View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by INRIA a CCSD electronic archive server

Gestion de la réactivité des communications réseau

François Trahay

To cite this version:

François Trahay. Gestion de la réactivité des communications réseau. [Rapport de recherche] Universite Bordeaux 1. 2006, pp.31. inria-00177149

HAL Id: inria-00177149

https://hal.inria.fr/inria-00177149

Submitted on 5 Oct 2007

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

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

Mémoire de recherche
Gestion de la réactivité des communications réseau

François Trahay

Le 26 juin 2006

Table des matières

Chapitre 1

Introduction

Les besoins de puissance de calcul pour des domaines aussi divers que la modélisation, la fouille de données ou le calcul numérique ne cessent d'augmenter et nécessitent une constante amélioration du matériel. Cette amélioration étant de plus en plus difficile et coûteuse à obtenir, on a commencé à agglomérer des unités de calcul individuellement moins puissantes qu'un super calculateur mais pouvant rivaliser avec lui lorsqu'elles sont utilisées ensemble. Cette agglomération pour former des *grappes* a été rendue possible par l'amélioration des réseaux rapides dont les performances se rapprochent aujourd'hui de celles des bus mémoire utilisés dans les super calculateurs. Toutefois, ces différences de performances font que les grappes ne se programment pas de la même manière que les machines massivement parallèles et les communications tiennent une part importante de cette programmation.

Les grappes, contrairement aux super calculateurs, ne disposent pas d'une mémoire partagée mais d'une mémoire distribuée sur les différentes machines. Les applications, pour remplacer le bus mémoire, utilisent donc le réseau et celui-ci doit donc être exploité efficacement. Outre cette différence d'accès à la mémoire, l'utilisation des grappes est similaire à celle des super calculateurs : de nombreux threads sont créés sur les différents processeurs disponibles et ces threads communiquent entre eux pour faire progresser plus rapidement les calculs. Mais en pratique, le programmeur ne veux pas avoir à ce soucier d'où sont créés les threads ni de comment les faire communiquer. Il utilise donc des environnements de programmation comme MPI (Message Passing Interface) ou OpenMP qui s'occupent des communications entre les threads des processeurs. De plus, afin de ne pas avoir à réécrire l'application lorsque l'on change de grappe, il faut que le code soit portable. Pour cela les environnements de programmation se basent sur des supports d'exécution chargés d'offrir une interface homogène quel que soit le matériel utilisé : des bibliothèques de threads ou de communication.

Avec le développement des grappes, ces bibliothèques ont été étudiées et sont devenues de plus en plus performantes et faciles d'utilisation. Mais même si elles sont beaucoup étudiées, que ce soit en France (notamment par des équipes de Bordeaux, Lyon, Rennes, etc. rassemblées au sein du projet Grid'5000 [1]) ou dans le reste du monde (Argonne, Amsterdam, ...), l'exploitation des grappes reste différente de celle des super calculateurs car il faut prendre en compte les communications entre les différents nœuds.

[1] http ://www.grid5000.fr/

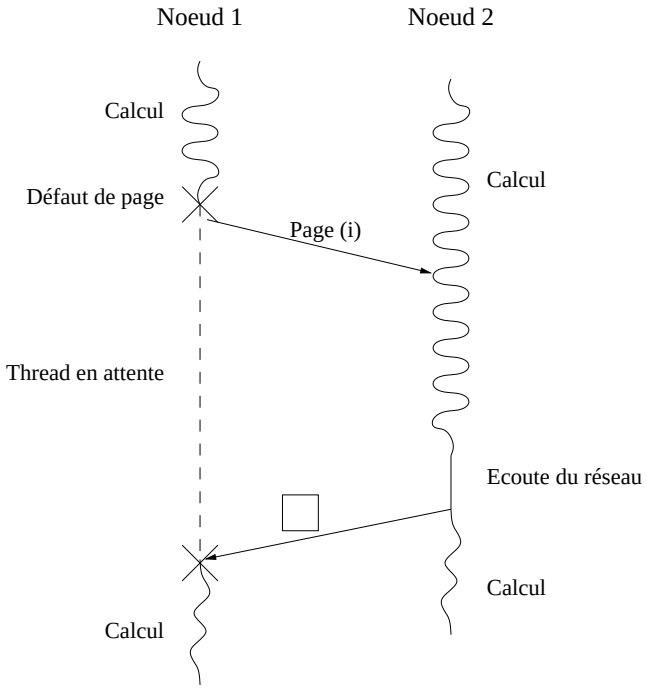


FIG. 1.1 – Influence du temps de réponse dans une DSM

1.1 Problématique

Les communications tiennent une part importante dans la programmation de grappes. En effet, si les threads communiquent entre eux par le réseau, certaines de ces communications doivent se faire rapidement. C'est notamment le cas lorsque l'on utilise une *DSM* (*Distributed Shared Memory* ou *mémoire Virtuellement partagée*, voir Fig 1.1). Dans ce modèle de programmation, la mémoire est commune à tous les threads et les pages mémoire sont réparties sur les différents nœuds de la grappe. Lorsqu'un thread veut accéder à une zone mémoire dont il ne dispose pas, il doit demander la page correspondante à un autre nœud. Il lui envoie donc une requête par le réseau et attend pour poursuivre ses calculs. Mais si le nœud propriétaire de la page est occupé à calculer, il risque de ne pas détecter la requête rapidement et donc de mettre beaucoup de temps à envoyer la page mémoire. Or pendant tout ce temps le thread qui a demandé la page ne fait rien et c'est donc de la puissance de calcul inutilisée. Un concept se dégage de cet exemple : il s'agit de la réactivité d'une application.

1.1.1 Définition de la réactivité

La réactivité d'une application est définie par sa capacité à *prendre en compte* rapidement un événement extérieur et à *lancer* le traitement approprié. La réactivité aux communications réseau peut être vue comme la latence perçue par l'application. Cette latence est différente de celle du réseau lui-même car il faut lui ajouter la durée des différents traitements de toute la pile logicielle ainsi que le temps mis par la bibliothèque de communication à détecter

l'événement. Je me suis principalement intéressé à l'optimisation de cette détection par la bibliothèque de communication.

1.1.2 Détection des événements

Pour détecter une communication réseau, il existe plusieurs méthodes suivant la technologie réseau utilisée. Les bibliothèques de communication permettent de les utiliser afin d'optimiser la latence du réseau. Mais les performances obtenues lors des tests ne tiennent pas compte de la réalité : aucun thread de calcul ne vient perturber les communications. Or, une application crée des threads de calcul et les performances qu'elle obtient pour les communications sont nettement moins intéressantes, notamment en ce qui concerne la réactivité.

