i^j qui indique si le module j est sur le processeur i.

Soit $X_i^j = \begin{cases} 0 & j \notin i \\ 1 & j \in i \end{cases}$

On écrit dans un premier temps un PLE pour évaluer la fréquence (en évaluant le temps moyen d'itération de chaque module). La fonction d'évaluation de la fréquence nous permet d'obtenir le temps d'itération moyen des modules sur la machine i, $t_i = \frac{\sum_j (X_i^j . \mu_i^j)}{\nu_i}$. Cela donne la contrainte :

$$\nu_i . t_i - \sum_j X_i^j . \mu_i^j = 0 \ \forall i$$

On définit t_α le plus grand temps d'itération moyen parmi les modules ce qui correspond au temps d'itération moyen d'une itération de l'application. On a donc $\forall i, t_\alpha \geq t_i$, d'où les contraintes :

$$t_a - t_i \geq 0 \ \forall i$$

Enfin, les modules doivent être placés sur une et une seule machine. Par conséquent on obtient la contrainte :

$$\sum_i X_i^j = 1 \ \forall j$$

Le PLE s'écrit alors :

$$\begin{array}{rcll} \min \ z_{IP} = c.x & & & \\ \text{s.c. } \nu_i . t_i - \sum_j X_i^j . \mu_i^j & = & 0 & \forall i \\ t_a - t_i & \geq & 0 & \forall i \\ \sum_i X_i^j & = & 1 & \forall j \\ X_i^j & \leq & 1 & \forall (i,j) \\ X_i^j & \geq & 0, \ X_i^j \ entiers & \forall (i,j) \\ c.x & = & t_\alpha & \end{array}$$

Si un module j est contraint d'être placé sur le processeur i, on ajoutera la contrainte $X_i^j = 1$.

Pour la latence, on peut calculer la borne L_{min} grâce à des PLE. On note T^j, la date de fin de la tâche j. d_i^j est la durée de calcul de la tâche j sur le processeur i. On peut alors adapter l'expression de d_{min}^j (chapitre 2.3.1).

$d^j_{i;min}$ est la durée minimum d'exécution de la tâche j sur le processeur i. Les fonctions d'évaluation permettent de calculer ces valeurs.

$$d^j_{i;min} = \frac{X^j_i.\mu^j_i}{\nu_i}$$

On peut alors calculer le temps de calcul total sur toutes les machines de la tâche j. En effet, on a $d^j_i = \sum_i d^j_i$. D'où les bornes :

$$d^j_{min} = \sum_i d^j_{i;min}$$

Soit D^j_{min}, la date de début minimum de la tâche j. S'il existe une transition d'un module k vers j alors le module j doit démarrer après la fin du calcul du module k. Comme la date de fin du module k est T^k on a alors $\forall k \; telque \; k \rightarrow j, \; D^j_{min} \geq T^k_{min}$

On peut remplacer la date de fin d'une tâche par une expression qui est fonction de sa date de début. En effet, on connaît la durée d'exécution d^j d'une tâche, on a donc la date de fin qui est égal à la date de début plus la durée d'exécution ($T^j = D^j + d^j$)

On a donc la relation :

$$T^j_{min} = D^j_{min} + \max(d^j_{i;min})$$

Ce qui se traduit par les contraintes :

$$T^j_{min} - D^j_{min} \geq d^j_{i;min}$$

La latence est la date de fin de la tâche qui finit le plus tard, on a alors

$$L_{min} = \max_j T^j_{min}$$

Ce qui se traduit par les contraintes :

$$\forall j, L_{min} \geq T^j_{min}$$

Comme précédemment, on a les contraintes sur les X^j_i. Les modules doivent être placés sur une seule machine ($\sum X^j_i = 1$).

On obtient le PLE pour optimiser L_{min} :

$$
\begin{array}{rcll}
\min\ z_{IP} = c.x & & & \\
\text{s.c. } \nu_i.d^j_{i,min} - X^j_i.\mu^j_i & = & 0 & \forall(i,j) \\
d^j_{min} - d^j_{i;min} & \geq & 0 & \forall(i,j) \\
D^j_{min} - T^k_{min} & \geq & 0 & \forall k \rightarrow j \\
T^j_{min} - D^j_{min} & \geq & d^j_{min} & \forall(i,j) \\
L_{min} - T^j_{min} & \geq & 0 & \forall j \\
\sum_i X^j_i & = & 1 & \forall j \\
T^j_{min} & \geq & 0 & \forall j \\
D^j_{min} & \geq & 0 & \forall j \\
d^j_{min} & \geq & 0 & \forall j \\
d^j_{i;min} & \geq & 0 & \forall(i,j) \\
X^j_i & \leq & 1 & \forall(i,j) \\
X^j_i & \geq & 0,\ X^j_i\ entiers & \forall(i,j) \\
c.x & = & L_{min} &
\end{array}
$$

Nous n'avons pas trouvé d'écriture du programme linéaire en prenant en compte les communications. L'écriture du programme linéaire de L_{max} est difficile à cause de l'expression de la durée de la tâche j. En effet, il est nécessaire dans l'expression $d^j_{i;max} = \frac{\sum_{k\in i}(\min(X^j_i.\mu^j_i, X^k_i.\mu^k_i)}{\nu_i}$ de supprimer l'opérateur min. Pour le supprimer, une solution est de décomposer le domaine des solutions en sous domaines où la valeur du minimum est connu. Nous n'avons pas écrit ce programme linéaire, car il ne nous semblait pas primordial de développer cette méthode pour L_{max}. En effet, cette borne est comme nous allons le voir dans la partie expérimentation, très pessimiste. Par conséquent, il nous semble plus interressant d'affiner cette borne avant d'écrire le programme linéaire. En outre, dans la partie expériementation, nous verrons que la méthode de Branch and Bound tronqué donne des approximations intéressantes pour cette borne. L'écriture de ce programme linéaire est donc une perspective envisageable.

Nous avons formulé les problèmes grâce à des programmes linéaires à nombres entiers. La résolution de ces programmes donne la solution optimale. Dans l'optique d'obtenir une garantie sur la meilleur fréquence ou la meilleure latence, nous pouvons rechercher une minoration de la latence optimale ou du temps moyen d'une itération.

Nous pouvons relaxer ces problèmes en acceptant que les X^j_i soient réels[22]. La solution z_{IP} des deux problèmes de minimisation sera forcement supérieure à z_{LP}, la solution optimale du programme relaxé. Les problèmes relaxés peuvent facilement être résolus. En faisant une résolution de ces systèmes grâce à un logiciel (GNU Linear Programming Kit par exemple), on obtient une minoration de la latence la plus faible et une majoration de la plus grande fréquence.

4.3 Branch and Bound tronqué

Nous pouvons calculer une borne sur la meilleure fréquence ou la meilleure latence grâce à une exploration des solutions par un Branch and Bound tronqué. En arrêtant l'algorithme, nous obtenons une évaluation optimiste de l'ensemble des placements Φ.

