Bibliothèque de communication multi-threadée pour architectures multi-coeurs

François Trahay

RenPar'19 / SympA'13 / CFSE'7
Toulouse, France, du 9 au 11 septembre 2009

Bibliothèque de communication multi-threadée pour architectures multi-cœurs

François Trahay

LaBRI, Université Bordeaux 1
351 cours de la Libération
33405 TALENCE - France
trahay@labri.fr

Résumé
L'architecture des grappes de calcul a énormément évolué depuis quelques années. Alors qu'il y a peu la plupart des nœuds ne comportaient que quelques cœurs de calcul, les machines équipées de dizaines de cœurs deviennent monnaie courante. Cette évolution du matériel s'est accompagnée d'un changement des modèles de programmation : les approches purement MPI laissent la place à des modèles mélangeant passage de messages et multi-threading. Lors de la conception de bibliothèques de communications modernes, il faut donc prendre en compte les accès concurrents et les problèmes de scalabilité liés aux processeurs multi-cœurs. Cet article présente différentes approches pour concevoir une bibliothèque de communication adaptée aux architectures actuelles. Nous étudions l'impact sur les performances de ces méthodes et plusieurs techniques permettant d'exploiter les cœurs inutilisés sont détaillées. Les évaluations montrent que de tels mécanismes permettent de répartir la charge due aux traitements des réseaux et de recouvrir les communications par du calcul.

Mots-clés : Thread, Communication hautes performances, Recouvrement

---

1. Introduction

La tendance actuelle en matière de grappes de calcul montre une forte augmentation du nombre de cœurs par nœud de calcul. Il devient monnaie courante de disposer de 8 ou 16 cœurs par nœud et des machines possédant des dizaines ou des centaines de cœurs par nœud devraient apparaître prochainement grâce à l'évolution des processeurs. Les méthodes employées pour exploiter les grappes de calcul ont dû évoluer du fait des problèmes de scalabilité dont souffrent les approches purement MPI : le nombre croissant de processus MPI par nœud risque d'épuiser l'espace mémoire disponible ou le TLB. Les approches combinant l'utilisation de threads et de processus MPI semblent prometteuses pour outrepasser ces limitations et se développent de plus en plus [10, 12]. De tels paradigmes permettent de regrouper les ressources matérielles et de les exploiter au mieux. Toutefois, l'utilisation de threads nécessite certaines précautions dans les bibliothèques de communication afin d'éviter les risques liés aux accès concurrents.

Dans cet article, nous décrivons plusieurs solutions aux problèmes des accès concurrents dans les bibliothèques de communication. Les bénéfices apportés ainsi que les surcoûts engendrés par chaque technique sont analysés. L'utilisation efficaces des machines modernes peut être réalisée en deux phases. La première est le support des accès concurrents : une application multi-threadée doit pouvoir traiter ses communications depuis plusieurs threads simultanément. La norme MPI définit ce niveau de support par `MPI_THREAD_MULTIPLE`. La deuxième phase consiste à traiter les flots de communication en parallèle : la bibliothèque de communication doit être capable d'exploiter les éventuels cœurs libres pour faire progresser les communications. Nous décrivons plusieurs techniques exploitant les cœurs libres pour faire progresser les primitives de communication non bloquantes. L'impact de ces solutions sur les performances brutes ainsi que leurs avantages sont analysés.