1.2 Contribution

Pour résoudre ce problème de réactivité, j'ai commencé par étudier les différents schémas de communication pouvant être utilisés. J'ai décomposé complètement les communications à l'aide d'outils de *profiling*. Une fois ces schémas compris et après une étude des différentes solutions proposées, j'ai pu généraliser ces solutions et élaborer un schéma de communication «intelligent» faisant collaborer une bibliothèque de communication et un ordonnanceur afin d'obtenir des méthodes garantissant une bonne réactivité quel que soit le contexte. J'ai implémenté une partie du schéma de communication obtenu. Les tests d'évaluation montrent que l'application peut être réactive tout en laissant progresser les calculs.

Chapitre 2

Réactivité des communications

Ce chapitre a pour objectif de montrer de quelles façons les programmes destinés aux grappes de machines sont conçus afin de simplifier l'utilisation d'architectures de plus en plus complexes. Il présente ensuite les problèmes rencontrés lors de l'utilisation de ces méthodes.

2.1 Contexte

Lorsque les *grappes* sont apparues, elles étaient constituées d'ensembles de machines standard reliées par un réseau rapide. Les grappes ont ensuite intégré des machines biprocesseurs qui contiennent maintenant plusieurs cœurs par processeur. À cette multiplication du nombre de cœurs par machine s'ajoute la complexification des machines qui peuvent disposer de plusieurs bancs mémoire (*machines NUMA*). La programmation de grappe nécessite donc d'utiliser des outils pour gérer ces architectures complexes.

2.1.1 Utilisation des réseaux

Afin de faciliter la programmation de grappes de machines, des environnements de programmation ont été créés. Ceux-ci permettent de simplifier les communications entre les différents nœuds et de rendre l'utilisation du réseau transparente. On peut notamment citer MPI (Message Passing Interface) ou OpenMP. Ces environnements s'appuient sur des supports exécutifs tels que des bibliothèques de communications. Celles-ci sont chargées d'offrir une interface homogène quel que soit le réseau utilisé. Cette interface permet à un programme de pouvoir être utilisé sur n'importe quel réseau rapide sans devoir modifier de code.

2.1.2 Bibliothèques de threads

Outre les bibliothèques de communication, les environnements de programmation utilisent des bibliothèques de threads pour exploiter efficacement les architectures complexes des machines et contrôler finement l'ordonnancement des différents threads. Ce contrôle

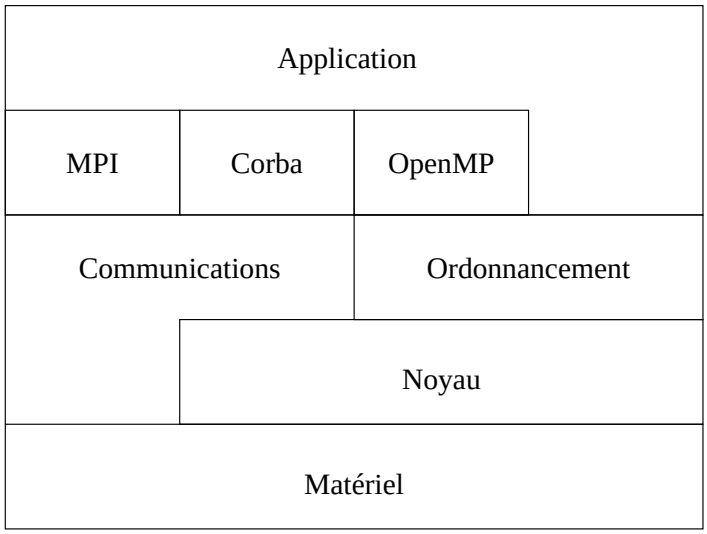


FIG. 2.1 – Pile logicielle d'une application parallèle

n'est possible qu'en utilisant des threads de niveau utilisateur par dessus les threads noyau proposés par le système d'exploitation.

Les threads noyau (aussi appelés *LWP* pour *Light Weight Processes*) permettent de profiter des architectures multiprocesseurs en exécutant plusieurs portions de code parallèlement. Ils sont relativement coûteux à utiliser, notamment lors des changements de contexte ou des synchronisations entre threads. Les applications pour les grappes créent beaucoup de threads et il faut donc éviter ces changements de contexte et ces synchronisation. Pour cela, elles se servent des threads utilisateurs proposés par les bibliothèques de threads. Ces threads permettent des synchronisations et des changements de contexte rapides. En créant plusieurs threads utilisateurs sur un thread noyau, les applications évitent les surcoûts.

Mais l'ordonnanceur de threads du noyau ne «voit» que les threads noyau et les threads créés par l'application ne sont donc pas contrôlés au niveau du noyau. Ils sont gérés par un ordonnanceur de threads propre à la bibliothèque.

2.1.3 Pile logicielle

Au final, la programmation de grappes de calcul entraîne le plus souvent l'utilisation d'une pile logicielle imposante (voir Fig. 2.1). Chaque élément de la pile étant développé indépendamment des autres, des compétitions peuvent apparaître. Ces conflits peuvent causer des problèmes de réactivité et les applications nécessitant des communications rapides risquent de ne pas fonctionner efficacement.

2.2 Problématique : la réactivité des communications

Les bibliothèques de communication utilisées pour les grappes affichent des temps de réactions très faibles (de l'ordre de la microseconde), mais quand ces bibliothèques sont utilisées dans une application de calcul, les performances obtenues sont moins bonnes. Pour s'en convaincre, nous effectuons quelques tests simples en utilisant une bibliothèque de communication et des threads de calcul.

2.2.1 La scrutation

Prenons un programme effectuant un «ping-pong» entre deux machines reliées par un réseau : une machine envoie un paquet de données à l'autre (par exemple un paquet de 4 octets) qui lui renvoie aussitôt. On mesure le temps moyen d'un aller-retour : il correspond au double de la latence.

Avec une utilisation normale de ce programme, on obtient une latence de l'ordre de quelques μs correspondant à la latence du réseau lui-même à laquelle s'ajoute un léger surcoût dû aux différents traitements dans la pile logicielle. Mais si l'on ajoute au thread qui effectue le ping-pong un thread effectuant des calculs, on voit que les quelques microsecondes se transforment en millisecondes.