On définit $\overline{f_{freq}}$ et $\overline{f_{lat}}$, les évaluations optimistes du temps d'itération moyen et de la latence. Soit φ un ordonnancement partiel, où k modules sur les N modules sont placés. Pour tout placement φ' conservant le placement φ et plaçant les N-k autres modules, on a

$$f_{freq}(\varphi') \geq \overline{f_{freq}(\varphi)}$$

$$f_{lat}(\varphi') \geq \overline{f_{lat}(\varphi)}$$

Pour évaluer la fréquence, on suppose que k modules ont été placés. On pose $\mu^j = \min_i \mu_i^j$ et $t_\alpha = \max_i t_i$. Le travail qui reste à placer est $\sum_{k+1,\dots,N} \mu^k$. On peut ajouter sur chaque machine le travail $(t_\alpha - t_i).\nu_i$ sans augmenter le temps d'itération moyen. Donc si $\sum_{k+1,\dots,N} \mu^k \leq \sum_i (t_\alpha - t_i).\nu_i$, on peut placer le travail sans augmenter le temps d'itération du module le plus lent (t_α).

S'il reste du travail, celui-ci vaut alors $\frac{\sum_{k+1,\dots,N}}{\mu^k} - \sum_i (t_\alpha - t_i).\nu_i$. On partage ce travail restant proportionnellement à la vitesse des processeurs. C'est à dire que le processeur i récupère le travail $\left(\sum_{k+1,\dots,N} \mu^k - \sum_i (t_\alpha - t_i).\nu_i\right) \frac{\nu_i}{\sum_k \nu_k}$. Ce travail sera réalisé à la vitesse ν_i, le temps d'itération supplémentaire par processeur sera alors $\frac{\sum_{k+1,\dots,N} \mu^k - \sum_i (t_\alpha - t_i).\nu_i}{\sum_k \nu_k}$

En déduit donc une expression de la fonction d'évaluation optimiste de la fréquence :

$$\overline{f_{freq}}(\varphi) = t_\alpha + \max\left(0, \frac{\sum_{k+1,\dots,N} \mu^k - \sum_i (t_\alpha - t_i).\nu_i}{\sum_k \nu_k}\right)$$

Pour la latence, la durée d'exécution d'une tâche sera celle de son exécution si elle est seule sur la machine la plus rapide. On évalue donc les modules non placés avec un temps d'itération $\overline{d^j} = \frac{\mu^j}{\max_i \nu_i}$ qui est celui de l'exécution de cette tâche si elle est seule sur la machine la plus rapide. Dans le DAG d'une itération, on affecte au module placés leurs durées d'exécution d^j_{min} ou d^j_{max}. Pour les modules non placés, on affecte le poids $\frac{\mu^j}{\max_i \nu_i}$. Pour les arêtes, on prend les coûts $c_{j\to k;min}$ et $c_{j\to k;max}$. Si on calcule le plus long chemin dans ce graphe, on obtiendra deux bornes de la latence L_{min} et L_{max}.

On peut alors calculer les évaluations optimistes $\overline{L_{min}}$ et $\overline{L_{max}}$.

Grâce à cette méthode, il est aussi possible d'optimiser un paramètre en fixant une valeur minimum au second paramètre. On peut définir les fonctions d'évaluation suivantes :

Pour optimiser la fréquence en assurant une latence minimale inférieure à L_o :

$$\overline{f_{L_o}}(\varphi) = \begin{cases} \overline{f_{freq}}(\varphi) & \text{si } \overline{L_{min}}(\varphi) \leq L_o \\ \infty & \text{sinon} \end{cases}$$

Pour optimiser la latence minimale en assurant une fréquence supérieure à f_o :

$$\overline{f_{f_o}}(\varphi) = \begin{cases} \overline{L_{min}}(\varphi) & \text{si } \overline{f_{freq}}(\varphi) \leq \frac{1}{f_o} \\ \infty & \text{sinon} \end{cases}$$

4.4 Conclusion

Le problème de placement est dans le cas général NP-complet. Il existe cependant des méthodes d'exploration permettant d'approximer le placement optimal en temps raisonnable. Grâce à une relaxation du programme linéaire, nous obtenons facilement une approximation de la valeur optimale des paramètres de latence et de fréquence. On peut affiner la borne en démarrant une exploration des solutions grâce à un algorithme de Branch & Bound.

Chapitre 5

Expérimentations

Dans ce chapitre, nous allons expérimenter les deux méthodes de résolution par relaxation du programme linéaire et par Branch and Bound tronqué. Nous allons à partir d'une application simple, calibrer le modèle puis, effectuer la résolution sur une véritable application de réalité virtuelle distribuée.

Enfin, nous montrerons à partir de cette application, que le problème est multi-critères en cherchant une solution optimale en fixant une contrainte sur un des paramètres.

L'ensemble des expérimentations ont été effectuées sur la plateforme Grimage[2]. Cette plateforme est composée de 11 PC bi-Xeon 2.66 Ghz et 16 PC bi-Opteron 2 Ghz. Les machines fonctionnent sous système GNU/Linux 2.6. Nos applications sont construites grâce à l'intergiciel FlowVR[1].

5.1 Vérification du modèle

Sur une application simple (figure 5.1), nous allons vérifier la pertinence du modèle et des fonctions d'évaluation.

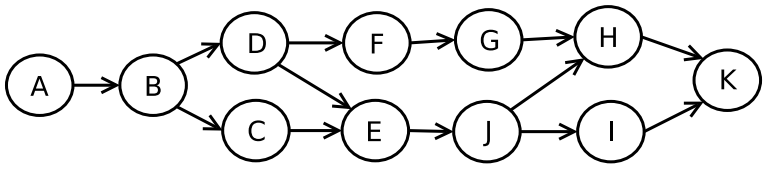

FIG. 5.1 – Application de test

5.1.1 Evaluation de la vitesse des processeurs

Nous considérons que la vitesse d'un processeur est de l'ordre de la fréquence de son horloge. Par conséquent, pour chaque processeur, nous prenons pour vitesse la fréquence

de l'horloge donnée par le système d'exploitation.

Nous avons donc $\nu_{Xeon} = 2000op/s$ et $\nu_{Opteron} = 2666op/s$.

5.1.2 Evaluation du coût des modules

A partir de la latence de chaque module mesurée grâce à un module FlowVR que nous avons écrit. Nous pouvons en déduire le coût de chaque module sur les différentes architectures. (tableau 5.1). Il est à noter que la mesure des modules doit se faire sur chaque type d'architecture. De plus, nous sommes obligés pour mesurer les modules de les faire itérer pendant une minute. Cela permet d'avoir assez de mesures pour calculer avec précision le coût moyen du module.

Par conséquent, une application d'une centaine de modules s'exécutant sur trois architectures différentes nécessiterait près de 5 heures de mesures pour calibrer l'application.

Module	a	b	c	d	e	f	g	h	i	j	k
bi-Xeons	141	30.1	50.6	69	101	117	61.3	99.2	24.2	192	147
bi-Opterons	121	30.3	50.2	66.9	98.4	101	60.1	96.6	24.1	187	141

TAB. 5.1 – Mesure des modules de l'application de test en Mega opérations CPU

Les valeurs mesurées sur les deux architectures sont très proches. Pour la suite des expérimentations, nous pouvons donc nous affranchir de l'hypothèse de processeurs hétérogènes. Nous considérons un coût unique pour chaque module qui vaut la moyenne des coûts sur les deux architectures : $\mu^j = \frac{\mu^j_{Opteron} + \mu^j_{Xeon}}{2}$.