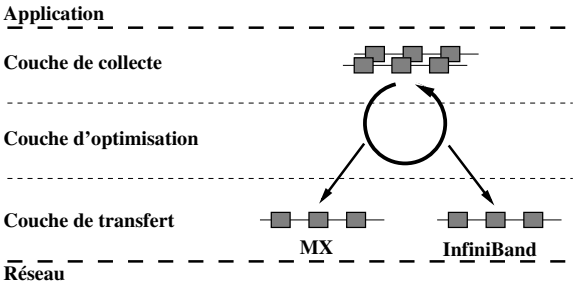

FIG. 1 – Architecture de NEWMADELEINE

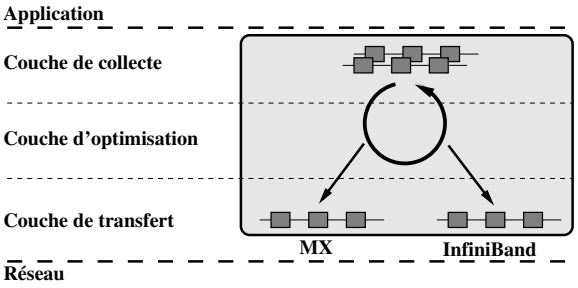

FIG. 2 – Verrou à gros grain dans NEWMADELEINE

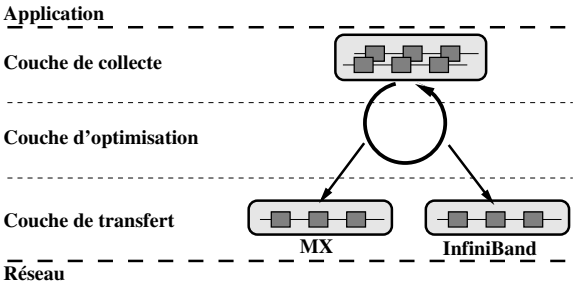

FIG. 3 – Verrous à grain fin dans NEWMADELEINE

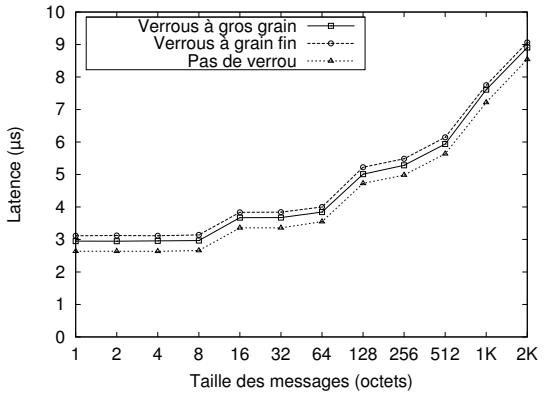

FIG. 4 – Impact des verrous sur la latence

2. Plate-forme d'expérimentation

Afin d'analyser l'impact des mesures de protection contre les accès concurrents, nous avons utilisé la suite logicielle PM2 qui est composée d'une bibliothèque de communication (NEWMADELEINE), une bibliothèque de threads (MARCEL) et d'un gestionnaire d'événements d'entrée/sortie (PIOMAN).

NEWMADELEINE [2] est notre bibliothèque de communication pour réseaux hautes performances. Elle est disponible pour la plupart des réseaux rapides contemporains : MX/Myrinet, Verbs/InfiniBand, Elan/QsNet et TCP/Ethernet. La figure 1 présente l'architecture en 3 couches de NEWMADELEINE. Contrairement à la plupart des bibliothèques de communication, l'activité de NEWMADELEINE est dirigée par les cartes réseau : les nouveaux paquets de données ne sont soumis aux cartes réseau que lorsque celles-ci deviennent inutilisées. Les messages accumulés par la couche de collecte peuvent alors être optimisés (par agrégation par exemple) par la couche d'optimisation avant d'être transmis sur le réseau. NEWMADELEINE propose une large gamme de stratégies d'optimisation : agrégation, réordonnancement de paquets, distribution sur plusieurs réseaux en parallèles, etc.

La gestion des threads est assurée par la bibliothèque de threads MARCEL [9]. L'ordonnanceur de threads à deux niveaux fourni par MARCEL permet de bénéficier des performances d'une bibliothèque de threads de niveau utilisateur tout en exploitant efficacement les machines multi-processeurs. MARCEL est aisément configurable et est donc particulièrement adapté pour aider à la progression des communications en arrière-plan. La suite logicielle PM2 utilise PIOMAN [11] comme moteur de progression des communications. PIOMAN gère la détection des événements liés aux réseaux à la place de NEWMADELEINE. En collaborant étroitement avec MARCEL, PIOMAN scrute les réseaux lorsque l'ordonnanceur atteint certains points (lorsqu'un cœur est inutilisé, lors d'un changement de contexte ou lors d'un signal d'horloge).