Pour comprendre d'où vient le problème, regardons de plus près le fonctionnement des bibliothèques : lorsque le programme souhaite lire sur le réseau, il soumet une requête à la bibliothèque de communications. Celle-ci s'enregistre auprès de l'ordonnanceur de threads utilisateurs afin de pouvoir vérifier régulièrement si la communication est arrivée. Lors de certains événements (changements de contexte, processeur inoccupé ou au bout d'un certain temps) l'ordonnanceur lance une fonction de scrutation, garantissant ainsi la fréquence de scrutation. Lorsque le thread utilisateur est seul, l'ordonnanceur peut scruter le réseau en attendant qu'un thread soit prêt. Quand la communication arrive, elle est donc détectée rapidement et la latence relevée est alors de quelques microsecondes.

Dans le cas où un thread de calcul est présent (Fig. 2.2), lorsque la bibliothèque s'enregistre (1), l'ordonnanceur scrute le réseau. Comme la communication n'est pas encore arrivée, il donne la main à un thread utilisateur prêt sur le même thread noyau : le thread de calcul. Celui-ci calcule alors pendant *quelques millisecondes*. Pendant le calcul, la communication arrive (2), le noyau la détecte et voit que le thread noyau qui a demandé la communication est ordonnancé. Il le laisse donc travailler. Ce n'est que lorsque le thread de calcul se fait préempter par l'ordonnanceur qu'il rend la main (3). L'ordonnanceur peut alors scruter et détecte que la communication est arrivée. La main est alors redonnée au thread communiquant. Le temps de réaction s'élève donc à plusieurs millisecondes (voir Tab. 2.1).

Le mécanisme de scrutation, bien qu'offrant des temps de réponse très courts quand le processeur est inoccupé, souffre de problèmes de réactivité en présence de threads. La latence est alors multipliée par 100, ce qui peut être inquiétant lorsque les communications sont urgentes.

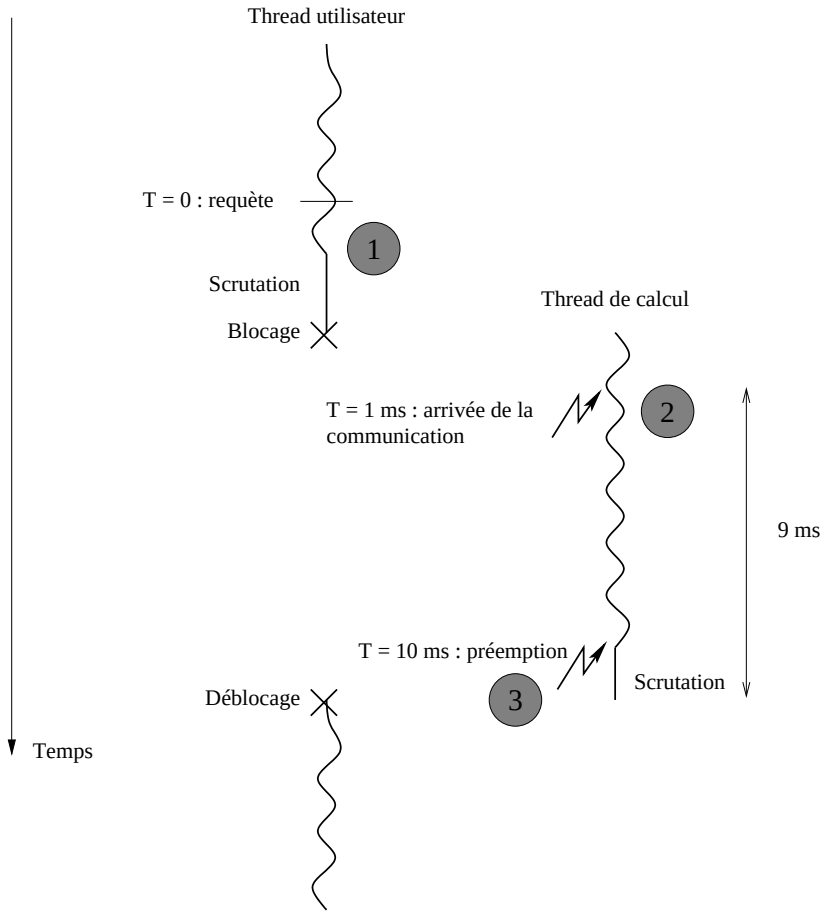


FIG. 2.2 – Déroulement d'une requête de communication avec un thread de calcul

	nombre de threads			
	0	1	2	3
Temps de réponse (ms)	0,04	5,49	5,49	5,49

TAB. 2.1 – Temps de réponse observé en fonction du nombre de threads de calcul

2.2.2 Les interruptions

Pour lire des données sur le réseau, l'application peut utiliser un autre mécanisme que la scrutation : il s'agit des interruptions. Avec cette méthode, l'application peut effectuer un appel système qui la bloquera jusqu'à l'arrivée des données. Lorsque les données arrivent, la carte réseau le signale au système qui peut débloquer l'application. Cela permet de garantir le temps de réaction de l'application.

L'inconvénient de cette méthode est que le système bloque le thread noyau complet et si des threads utilisateurs de calcul sont actifs ceux-ci ne peuvent pas progresser en attendant les données. Ceci peut être particulièrement problématique, par exemple lorsque l'on utilise une *DSM* : un thread utilisateur est chargé d'écouter le réseau et envoyer des pages mémoires. Si ce thread fait un appel système bloquant, les threads de calcul sont continuellement bloqués.

2.2.3 Scrutation sur plusieurs réseaux

Les grappes de machines utilisent souvent plusieurs réseaux : un ou plusieurs réseaux rapides (Myrinet, Quadrics, etc.) et un réseau "classique" (Ethernet). Ces différents réseaux ont des caractéristiques distinctes et la durée d'une scrutation n'est pas la même de l'un à l'autre. Lorsqu'un thread sait que des données vont arriver très prochainement sur un réseau rapide (MX/Myrinet par exemple), il demande à l'ordonnanceur de scruter. Si un autre thread attend lui aussi des données sur un autre réseau plus lent (TCP/Ethernet par exemple), l'ordonnanceur effectuera les deux scrutations : il vérifiera le premier réseau très rapidement puis le deuxième beaucoup plus lentement. La scrutation du réseau Ethernet va donc pénaliser la scrutation de Myrinet.