5.2 Fréquence

Dans un premier temps, nous ignorons les communications. Les modules échanges des messages de quelques octets. On considère ce coût de communication négligeable.

Nous remarquons que les valeurs données par la fonction d'évaluation de la fréquence sont proches des valeurs mesurées sur la plateforme (tableau 5.2).

5.2.1 Latence

Le tableau 5.3 présente pour la latence une comparaison entre la latence mesurée sur l'application et les valeurs rendues par les fonctions d'évaluation.

On remarque que l'estimation de L_{max} est très pessimiste (voir le chapitre 2.3.2 à la page 18). Cette évaluation ne prend pas en compte les dépendances entre les modules et considère que tous les modules s'exécutent simultanément sur la même machine. Pour rendre l'évaluation plus précise, il suffirait de prendre en compte les dépendances.

Nombres de Opterons Xeons	Placements sur les Opterons Xeons	Fréquence mesurée(Hz)	$f_{Freq}(\varphi)$ (Hz)	Ecart (%)
0	$\emptyset$	2.8	2.6	7.1
1	{a,b,c,d,e,f,g,h,i,j,k}			
1	{a,b,c,d,e,f,g,h,i,j,k}	2.2	2.0	9.1
0	$\emptyset$			
2	{a,c,d,e,f}{b,g,h,i,j,k}	3.9	3.7	5.1
0	$\emptyset$			
0	$\emptyset$	4.7	4.9	4.3
2	{a,c,d,e,f}{b,g,h,i,j,k}			
1	{a,c,d,e,f}	4.1	4.3	4.9
1	{b,g,h,i,j,k}			
3	{a,b}{c,e}{d,f}	8.4	8.8	4.8
2	{b,g,h,i,j,k}			
2	{j,i}{g,h,k}	8.1	6.6	17
3	{a,b}{c,e}{d,f}			
2	{a,b}{j}	11	10.5	4.54
6	{f,g}{i,k}{c}{d}{e}{h}			
4	{a}{b}{c,d}{e}	11.2	10.6	5.35
4	{j,g}{i,k}{f}{h}			
6	{f,g}{i,k}{c}{d}{e}{h}	10.8	11.7	8.33
2	{a,b}{j}			
11	{a}{b}{c}{d}{e}{f}{g}{h}{i}{j}{k}	11	10.5	4.54
0	$\emptyset$			
5	{a}{b}{c}{d}{e}{f}	14.2	14.0	1.4
0	{g}{h}{i}{j}{k}			

TAB. 5.2 – Comparaison entre la fréquence mesurée sur Grimage et la valeur de la fonction d'évaluation. Chaque double ligne représente un placement différent, les modules placés sur les Opterons sont indiqués sur la sous-ligne du haut, ceux placés sur les Xéons sont indiqués sur la sous-ligne du bas

5.3 Le réseau

Dans cette partie, nous étudions l'effet des communications sur l'évaluation la latence de l'application.

Nous pouvons fixer la taille des messages envoyés entre les modules. Nous allons prendre une application simple avec deux modules s'échangeant un message de taille N (figure 5.2). On place les deux modules sur des bi-xeon différents. Grâce à un module que nous avons écrit, nous pouvons mesurer la latence du système. Nous pouvons donc construire à partir des mesures une courbe reliant la fréquence à la taille des messages N (figure

Nombres de Opterons Xeons	Placements sur les Opterons Xeons	Latence mesurée(ms)	L_{min} évaluée(ms)	L_{max} évaluée(ms)
0	∅	388	287	2080
1	{a,b,c,d,e,f,g,h,i,j,k}			
1	{a,b,c,d,e,f,g,h,i,j,k}	499	383	2770
0	∅			
2	{a,c,d,e,f}{b,g,h,i,j,k}	450	383	1420
0	∅			
0	∅	399	287	1060
2	{a,c,d,e,f}{b,g,h,i,j,k}			
1	{a,c,d,e,f}	401	325	1210
1	{b,g,h,i,j,k}			
3	{a,b}{c,e}{d,f}	320	329	548
2	{j,i}{g,h,k}			
2	{j,i}{g,h,k}	340	341	579
3	{a,b}{c,e}{d,f}			
2	{a,b}{j}	322	338	371
6	{f,g}{i,k}{c}{d}{e}{h}			
4	{a}{b}{c,d}{e}	355	329	386
4	{j,g}{i,k}{f}{h}			
6	{f,g}{i,k}{c}{d}{e}{h}	335	331	398
2	{a,b}{j}			
11	{a}{b}{c}{d}{e}{f}{g}{h}{i}{j}{k}	371	383	383
0	∅			
5	{a}{b}{c}{d}{e}{f}	321	329	329
6	{g}{h}{i}{j}{k}			

TAB. 5.3 – Comparaison entre la latence mesurée sur Grimage et la valeur de la fonction d'évaluation. On ne prend pas en compte l'influence du réseau.

5.3). Chaque point sur la courbe est la moyenne de 20 mesures sur la plateforme. Nous obtenons une variance de l'ordre de 3% pour chaque point.

FIG. 5.2 – Application permettant de mesurer les coûts de communications

Dans l'application décrite dans la figure 5.2, il n'y a qu'un module par machine. On sait donc d'après le modèle que $L_{min} = L_{max} = \frac{\mu^1+\mu^2}{\nu} + \frac{N}{\beta}$. Par conséquent, le coefficient

directeur de la droite que l'on obtiendra vaudra $\frac{1}{\beta}$. On peut donc obtenir β à partir du coefficient directeur.

$$\begin{aligned} \frac{1}{\beta} &= \frac{L_1 - L_0}{N_1 - N_0} \\ \Rightarrow \beta &= \frac{N_1 - N_0}{L_1 - L_0} \end{aligned}$$

On fait l'application numérique, on obtient alors que β = 931 Mb/s.

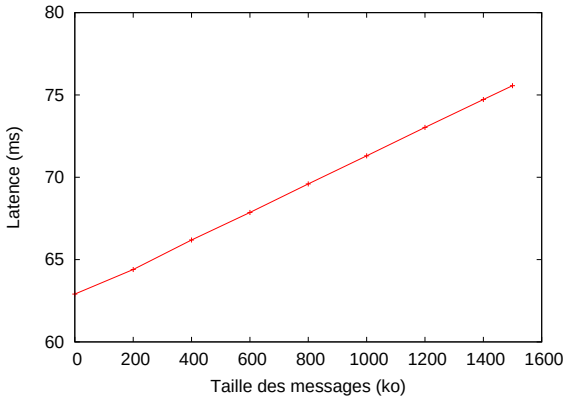

FIG. 5.3 – Latence en fonction de la taille des messages envoyés

Nous ne décrivons pas la comparaison entre la latence mesurée et la latence donnée par la fonction d'évaluation en prenant en compte les communications sur l'application (décrite à la figure 5.1). Nous nous attachons maintenant à expérimenter nos fonctions d'évaluation sur une véritable application de réalité virtuelle.