2.2.4 Traitement des communications

D'une manière générale, la réactivité d'une application touche toute la pile logicielle. Aux problèmes rencontrés dans les couches basses de la pile (les bibliothèques de communication et de threads) s'ajoutent ceux rencontrés dans les environnements de programmation et dans l'application elle-même. Il faudrait donc étudier cette partie de la pile logicielle afin de comprendre d'où viennent les problèmes de réactivité.

2.2.5 Des progrès à faire

Ce chapitre a montré que bien que les bibliothèques de communication affichent des latences très faibles, celles-ci sont obtenues pour des programmes non représentatifs de la réalité car ne comportant pas de threads de calcul. Il faudrait donc une méthode offrant une bonne réactivité en présence de threads de calcul. Les mécanismes proposés par les bibliothèques étant toutefois utiles lorsqu'aucun thread ne perturbe les communications, il serait intéressant d'utiliser «intelligemment» les méthodes disponibles : être capable de choisir la méthode la plus adaptée à l'état de la machine.

Chapitre 3

Vers une réactivité maîtrisée

Ce chapitre expose une proposition d'architecture logicielle capable de réagir rapidement aux communications réseau sans perturber les threads de calcul de l'application. Cette architecture est une extension du serveur d'événements asynchrones proposé par Danjean [2].

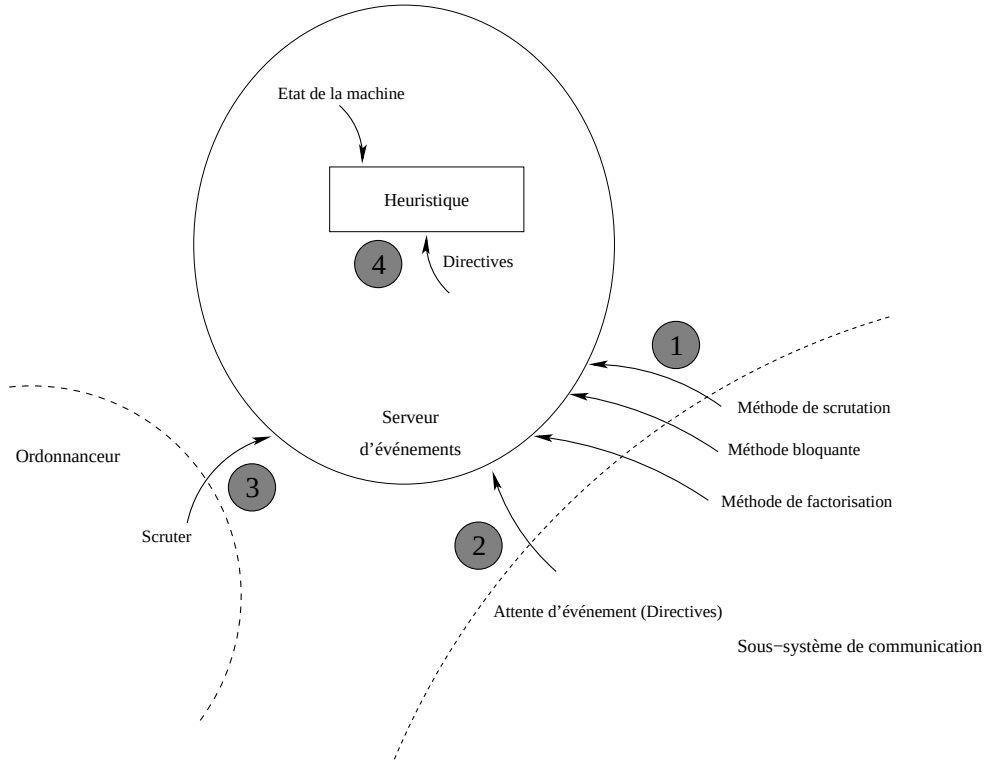
3.1 Interface du serveur d'événements

Le principe de l'architecture proposée est d'utiliser «intelligemment» les méthodes de la bibliothèque de communication. Pour cela, l'application doit renseigner le serveur d'événements et définir des préférences (voir Fig 3.1).

À l'initialisation, la bibliothèque de communication commence par définir les méthodes utilisées pour scruter le réseau, effectuer un appel bloquant et agréger des requêtes. Elle précise aussi des préférences, par exemple, elle peut demander à ce que lorsqu'elle demande une scrutation sur un réseau (MX/Myrinet par exemple), seul celui-ci soit vérifié (et pas TCP/Ethernet) afin d'éviter les problèmes de réactivité rencontrés lorsque l'on utilise plusieurs réseaux. Elle peut définir la fréquence de scrutation ou les événements pour lesquels il faut scruter le réseau.

Lorsque l'application demande une communication réseau, la bibliothèque effectue une requête auprès du serveur d'événements. Dans cette requête elle peut préciser quelle est la méthode à utiliser. Par exemple, s'il y a de fortes chances que les données demandées soient déjà arrivées, l'application peut demander au serveur d'événements d'effectuer une scrutation, et ce quel que soit l'état de la machine.

Une fois la requête transmise au serveur d'événements, celui-ci regarde l'état de la machine et choisit quelle méthode il vaut mieux utiliser. Lorsque l'ordonnanceur arrive sur un point de scrutation (processeur inoccupé, changement de contexte, etc.), il active le serveur d'événements qui en fonction des requêtes en cours, de l'état de la machine ainsi que de l'historique des requêtes peut choisir de modifier certains paramètres : méthode utilisée, fréquence de scrutation, etc.



(1) Enregistrement : L'application informe le serveur d'événements des fonctions à exécuter pour différentes tâches : comment scruter le réseau, faire un appel bloquant, agréger des requêtes. Elle enregistre aussi ses préférences : utiliser de préférence un appel bloquant, sinon scruter le réseau toutes les n μs, etc.
(2) Requête : L'application informe le serveur d'événements qu'elle souhaite communiquer (lire sur le réseau, écrire, etc.). Elle peut indiquer une préférence (ne scruter qu'un réseau particulier, utiliser la méthode de scrutation, etc.). L'application passe en état bloqué.
(3) Point de scrutation : L'ordonnanceur détecte un événement pour lequel il faut peut être scruter et en informe le serveur d'événements. Ce dernier vérifie la liste des requêtes et exécute les méthodes nécessaires. Si une requête se termine, le serveur débloque le thread correspondant et lui transmet le résultat.
(4) Choix de la méthode : Le serveur d'événements demande à la bibliothèque de communication quelle méthode utiliser. Celle-ci prend en compte les préférences ou le souhait de l'application, l'état de la machine, etc. Au final, elle décide d'utiliser la scrutation ou la méthode bloquante. Elle renseigne aussi la fréquence de scrutation ou les événements entraînant une scrutation.

FIG. 3.1 – Utilisation du serveur d'événements

3.2 Déporter les appels bloquants

Le modèle proposé par Danjean est asynchrone : les appels bloquants des threads applicatifs (read, write, etc.) sont réalisés avec des appels non bloquants, évitant ainsi de bloquer le thread noyau complet. Une scrutation est effectuée régulièrement et un thread est débloqué quand la fin de sa requête est détectée. Nous avons vu que cette méthode peut empêcher une bonne réactivité dans la plupart des cas. Je propose donc de créer une autre méthode afin de rendre l'application plus réactive lors des appels bloquants. Le principe est d'effectuer un appel bloquant sur un thread noyau spécialisé et ne comportant pas de thread de calcul. Le thread noyau peut donc être bloqué sans bloquer la progression des calculs. Cette méthode en pratique se déroule en plusieurs phases (Fig. 3.2) :

1. L'application enregistre sa requête auprès du serveur d'événements qui, en fonction de l'état de la machine ou de la requête, choisit d'exporter un appel bloquant sur un autre thread noyau.

2. Le serveur d'événements débloque ou crée un thread noyau prévu pour faire des appels bloquants et lui transmet la requête à effectuer. Le serveur d'événements bloque le thread utilisateur qui a demandé la requête et le thread de communication effectue l'appel bloquant.

3. L'ordonnanceur utilisateur regarde la liste des threads disponibles et en choisit un à qui il donne la main. Ce thread peut calculer jusqu'à ce que la requête se termine.

4. Lorsque la requête aboutit, le noyau va réveiller le thread de communication qui peut alors détecter la fin de la communication.
5. Le thread de communication débloque le thread utilisateur qui a posté la requête et se remet en attente de nouvelles requêtes.

Cette méthode permet donc d'offrir une bonne réactivité même lorsque des threads calculent. Toutefois l'exportation d'un appel bloquant a un coût et lorsque le processeur est inoccupé une scrutation sera plus efficace.

3.3 Une scrutation adaptative

La méthode consistant à exporter un appel bloquant permet des temps de réaction très courts mais induit un léger surcoût. La méthode de scrutation peut donc s'avérer parfois plus efficace (lorsqu'un processeur est libre par exemple) et il faut décider quelle méthode utiliser pour être le plus réactif. Il est donc important d'avoir une méthode décidant dynamiquement comment compléter une requête en s'appuyant sur plusieurs paramètres :

– **L'efficacité des méthodes :** les performances des méthodes diffèrent d'un réseau à l'autre. Sur certains réseau, la scrutation est extrêmement peu coûteuse (MX/Myrinet par exemple) alors que pour d'autres (TCP/Ethernet) la scrutation prend du temps. Ce

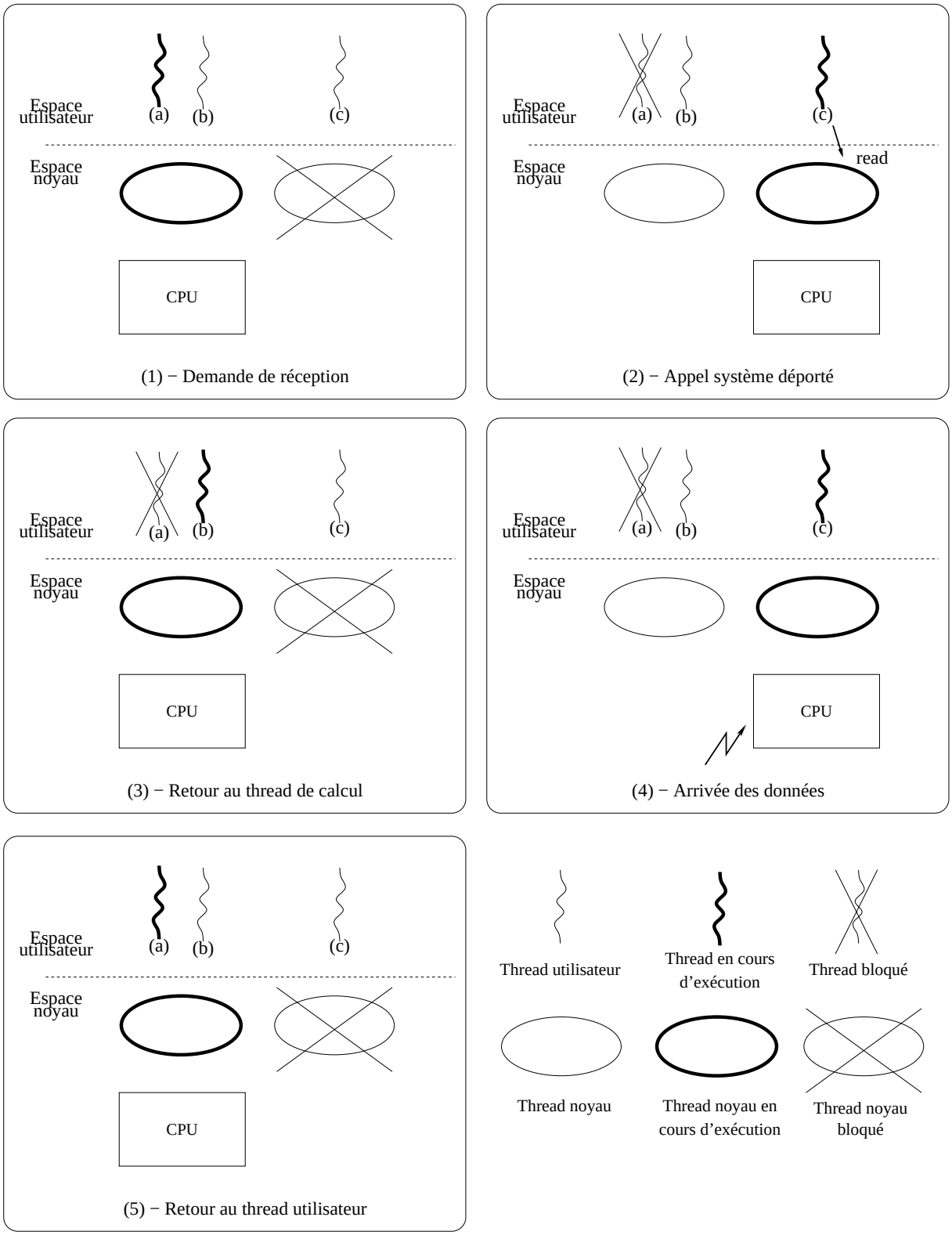


FIG. 3.2 – Déroulement d'un appel bloquant déporté

coût des méthodes doit donc être pris en compte.