Nous obtenons donc le paramètre β nécessaire pour calibrer notre modèle. Nous pouvons maintenant évaluer la latence d'applications de réalité virtuelle. Nous allons maintenant utiliser cette calibration du modèle pour approximer le placement optimal pour une application de réalité virtuelle.

5.4 Approximation de la solution optimale pour une application

Nous avons maintenant calibré le modèle en mesurant tous les paramètres. Il est possible maintenant d'expérimenter les solutions proposées au chapitre 4. Pour résoudre le programme linéaire relaxé, nous utilisons le logiciel GNU Linear Programming Kit (GLPK). Le Branch & Bound tronqué et les fonctions d'évaluation ont été implémentés sous forme d'une librairie en C++ durant le stage. Cette librairie permet de trouver un placement optimisant un paramètre en respectant une borne sur l'autre paramètre.

5.4.1 Présentation de l'application

Nous allons comparer les différentes méthodes de résolution sur une version simplifiée d'une véritable application FlowVR. Cette application fait de la reconstruction 3D[9] et permet l'immersion de l'utilisateur dans un univers virtuel. Le graphe FlowVR simplifié de l'application est présenté sur la figure 5.4.

Les modules Acq sont reliés aux caméras et font l'acquisition d'images. Ces images sont traitées par les modules BGSub qui font de l'extraction de fond et rendent la silhouette correspondant à l'utilisateur filmé. Les modules Rec font la reconstruction 3D de la silhouette de l'utilisateur pour le rendu. Parallèlement les silhouettes sont traitées pour gérer les interactions avec le milieu virtuel. Le module Voxel permet de recréer la scène dans une grille de Voxel (équivalent d'un pixel pour une scène 3D). Les modules EDT et Rigid permettent de gérer les collisions entre les objets. Le module Carving permet à l'utilisateur de faire de la poterie en modelant un objet de l'univers virtuel. Les résultats des modules de reconstruction, Carving et Rigid sont envoyés aux modules de rendu (Render). Ce sont ces modules qui s'occupent du rendu sur l'environnement multi-projecteurs.

Dans cette application, certains modules sont contraints. Les modules Acq d'acquisition doivent être placés sur les xeons où sont branchés les caméras. Les modules Render de rendu doivent être placés sur les opterons reliés aux vidéoprojecteurs.

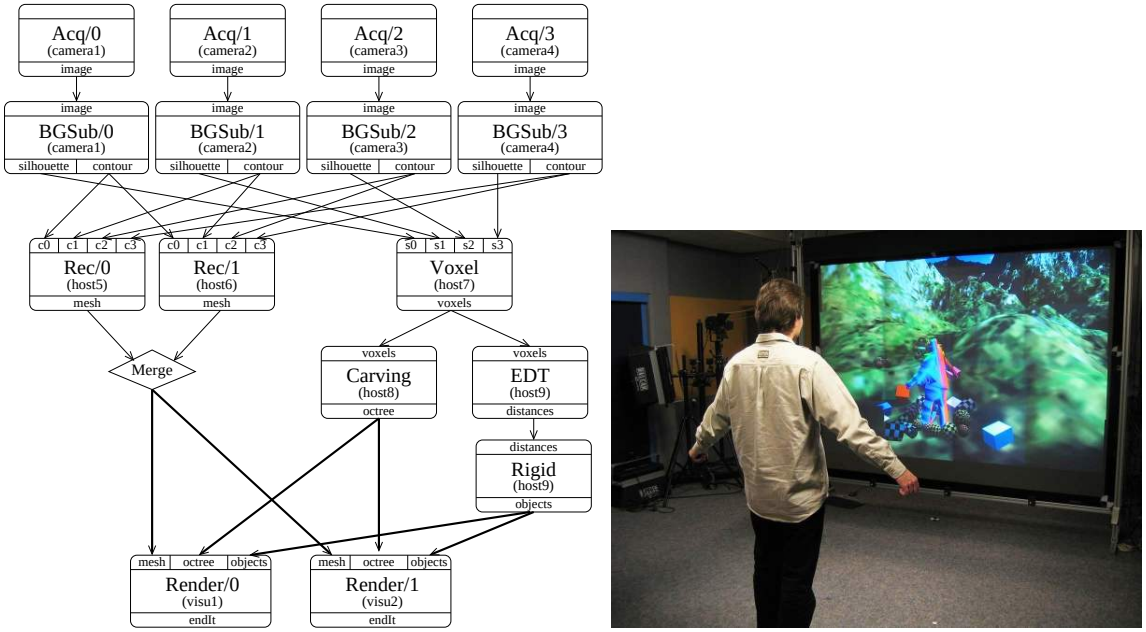

FIG. 5.4 – (a) Graphe de l'application ; (b) Exécution de l'application

5.4.2 Approximation de la fréquence

Nous allons rechercher le placement optimisant la fréquence uniquement. Nous comparons les méthodes de Branch & Bound tronqué avec la relaxation du programme linéaire.

La solution optimale est obtenue en explorant l'ensemble des solutions avec un Branch and Bound non tronqué.

Nombre de machines machines Xeons - Opterons	2 - 4	4 - 4	6 - 6
Solution optimale par Branch&Bound non tronqué	25.2	28.1	28.4
Résolution du programme linéaire relaxé avec GLPK	28.2	36.1	55
Branch&Bound tronqué à k=0	28.4	36.1	54.2
Branch&Bound tronqué à k=2	28.1	29.3	29.3
Branch&Bound tronqué à k=5	27.5	29.3	29.3
Branch&Bound tronqué à k=9	26.7	28.7	28.7

TAB. 5.4 – Comparaison des approximations avec la fréquence optimale (en Hz)

Nous remarquons que l'exploration de quelques profondeurs dans l'arbre des placements permet d'améliorer la borne obtenue par relaxation du programme linéaire (tableau 5.4).

Il suffit de placer deux modules pour que le Branch & Bound permette de donner des résultats avec un écart de l'ordre de 10% avec la solution optimale pour cette application. Cela s'explique par l'heuristique utilisée pour l'exploration. Nous commençons par placer les modules les plus coûteux. Ici les deux modules de reconstruction (Rec1 et Rec2) ont un coût bien supérieur par rapport au reste des modules. Ces modules vont donc imposer la fréquence au système. Dans les applications où il y a des modules avec un coût prédominant sur les autres modules, le choix d'une bonne heuristique permet de rendre le Branch & Bound tronqué très efficace dans la recherche d'un placement optimisant la fréquence.

5.4.3 Approximation de la latence

Nous allons pour l'application trouver le placement qui optimise la latence sans contrainte sur la fréquence. Nous allons d'abord calculer L_{min}, la borne minimal de la latence (tableau 5.5). Nous comparons le résultat de la relaxation du programme linéaire avec l'algorithme de Branch & Bound tronqué. La solution optimale est calculée en explorant l'ensemble des solutions avec le Branch & Bound.

On prend en compte dans ces résultats les communications en considérant que les modules s'échangent des messages de 1Mo avec un débit β = 931 Mb/s.