- **La taille des données :** en fonction de la taille des données à envoyer, il peut être judicieux d'utiliser une méthode particulière. Par exemple, pour MX/Myrinet, si l'application veut envoyer un petit message (quelques octets), il n'est pas nécessaire de déporter l'envoi sur un autre thread noyau car celui-ci se fera rapidement. La taille à partir de laquelle il vaut mieux exporter l'envoi est à étudier en fonction du réseau utilisé.

- **Le nombre de processeurs inoccupés :** si des processeurs n'ont rien à faire, il est inutile de mettre en œuvre une déportation de l'appel bloquant alors que l'on peut scruter activement le réseau et donc compléter la requête rapidement.

- **La priorité du thread utilisateur :** un thread ayant une très forte priorité doit être très réactif et la scrutation active dans ce cas peut s'avérer utile.

- **La préférence du thread utilisateur :** bien que le serveur d'événements puisse optimiser une partie des requêtes, il ne connaît pas à l'avance l'état des communications, contrairement au programmeur qui sait quand une requête a de fortes chances d'aboutir rapidement (lorsqu'une communication est déjà arrivée). Il est donc le mieux placé pour décider quelle méthode utiliser : si il sait que la requête va aboutir directement, il n'est pas nécessaire de déporter un appel bloquant (ce qui reste coûteux). À l'inverse, si il sait que la communication n'arrivera pas tout de suite mais qu'il veut être réactif malgré les threads de calcul, il vaut mieux déporter l'appel bloquant.

- **L'historique des requêtes :** en étudiant les requêtes précédentes, on peut améliorer la fréquence de scrutation : si une communication est attendue depuis longtemps, il n'est pas forcément nécessaire de vérifier sa complétion fréquemment et on peut donc baisser sa fréquence de scrutation ou même choisir d'exporter cette requête.

Tous ces paramètres sont susceptibles de changer au cours de l'exécution du programme donc le choix doit être fait au cas par cas. Durant l'exécution d'une requête, ils risquent aussi de changer (le nombre de processeurs inoccupés au moins) et il peut donc être nécessaire de pouvoir modifier une requête : transformer une scrutation active en appel bloquant déporté. Cette solution est a étudier mais devrait être réalisable facilement.

La transformation inverse est moins intéressante car une partie du surcoût de l'exportation d'un appel bloquant survient lors de l'invocation. L'annulation ainsi que la création d'une scrutation est plus coûteux que le surcoût de la déportation.

3.4 Résumé des propositions

Ce chapitre a montré qu'il est possible d'avoir une application réactive tout en laissant progresser les threads de calcul. La méthode proposée est plus efficace que la scrutation dans la plupart des cas «réels» et le système «intelligent» permet de de déterminer la meilleure

méthode à utiliser.

Chapitre 4

Éléments d'implémentation

Ce chapitre présente l'implémentation des méthodes proposées dans la suite logicielle *PM2*. Cette suite, développée dans le projet Runtime, est constituée d'une bibliothèque de communication (*Madeleine*), d'une bibliothèque de threads (*Marcel*) ainsi que d'un environnement de programmation pour les grilles (*PadicoTM*). L'implémentation a consisté à mettre en place une méthode permettant d'effectuer des appels bloquants sans perturber les threads de calcul tout en offrant une bonne réactivité. Cette méthode est intégrée au serveur d'événements mais il reste encore à mettre en œuvre le mécanisme de décision permettant de choisir la méthode la plus adaptée à une communication. Des problèmes demandant quelques réflexions se posent lors de la communication entre le thread demandant une communication et celui qui exécute la requête : comment transmettre une requête à un autre thread noyau et, une fois la communication terminée, comment être sûr que le thread demandeur est ordonnancé immédiatement ?

4.1 Gestion des appels bloquants

Le principal problème lors de l'exportation d'un appel bloquant vient de la communication entre les différents threads, qu'ils soient de niveau utilisateur ou de niveau noyau. Par exemple, comment transmettre les requêtes du thread utilisateur au thread chargé des communications ? Pour se rendre compte des problèmes, voyons comment se comporte l'exportation d'un *select()*, une fonction de communication utilisée pour les *sockets* et qui permet de factoriser les requêtes. Il s'agit ici du cas le plus problématique puisqu'en plus de devoir lui transmettre des ordres, il faut aussi l'interrompre. Pour cela plusieurs solutions étaient à étudier.

Une première méthode est d'utiliser des sémaphores : le thread de communication se bloque (*sem_P( )*) et un thread voulant lui soumettre une requête le débloque à l'aide de la fonction *sem_V( )*. Ceci permet effectivement de débloquer rapidement le thread de communication mais dans le cas où il faut interrompre un appel bloquant, cela n'est pas possible. Les sémaphores ne sont donc pas utilisées ici mais pourraient l'être pour d'autres méthodes qui n'ont pas besoin d'être interrompues.

Une autre méthode est d'utiliser des *signaux* : le thread utilisateur envoie un *signal* au

thread noyau de communication qui est alors interrompu et peut se rendre compte qu'une nouvelle requête lui est proposée. Un problème se pose toutefois du fait de l'utilisation d'un ordonnanceur à deux niveaux : la délivrance des signaux à l'intérieur d'un thread noyau est un problème difficile et n'est pas encore implémenté Cette méthode est donc inutilisable en pratique.

Enfin, une solution plus coûteuse que les autres est d'envoyer un message au thread de communication à travers un *tube*. Ainsi, il suffit d'ajouter aux *sockets* à écouter le descripteur de fichier du *tube*. Lorsque le thread utilisateur envoie un message, le *select( )* se termine et le thread de communication se rend compte qu'une requête est en attente. Cette solution est plus coûteuse que l'utilisation de sémaphores à cause de l'écriture dans le *tube* qui nécessite un appel système à *write( )*. Toutefois ce surcoût peut être compensé par la factorisation des requêtes et c'est donc cette méthode qui est utilisée ici.

4.2 Gestion des priorités entre threads

Une fois que la communication est terminée, il reste encore à réveiller le thread utilisateur qui a fait la demande. Ceci peut être compliqué car il faut que le bon thread noyau soit ordonnancé par le système, mais aussi que le bon thread utilisateur soit choisi par l'ordonnanceur de la bibliothèque de threads.

Pour ce qui est du thread noyau, cela ne pose pas trop de problèmes car l'application n'utilisant que des threads utilisateurs, ceux-ci sont concentrés sur un thread noyau par processeur. Si l'on compte le thread de communication, on aura donc $n + 1$ threads noyau pour n processeurs. Comme le thread de communication se bloque après avoir réveillé le thread utilisateur, il ne reste pas plus que n threads noyau actifs et on est sûr que celui qui a demandé la communication est ordonnancé immédiatement.

En ce qui concerne le thread utilisateur, le problème peut être réglé en augmentant sa priorité : juste avant de demander une communication, le thread choisi une priorité maximale. Comme il se bloque tout de suite après, il ne risque pas d'empêcher les autres threads de progresser. Lorsqu'il est débloqué et que le thread noyau est ordonnancé, la bibliothèque de threads examine les threads utilisateurs disponibles. Elle remarque un thread avec une priorité très élevée et lui donne la main. Le thread demandeur de la communication peut donc continuer son traitement rapidement.

Chapitre 5

Évaluation

La méthode implémentée et présentée dans le chapitre précédent a été testée afin de comparer ses performances à celles des méthodes antérieures. Les tests ont été effectués sur des machines NUMA comportant 4 cœurs (répartis sur deux bancs mémoires) cadencés à 1,8 GHz chacun. Ces machines sont reliées par un réseau GigaEthernet (1 Gb/s). Il s'agissait ici de comparer la réactivité obtenues pour différentes méthodes de communication. Une façon simple de mesurer cette réactivité est de calculer le temps d'un «ping-pong» : deux machines s'échangent des paquets de petite taille (4 octets) et on observe le temps mis par un message pour faire un *aller-retour*. En utilisant le ping pong de cette façon, on mesure la latence d'une pile logicielle. Pour mesurer la réactivité (et donc s'approcher des conditions réelles d'exploitation des grappes), on ajoute des threads de calcul qui vont perturber les communications.

5.1 Résultats obtenus sous charge

Jusqu'à présent, la méthode utilisée dans *PM2* pour communiquer était de scruter le réseau régulièrement. Comparons les résultats que l'on obtient avec ceux obtenus pour la méthode qui exporte un appel bloquant (Tab. 5.1).

On voit que dès que des threads de calcul perturbent les communication, le temps de réaction de la méthode de scrutation sont très mauvais et l'application n'est pas réactive. En déportant les appels bloquants, le temps de réaction est constant quel que soit le nombre de threads de calcul. L'application est alors très réactive.

	nombre de threads			
	0	1	2	3
Scrutation (ms)	0,04	5,50	5,50	5,50
Appel système déporté (ms)	0,05	0,05	0,05	0,05

TAB. 5.1 – Temps de réponse observé en fonction du nombre de threads de calcul

5.2 Surcoût de l'exportation

Les résultats obtenus (Tab. 5.1) montrent des temps de réaction faibles en présence de threads de calcul. Mais on remarque aussi que lorsqu'aucun thread ne vient perturber les communications, l'exportation de l'appel bloquant provoque un léger surcoût. En analysant le déroulement de la communications, on se rend compte que ce surcoût se décompose en deux :

- Un surcoût recouvrable dû au réveil du thread de communication et à la transmission de la requête. Ce surcoût est d'environ $6\mu s$ dans le pire des cas : quand il est nécessaire d'interrompre un appel bloquant pour le relancer. Ce surcoût peut être recouvrable lorsque l'on fait une lecture sur le réseau : si les données ne sont pas encore arrivées, cette phase n'aura pas d'importance sur le temps de réaction.

- Un surcoût non recouvrable dû au déblocage du thread ayant demandé la communication et la mise en attente du thread de communication. Ce surcoût est inférieur à $9\mu s$.

5.3 Conclusion

Les chapitres précédents ont exposé les mécanismes proposés pour améliorer la réactivité d'une application. Après en avoir testé l'implémentation, on remarque qu'en présence de threads de calcul, le temps de réaction est divisé par 100 par rapport à la méthode de scrutation. Le surcoût quand il n'y a pas de threads de calcul n'est que de 20%. Les performances obtenues sont donc bonnes, mais elles le seront encore plus lorsque la méthode permettant de décider la meilleure stratégie à adopter sera implémentée.

Chapitre 6

Travaux Apparentés

Ce chapitre présente les principales suites logicielles autres que *PM2* utilisées pour les grappes et les grilles de calcul ainsi que les solutions proposées pour améliorer la réactivité des applications.

6.1 Autres suites logicielles

Avec le développement des grappes de calcul, des outils pour les exploiter se sont répandues. Des suites logicielles regroupant des outils pour les grappes et les grilles sont donc apparues. Parmi celles-ci, on peut citer Globus[1], développé par le laboratoire d'Argonne à Chicago, ou IBIS [2] de la Vrije Universiteit d'Amsterdam.

Ces suites logicielles intègrent des bibliothèques de communication performantes permettant d'obtenir des latences faibles. Mais ces bibliothèques n'étant que peu étudiées pour le *multithreading*, leur réactivité pour un programme comportant de nombreux threads de calcul n'est pas bonne.

6.2 Scheduler Activations

La réactivité des applications est généralement peu étudiée et les solutions proposées sont rares. Un mécanisme intéressant a toutefois été proposé par Anderson en 1990 : il s'agit des *Scheduler Activations* [1]. Il s'agit de modifier le noyau pour qu'il collabore avec l'ordonnanceur de niveau utilisateur. Cela permet d'utiliser des appels bloquants sans interrompre les threads utilisateurs se trouvant sur le même thread noyau.

[1] http ://www.globus.org/
[2] http ://www.cs.vu.nl/ibis/

6.2.1 Principe des upcall

Un noyau intégrant des *scheduler activations* est capable de communiquer avec l'ordonnanceur de niveau utilisateur et de l'informer des décisions d'ordonnancement prises au niveau du noyau. Ainsi lorsqu'un thread utilisateur effectue un appel système bloquant, le noyau prévient la bibliothèque de threads qui peut ne bloquer que ce thread. Les autres threads utilisateurs peuvent alors continuer à progresser. Lorsque l'appel système se termine, le noyau envoie un *upcall* à l'ordonnanceur utilisateur qui peut débloquer le thread et lui donner la main. Cette technique permet d'utiliser des appels bloquants, et donc être très réactif, tout en laissant progresser les calculs.