Nombre de machines machines Xeons - Opterons	2 - 4	4 - 4	6 - 6
Solution optimale par Branch&Bound non tronqué	60.7	60.7	60.7
Résolution du programme linéaire relaxé avec GLPK	60.7	60.7	60.7
Branch&Bound tronqué à k=0	60.7	60.7	60.7
Branch&Bound tronqué à k=2	60.7	60.7	60.7
Branch&Bound tronqué à k=5	60.7	60.7	60.7
Branch&Bound tronqué à k=9	60.7	60.7	60.7

TAB. 5.5 – Comparaison des approximations de L_{min} (en ms)

La définition de L_{min} permet d'expliquer les résultats obtenus. En effet, pour estimer L_{min}, la fonction d'évaluation donne le temps d'exécution du module en considérant qu'il est seul sur la machine. Par conséquent, la meilleure solution consiste à placer tous les modules sur les machines les plus rapides. Cette solution est trouvée directement grâce à la résolution du programme linéaire relaxé.

Pour calculer une borne sur L_{min}, le Branch & Bound n'est pas utile, un programme linéaire relaxé est suffisant.

Nous allons faire la même étude pour L_{max}. Le tableau 5.6 présente ces résultats. Il n'est pas possible de présenter la solution optimale. En effet, pour k = 10, l'algorithme nécessite plusieurs heures de calcul pour trouver le placement optimal. Dans ces conditions il n'est pas envisable de parcourir toutes les solutions.

Nombre de machines machines Xeons - Opterons	2 - 4	4 - 4	6 - 6
Branch&Bound tronqué à k=0	60.7	60.7	60.7
Branch&Bound tronqué à k=2	60.7	60.7	60.7
Branch&Bound tronqué à k=5	61.4	61.4	60.7
Branch&Bound tronqué à k=9	65.2	65.2	61.4

TAB. 5.6 – Comparaison des approximations de L_{max} (en ms)

Comme pour la fréquence, l'algorithme de Branch and Bound permet d'améliorer la borne sur L_{max}. Néanmoins, la méthode semble passer plus difficilement à l'échelle avec la latence. En effet, l'heuristique utilisée ici, qui place en premier les modules les plus coûteux semble moins adaptée à l'exploration des solutions pour la latence. Le Branch est Bound tronqué à k = 10 nécessite déjà une heure de calcul sur une machine récente. L'exploration totale de l'arbre des solutions est maintenant difficilement envisageable. Cependant, nous remarquons grâce aux expérimentations, que les placements optimisant la latence ont tendance à regrouper les modules sur les machines les plus rapides. Une heuristique permettant une exploration qui prend en compte ses regroupements permettrait d'explorer plus efficacement l'ensemble des solutions et augmenter le nombre de coupes dans l'arbre des solutions par le Branch and Bound.

5.4.4 Optimisation de la fréquence en fixant une contrainte sur L_{min}

Nous avons vu que l'on peut grâce au Branch & Bound tronqué optimiser la fréquence seule facilement. Comme le problème est multi-critères, les placements obtenues pour les deux problèmes d'optimisation étaient très différents. Nous allons ici, considérer le problème multi-critères en optimisant la fréquence tout en posant une contrainte sur la latence. La figure 5.5 présente la fréquence optimale en fonction d'une contrainte sur L_{min} dans le cas où l'on a 2 Xéons et 4 Opterons. Les placements obtenus ont donc une garantie sur la latence.

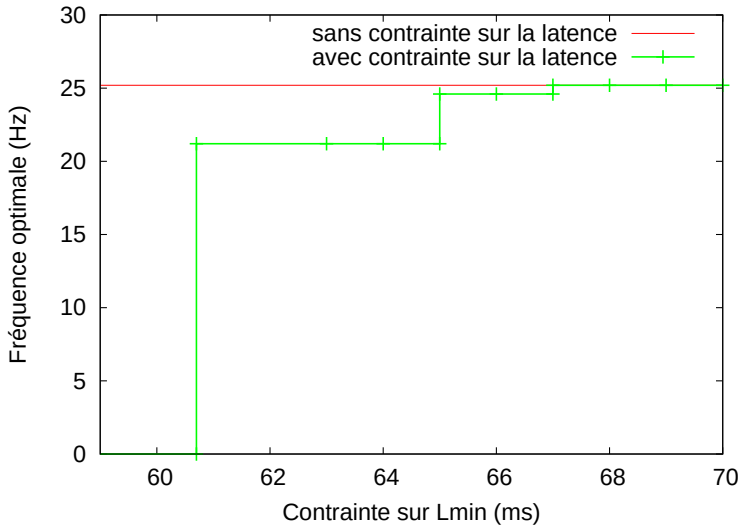

FIG. 5.5 – Relation entre la fréquence optimale (Hz) et la contrainte sur L_{min} (ms)

La courbe obtenue par expérimentation montre que pour cette application il est impossible d'obtenir une solution optimisant L_{min} et la fréquence en même temps. Les points représentés sur la courbe représentent des solutions optimales avec une contrainte sur la latence. L'utilisateur devra donc choisir parmi un de ses placements. Une solution envisageable pour choisir parmi ces solutions pourrait être de décider une zone (par exemple L_{min} < 60 ms et fréquence > 20 Hz) et choisir un placement dans cette zone.

L'application utilisée ici est une version très simple d'une application comportant plusieurs dizaines de modules. On peut penser que la même expérience sur la véritable application donnera une relation plus complexe entre les deux paramètres.

5.5 Résumé

Nous avons grâce aux expériences montré que notre modèle semblait pertinent. Nous avons réalisé une calibration du modèle en proposant une méthode mesurant les paramètres de l'application et du système. La méthode de Branch & Bound semble assez efficace pour résoudre le problème de placement qui optimise la fréquence. Pour mesurer la latence, nous avons proposé une fonction d'évaluation qui évalue un intervalle encadrant la latence. Il est possible d'améliorer l'approximation de la latence en prenant en compte les précédences entre les tâches (chapitre 2.3.1).

Nous avons testé les méthodes de résolution sur une version simplifiée d'une application de réalité virtuelle distribuée. La véritable application comporte plusieurs dizaines de modules. Par manque de temps, nous n'avons pas réussi durant ce stage, à calibrer et résoudre le problème de placement pour une grande application. Cependant, avec la version simplifiée, nous obtenons expérimentalement une relation donnant la fréquence optimale selon une contrainte sur la latence. Les résultats tendent à montrer que le problème est multi-critères dans le cas général et que nos méthodes de résolution peuvent résoudre des problèmes de placement plus complexes.

Il est maintenant possible d'envisager de nouvelles expérimentations permettant d'améliorer la librairie développée pendant le stage. On pourra envisager d'affiner l'évaluation de la latence. De plus, la calibration demande actuellement plusieurs heures de calculs, il est donc nécessaire de la simplifier pour qu'elle soit facilement utilisable. Enfin, la version de l'application présentée dans cette étude est très simple en comparaison avec la véritable application. Une suite possible de ces expérimentations pourrait être l'application de ces méthodes de résolution à une véritable application de réalité virtuelle de grande échelle.

Chapitre 6

Placement de modules parallélisables

Nous allons présenter dans ce chapitre des perspectives possibles pour améliorer un placement. Nous allons exploiter deux caractéristiques des modules d'applications de réalité virtuelle qui permettraient d'ajouter du parallélisme. Nous allons détailler des méthodes qui permettraient de tirer parti de ce parallélisme. Nous présenterons les résultats théoriques qui prouvent l'efficacité de ces méthodes.

6.1 Modules sans état

Un module *sans-état* est un module dont les sorties à l'itération k ne dépendent que des entrées à l'itération k. En d'autres termes, un module sans état n'a pas besoin d'avoir exécuté les k-1 itérations précédentes pour pouvoir effectuer l'itération k. Cette propriété permet d'exploiter un parallélisme potentiel en recouvrant plusieurs itérations. Par exemple en dupliquant deux instances d'un module sans état, il est possible de distribuer les itérations paires à la première instance et les itérations impaires à la seconde. La fréquence de l'application sera améliorée dans la mesure où les autres modules ont une fréquence supérieure au module sans état. Dans la suite nous formalisons la définition et l'utilisation de ce type de parallélisme.

On note $G_1(V_1, E_1)$, le graphe d'itération d'une application de réalité virtuelle. Les sommets E_1 représentent les modules de l'application. On note $M_1^1, \cdots M_1^N$, les N modules de E_1. On note μ^i le coût du module M_1^i. Les transitions V_1 représentent les relations de précédence entre les modules. Elles correspondent à des messages envoyés d'un module vers un autre.

On construit $G_k(V_k, E_k)$, le graphe représentant k itérations de l'application. On duplique chaque module k fois. Pour tout i et j, le coût de M_j^i la j^{eme} itération du module M^i vaut μ^i. On définit donc E_k de la manière suivante :

$$E_k = \{M_j^i \mid j \leq k \text{ et } M_1^i \in E_1\}$$

De même, on construit l'ensemble des transitions V_k en dupliquant chaque transition de V_1 et en ajoutant une transition d'une itération vers la suivante pour chaque module. Les transitions $M_k^i \rightarrow M_{k+1}^i$ modélise le fait que le module M^i doit effectuer l'itération k avant de pouvoir commencer l'itération k + 1.

$$V_k = \{M_j^i \rightarrow M_j^{i'} \mid j \leq k \text{ et } M_1^i \rightarrow M_1^{i'} \in V_1\} \cup \{M_j^i \rightarrow M_{j+1}^i \mid \forall i \text{ et } j < k\}$$

Les figures 6.1(a, b) présentent un exemple de construction de G_1 et G_2 pour une application.

Un module sans-état M^i est un module tel que pour tout k, il est possible de remplacer les transitions $M_j^i \rightarrow M_{j+1}^i$ de l'ensemble V_k par des transitions du type $M_j^i \rightarrow M_{j+d}^i$, où d est le nombre d'instances du module M^i. Dans la pratique, on supprime ces transitions en dupliquant le module et les connexions entrantes et sortantes puis en le plaçant sur une nouvelle machine. La figure 6.1(c) présente une application avec le module sans-état.

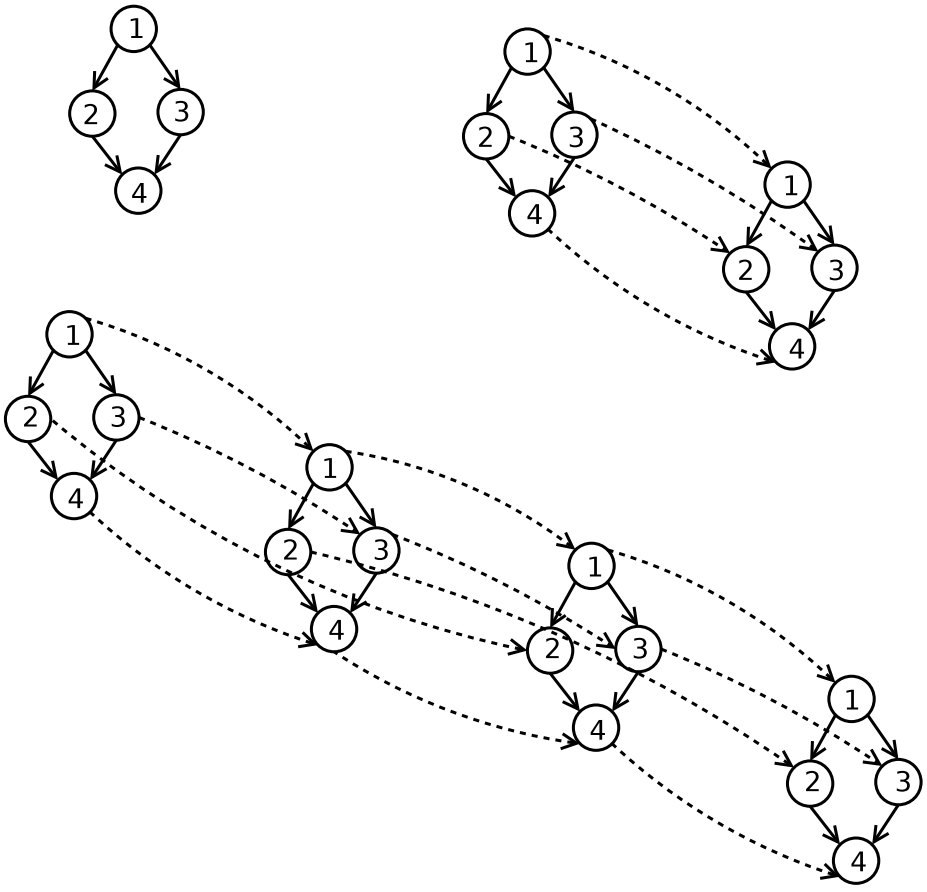

FIG. 6.1 – (a) DAG d'une itération ; (b) DAG de deux itérations, les transitions reliant une itération à la suivante sont en pointillé ; (c) DAG de quatre itérations en utilisant deux instances du module 2 sans état. Le module 2 est dupliqué, il y a donc des transitions pour le module 2 de l'itération k vers l'itération k + 2

Considérons des machines identiques de vitesse v, le plus coûteux des modules M^{max} impose la fréquence de l'application. Ce module est seul sur une machine. On a $f = \frac{1}{t_{max}}$ où t_{max} est le temps d'itération moyen de M^{max}. Nous allons alors montrer que la duplication du module M^{max} sur une nouvelle machine permet d'augmenter la fréquence.

f est la fréquence de l'application. On note pour tout k, f_k la fréquence d'exécution de k itérations. f_k est la fréquence d'itération d'une application représentée par le graphe G_k. La fréquence f vaut $\lim_{k\to\infty} k.f_k$.

S'il n'y a pas de duplication de modules, on a $f = f_1 = k.f_k$.

On pose $t_{i,1}$, le temps moyen d'itération de la machine i pour G_1. Si les modules ne sont pas dupliqués alors $t_{i,k}$ qui est le temps moyen pour k itérations sur la machine i vaut $k.t_{i,1}$ car les modules vont effectuer l'ensemble des k itérations.