6.2.2 Implémentation

Les implémentations des *scheduler activations* sont rares car les modifications à apporter au noyau sont conséquentes. On peut toutefois citer FreeBSD qui intègre un mécanisme basé sur les activations depuis sa version 5.0 ainsi que le micro noyau *MACH* qui implémente lui aussi une variante appelée *Sleeping Threads* [4].

Outre sur FreeBSD, les activations ont été implantées sur quelques autres noyaux sans beaucoup de succès. Un exemple de ces implémentations est proposé par Danjean : il s'agit des *Linux Activations* [2], une version des activations pour *Linux*. Malheureusement les résultats obtenus ne sont pas à la hauteur des espérances à cause de la complexité des mécanismes mis en œuvre et la dernière version à ce jour est destinée au noyau Linux 2.4.9 publié en 2001. Bien que le concept des *scheduler activations* soit très attrayant, les difficultés pour l'implémenter ainsi que pour maintenir le code font que ce modèle n'est pas présent dans la plupart des noyaux et donc que les applications ne l'utilisent pas par soucis de portabilité.

6.3 Panda

Pour palier les difficultés d'implémentation des *Scheduler Activations* dans un noyau, *Panda* [6] propose un support d'exécution fournissant des activations. L'application peut donc utiliser des appels bloquants. Mais ces activations sont «émulées» et, bien que l'application ait l'impression d'utiliser des activations, elle utilise en fait une méthode classique de communication.

Panda propose également un début de sélection «intelligente» des méthodes à utiliser [5] : quand le processeur est inoccupé, la requête est effectuée par scrutation alors qu'un appel bloquant est utilisé quand le processeur est occupé. Ce choix peut toutefois être mal adapté car il ne permet pas de modifier la fréquence de scrutation et les priorités des threads ne sont pas prises en compte. De plus, il n'est pas possible pour l'application de forcer le système à utiliser une méthode particulière. Cela est dommage car par exemple, si l'on souhaite envoyer un petit message (quelques octets) sur un réseau rapide, cet envoi est fait immédiatement et une simple scrutation suffit. Mais si le processeur est occupé, le système effectuera un appel bloquant, beaucoup plus coûteux qu'une scrutation. Le principe de ce modèle est

intéressant mais il doit être amélioré et rendu plus «intelligent».

Chapitre 7

Conclusion

L'utilisation des grappes et des réseaux rapides s'est largement répandu ces dernières années. Ceci a été rendu possible non seulement par l'amélioration du matériel, mais aussi par le développement des bibliothèques de communication.

7.1 Contribution

L'utilisation des bibliothèques de communication dans des applications comportant des threads de calcul a fait apparaître des problèmes de réactivité : bien que les bibliothèques de communication et de threads soient performantes indépendamment, leur utilisation conjointe ne se fait pas sans accrocs.

Après une étude des schémas de communication existant et des solutions proposées, on se rend compte qu'aucune méthode simple n'existe pour faire collaborer les modules d'un support d'exécution de manière efficace. Ce document propose un modèle faisant interagir ces modules afin que les communications se fassent rapidement tout en laissant les calculs progresser. Ce modèle est partiellement implémenté et l'évaluation de cette implémentation montre que le problème de la réactivité des applications n'est pas insoluble.

7.2 Perspectives

Les résultats obtenus avec la méthode proposée annonce des perspectives intéressantes pour l'avenir. En ce qui concerne l'avenir proche, la mise en place du système choisissant quelle méthode utiliser devrait permettre une réactivité optimale quel que soit le contexte des communications. La difficulté pour ce système vient des multiples facteurs à prendre en compte pour utiliser efficacement les méthodes disponibles : coût de la scrutation, possibilité de factoriser les requêtes, importance d'une requête particulière. Une étude des divers réseaux et de leurs caractéristiques devrait permettre une méthode choisissant correctement comment utiliser le réseau.

Il serait intéressant d'évaluer une méthode alternative à l'exportation des appels bloquants : il s'agirait d'exporter les threads de calcul sur une autre thread noyau. Cela devrait

permettre d'effectuer des appels bloquants sans avoir à payer le coût du réveil du thread de communication. Cette méthode, bien que plus compliquée à mettre en œuvre, pourrait diminuer le surcoût nécessaire aux appels bloquants actuels.

À moyen terme, la gestion des nombreux threads créés par les modules de la pile logicielle pour les communications devra être améliorée pour que l'information de la fin d'une requête soit transmise rapidement à l'application. En effet, les éléments d'une pile logicielle créent des threads pour traiter les communications. Lorsqu'une requête se termine, plusieurs changements de contexte interviennent avant que l'application n'en soit informée. Cela pose des problèmes de réentrance : si plusieurs requêtes se terminent en un laps de temps très court, certaines ne seront pas traitées rapidement. On peut imaginer créer des primitives indiquant à l'ordonnanceur les portions de code à exécuter rapidement. Ainsi les différentes requêtes pourraient être traitées sans attendre et les calculs reprendre une fois les opérations urgentes terminée.

L'étude des couches basses de la pile logicielle a permis l'améliorer la réactivité à bas niveau. Cette étude doit être complétée par une compréhension du déroulement des communications dans son ensemble afin de rendre le système entier réactif aux communications.

Bibliographie

[1] Thomas E. ANDERSON, Brian N. BERSHAD, Edward D. LAZOWSKA et Henry M. LEVY : Scheduler activations : Effective kernel support for the user-level management of parallelism. Rapport technique 90-04-02, Seattle, WA (USA), 1990.

[2] Vincent DANJEAN : *Contribution à l'élaboration d'ordonnanceurs de processus légers performants et portables pour architectures multiprocesseurs*. Thèse de doctorat, Ecole Normale Supérieure de Lyon, 2004.

[3] Alexandre DENIS : *Contribution à la conception d'une plate-forme haute performance d'intégration d'exécutifs communicants pour la programmation des grilles de calcul*. Thèse de doctorat, Université de Rennes 1, IRISA, Rennes, France, December 2003.

[4] Christoph KOPPE : Sleeping threads : A kernel mechanism for support of efficient user level threads. Rapport technique, Friedrich-Alexander-University Erlangen-Nürnberg, 1995.

[5] K. LANGENDOEN, J. ROMEIN, R. BHOEDIANG et H. BAL : Integrating polling, interrupts, and thread management. *In Symposium on The frontiers of massively parallel computation*, 1996.

[6] T. UHL, H. BAL, R. BHOUDJANG, K. LANGENDOEN et G. BENSON : Experience with a portability layer for implementing parallel programming systems, 1996.