On suppose maintenant que M^{max} est dupliqué. Il existe maintenant deux instances de ce module, l'un effectura les itérations paires et l'autre les itérations impaires. Au plus, un de ces modules effectura $\lceil \frac{k}{2} \rceil$ itérations. Par conséquent, $t_{max,k}$, le temps d'itération moyen pour ces modules est $\lceil \frac{k}{2} \rceil . t_{max,1}$

Calculons la valeur de f_k :

$$\frac{1}{f_k} = max(t_{i,k}) = max(k.t_i, \lceil \frac{k}{2} \rceil . t_{max})$$

Si k est pair alors $\frac{\lceil \frac{k}{2} \rceil}{k} = \frac{1}{2}$. Si k est impair alors $\frac{\lceil \frac{k}{2} \rceil}{k} = \frac{1}{2} + \frac{1}{k-1}$

Si nous posons $g(k) = 0$ si k est pair et $g(k) = \frac{1}{k-1} sinon$ alors on peut écrire que :

$$k.f_k = min(\frac{1}{t_i}, (\frac{1}{2} + g(k)).\frac{1}{t_{max}})$$

Comme $g(k) \leq \frac{1}{k}$, et que $f = \lim_{k\to\infty} k.f_k$, Nous déduisons la valeur de f :

$$f = \min(\frac{1}{t_i}, \frac{2}{t_{max}}) \geq \min(\frac{1}{t_i}, \frac{1}{t_{max}})$$

La duplication du module M^{max} permet donc d'augmenter la fréquence de l'application. Dans le cas où il y a assez de ressources disponibles, cette méthode permet simplement d'augmenter la fréquence de l'application. Ce mécanisme peut être appliqué récursivement en duplicant les modules qui imposent leurs fréquence au système.

6.2 Modules moldables

Il est aussi possible de construire grâce à MPI[3] ou Athapascan[14] par exemple, des modules paralléles qui exécutent une itération en parallèle sur plusieurs machines. Ces modules sont alors appelés moldables. Ce sont des modules parallélisables dont le degré de

parallèlisme est fixé au lancement et ne varie pas durant l'exécution de l'application[21]. Dans le graphe G_1 d'une itération, cela se traduit par la division du module moldable. Le coût des nouveaux modules est alors plus faible que celui du module d'origine. On note $\mu^i(p)$, le coût du module i dont le degré de parallalélisme est p. G_1 est le graphe sans division de module. Tous chemins de G_1 passant par un module moldable peuvent donc être réduits en se servant de cette méthode de parallélisme. G'_1 est le graphe représentant la même application que G_1 tel qu'i existe M avec un module dont le degré de parallélisme est plus élevé. Par conséquent le chemin le plus long dans G'_1 est plus petit que ceux de G_1. La latence étant liée au chemin critique (c'est à dire le chemin le plus long du graphe), cette technique de parallélisation a un impact sur la latence. La figure 6.2 est un exemple d'utilisation de modules moldable.

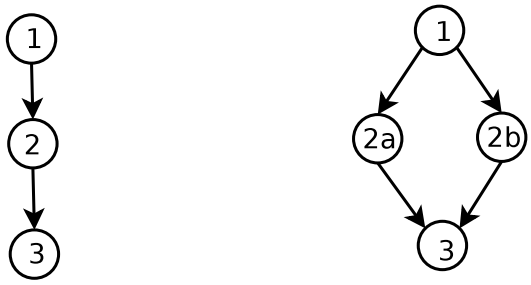

FIG. 6.2 – (a) DAG d'une itération ; (b) Division du module 2, les modules 2a et 2b ont normalement un coût plus faible que le module 2

Pour que cette technique soit efficace, il est nécessaire que $\mu^i(p) < \mu^i(1)$. Ces modules permettent donc de réaliser un placement plus complexes en prenant en compte le degré de parallélisme. Il est donc possible d'obtenir des placements avec des meilleures performances en divisant les modules moldables. Une perspective possible pour améliorer le placement est alors de décider en plus du placement le degré de parallélisme de chaque module moldable.

6.3 Conclusion

Dans un premier temps, nous avons vu comment il était possible grâce aux modules sans-état d'exécuter en parallèle plusieurs itérations du même module. Nous avons prouvé qu'en appliquant cette méthode aux modules les plus coûteux, il est possible augmenter la fréquence de l'application. Dans le cas où la quantité de ressources est suffisante, cette méthode permettrait d'augmenter la fréquence de l'application.

Une autre perspective est d'exploiter les modules parallélisables. Cette perspective permettrait d'obtenir un placement plus complexe qui déciderait du degré de parallélisme de chaque module. La difficulté est maintenant d'intégrer ces deux techniques dans la méthode de calcul du placement, notamment en définissant des heuristiques efficaces pour explorer les placement possibles.

Chapitre 7

Conclusion

Les applications de réalité virtuelle deviennent de plus en plus complexes et de plus en plus grandes et nécessitent le recours à des machines parallèles comme des grappes de PC pour leurs exécutions. Certaines peuvent être composées de plusieurs centaines de modules. Le placement des modules est un problème difficile qui a une influence sur les performances des applications. Dans le cas des applications interactives, le respect des contraintes sur la latence et la fréquence est critique pour assurer la sensation d'immersion de l'utilisateur dans un monde virtuel.

L'objectif de ce travail était de proposer des solutions pour automatiser le placement des applications interactives distribuées pour optimiser les critères de latence et de fréquence. Nous avons tout d'abord, grâce à un modèle simple, formulé les problèmes d'optimisation.

L'étude des deux problèmes d'optimisation nous a permis d'identifier un problème multi-critères. Nous avons donc recherché diverses solutions pour le résoudre et trouver une méthode donnant un placement garantissant des contraintes sur les critères de fréquence et de latence.

Dans la partie expérimentation nous avons détaillé une procédure de calibration de notre modèle. L'écriture en C++ de la méthode de Branch & Bound tronqué nous a permis de tester la méthode de résolution sur un modèle simple d'une véritable application de réalité virtuelle et montrer expérimentalement que le problème était multi-critères.

Grâce aux propriétés des modules malléables ou sans état de nos applications, nous avons présenté une amélioration de la méthode en tirant parti du parallélisme potentiel de ces modules.

Les méthodes de résolution proposées dans cette étude ont uniquement été testées sur des applications de petite taille. Il est nécessaire maintenant d'essayer de passer à l'échelle en les adaptant à des applications plus importantes. Une méthode d'exploration exacte comme le Branch and Bound nécessitera de nombreuses optimisations avant d'être utilisée sur une application de réalité virtuelle de plusieurs dizaines de modules. Il est nécessaire par exemple de tronquer la recherche au bon niveau. Nous pouvons aussi penser

à utiliser le cluster de l'environnement de réalité virtuelle pour paralléliser la recherche. Pour des grandes applications, il sera peut-être nécessaire d'utiliser des méthodes d'approximation de la solution optimale comme le recuit simulé ou les algorithmes génétiques par exemple.

Dans cette étude, nous avons considéré dans notre modèle la valeur moyenne du coût des modules d'une application. Avec cette hypothèse, nous avons obtenu des résultats satisfaisants sur les applications utilisées durant le stage. Ces valeurs peuvent néanmoins varier. En effet, il est possible de régler le niveau de détail d'une application. Par exemple, une scène détaillée finement sera plus longue à calculer et à afficher. L'utilisateur peut donc jouer sur ce paramètre pour varier le coût des modules de son application. Le niveau de détail est un paramètre ayant un impact sur les performances. Une perspective interressante serait d'intégrer ce paramètre dans le placement des modules.

L'étape suivante consistera maintenant à intégrer le mécanisme de placement dans la suite logicielle FlowVR. Pour que cette méthode soit facilement utilisable, il sera nécessaire d'automatiser le calculs des paramètres nécessaire à la méthode de résolution. Des tests avec d'autres applications plus grandes en fonctionnement sur la plateforme GRIMAGE sont aussi envisagés.

Bibliographie

[1] FlowVR. `http://flowvr.sf.net`.

[2] GrImage. `http://www.inrialpes.fr/grimage/`.

[3] Le standard MPI. `http://www-unix.mcs.anl.gov/mpi/`.

[4] OpenMASK. `http://www.irisa.fr/siames/OpenMASK`.

[5] Sudhir Aggarwal, Hemant Banavar, Sarit Mukherjee, and Sampath Rangarajan. Fairness in dead-reckoning based distributed multi-player games. In *NetGames '05 : Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*, pages 1–10, New York, NY, USA, 2005. ACM Press.

[6] J. Ahrens, C. Law, W. Schroeder, K. Martin, and M. Papka. A parallel approach for efficiently visualizing extremely large. Technical report, Los Alamos National Laboratory.

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

[8] J. Allard, V. Gouranton, L. Lecointre, E. Melin, and B. Raffin. Running VR Juggler with Multiple Displays on a Commodity Component Cluster. In *IEEE VR*, pages 275–276, Orlando, USA, March 2002.

[9] J. Allard, C. Ménier, E. Boyer, and B. Raffin. Running Large VR Applications on a PC Cluster : the FlowVR Experience. In *Proceedings of EGVE/IPT 05*, Denmark, October 2005.

[10] J. Allard and B. Raffin. Distributed Physical Based Simulations for Large VR Applications. In *IEEE Virtual Reality Conference*, USA, March 2006.

[11] Tracy D. Braun, Howard Jay Siegel, Noah Beck, Ladislau Bni, Muthucumaru Maheswaran, Albert I. Reuther, James P. Robertson, Mitchell D. Theys, Bin Yao, Debra A. Hensgen, and Richard F. Freund. A comparison study of static mapping heuristics for a class of meta-tasks on heterogeneous computing systems. In *Heterogeneous Computing Workshop*, pages 15–29, 1999.

[12] C. Cruz-Neira, D. J. Sandin, T. A. DeFanti, R. V. Kenyon, and J. C. Hart. The Cave Audio VIsual Experience Automatic Virtual Environement. *Communication of the ACM*, 35(6) :64–72, 1992.

[13] L. Cucu, R. Kocik, and Y. Sorel. Real-time scheduling for systems with precedence, periodicity and latency constraints. In *Proceedings of 10th Real-Time Systems Conference, RTS'02*, Paris, France, March 2002.

[14] T. Gautier, R. Revire, and J.L. Roch. ATHAPASCAN : an API for Asynchronous Parallel Programming. Technical report, INRIA.

[15] Greg Humphreys, Mike Houston, and Ren Ng. Chromium : A Stream Processing Framework for Interactive Rendering on Clusters. In *Computer Graphics Proceedings*, Annual Conference Series. ACM Press / ACM SIGGRAPH, Aug 2002. Proceedings of ACM SIGGRAPH 02.

[16] B. Leibe, D. Minnen, J. Weeks, and T. Starner. Integration of Wireless Gesture tracking, Object Tracking, and Reconstruction in the responsive workbench. In B. Schiele and G. Sagerer, editors, *ICVS 2001*, volume 2095 of *LNCS*, pages 73–92, 2001.

[17] K. Li, H. Chen, Y. Chen, D. W. Clark, P. Cook, S. Damianakis, G. Essl, A. Finkelstein, T. Funkhouser, Allison Klein, Z. Liu, E. Praun, R. Samanta, B. Shedd, J. Pal Singh, G. Tzanetakis, and Jiannan Zheng. Early Experiences and Challenges in Building and Using A Scalable Display Wall System. *IEEE Computer Graphics and Application*, 20(4) :671–680, 2000.

[18] V. M. Lo. Heuristic Algorithms for Task Assignment in Distributed Systems. *IEEE Trans. Comput.*, 37(11) :1384–1397, 1988.

[19] Bruce Lucas, Gregory D. Abram, Nancy S. Collins, David A. Epstein, Donna L. Gresh, and Kevin P. McAuliffe. An architecture for a scientific visualization system. In *VIS '92 : Proceedings of the 3rd conference on Visualization '92*, pages 107–114, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[20] D. Margery, B. Arnaldi, A. Chauffaut, S. Donikian, and T. Duval. OpenMASK : {Multi-Threaded | Modular} Animation and Simulation {Kernel | Kit } : a General Introduction. In Simon Richir, Paul Richard, and Bernard Taravel, editors, *VRIC 2002 Proceedings*, pages 101–110. ISTIA Innovation, June 2002.

[21] Lukasz Masko, Gregory Mounie, Denis Trystram, and Marek Tudruj. Moldable Task Scheduling in Dynamic SMP Clusters with Communication on the Fly. *parelec*, 00 :59–64, 2004.

[22] George L. Nemhauser and Laurence A. Wolsey. *Integer and combinatorial optimization*. Wiley-Interscience, New York, NY, USA, 1988.

[23] François Pellegrini and Jean Roman. SCOTCH : A Software Package for Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs. In *HPCN Europe*, pages 493–498, 1996.

[24] François Pellegrini. *Application de méthodes de partition à la résolution de problèmes de graphe issu du parallélisme*. PhD thesis, Université BordeauxI, January 1995.

[25] Bruno Raffin, Luciano Soares, Tao Ni, Robert Ball, Greg S. Schmidt, Mark A. Livingston, Oliver G. Staadt, and Richard May. PC Clusters for Virtual Reality. *vr*, 0 :215–222, 2006.

[26] Umakishore Ramachandran, Rishiyur S. Nikhil, James M. Rehg, Yavor Angelov, Arnab Paul, Sameer Adhikari, Kenneth M. Mackenzie, Nissim Harel, and Kathleen Knobe. Stampede : A Cluster Programming Middleware for Interactive Stream-Oriented Applications. *IEEE Transactions on Parallel and Distributed Systems*, 14(11) :1140–1154, 2003.

[27] Larry L. Smarr, Andrew A. Chien, Tom DeFanti, Jason Leigh, and Philip M. Papadopoulos. The OptIPuter. *Commun. ACM*, 46(11) :58–67, 2003.

[28] Yves Sorel. Syndex reference manual : A graph oriented methodology and its system level cad software for the optimization of distributed real-time embedded applications. `http://www.syndex.org/v6/refMan.pdf`.

[29] A. Vicard and Y. Sorel. Formalization and static optimization for parallel implementations. In *Proceedings of Workshop on Distributed and Parallel Systems, DAPSYS'98*, Budapest, Hungary, September 1998.