Les mesures relevées dans les sections suivantes ont été effectuées sur une grappe de machines équipées

de Xeon quad-core (X5460) cadencés à 3.16 GHz. Les machines comportent 4 Go de mémoire et utilisent Linux (version 2.6.26). Les nœuds sont équipés de cartes Myri-10G (avec le driver MX 1.2.7) et de cartes InfiniBand ConnectX (MT25418, utilisant le driver OFED 1.3.1). Les résultats présentés ont été obtenus en utilisant MX, mais des performances similaires sont observées avec InfiniBand.

3. Support des communications concurrentes

Dans cette section, nous étudions quelques mécanismes permettant d'implémenter un support d'accès concurrents dans une bibliothèque de communication. Les avantages et les surcoûts engendrés par ces techniques sont également évalués.

3.1. Verrous à gros grains

La technique la plus simple pour autoriser les accès concurrents à une bibliothèque de communication consiste à utiliser un mécanisme de protection à gros grain. Lorsqu'un thread doit accéder à la bibliothèque, il acquiert un *mutex* global (la figure 2 montre la portée du verrou en gris) qui n'est relâché que lorsque le thread ressort de la bibliothèque. Le *mutex* doit aussi être relâché avant d'entrer dans une section bloquante afin d'éviter les risques d'interblocage. Cette méthode permet d'assurer les accès concurrents tout en limitant le surcoût : chaque accès à la bibliothèque n'implique d'une prise de *mutex*. Comme le *mutex* n'est pris que pendant une très courte période (au plus quelques micro-secondes), l'implémentation de ce mécanisme à gros grain dans NEWMADELEINE est basée sur des *spinlocks*. Pour les sections critiques telles que celles présentes dans NEWMADELEINE, utiliser des *spinlocks* se révèle bien plus efficace qu'utiliser des *mutex* classiques : si un thread détient le *mutex* lorsqu'un autre tente d'accéder à la bibliothèque de communication, ce dernier effectuera une attente active jusqu'à ce que le verrou soit libéré. Aucun changement de contexte ne sera donc nécessaire.

Afin d'évaluer le surcoût engendré par ce mécanisme de protection, nous avons effectué un test de ping-pong. Les résultats obtenus sont présentés figure 4. L'utilisation d'un *mutex* global ("verrou à gros grain" sur la figure) entraîne un surcoût constant de 140 ns. Dans notre implémentation, le *mutex* est pris et relâché deux fois (lors de la soumission du message à la couche de collecte, et lors de la transmission des données par le réseau), chaque prise/relâchement du *mutex* coûtant 70 ns.

Le *mutex* global permet le support des accès concurrents avec un impact sur la latence limité. Toutefois, cette méthode souffre d'un manque de parallélisme : lorsqu'un thread accède à la bibliothèque, les opérations de communication (soumission de messages au réseau, scrutation, etc.) sont alors limitées aux opérations exécutées par ce thread. Le traitement des communication est donc sérialisé lorsque plusieurs threads communiquent. La figure 5 montre les résultats obtenus lorsque deux threads effectuent un ping-pong en parallèle. La latence mesurée pour chaque thread correspond au double de la latence obtenue sans thread. Cela est dû à l'exclusion mutuelle entre les deux threads qui doivent s'attendre mutuellement pour accéder à la bibliothèque de communication.

3.2. Verrous à grains fin

Afin de gérer efficacement les accès concurrents à une bibliothèque de communication, il est nécessaire de permettre à plusieurs threads de traiter les communications en parallèle. Les actions similaires doivent toujours être sérialisées (par exemple scruter un même réseau), mais il est nécessaire de pouvoir traiter les actions non corrélées en parallèle : envoyer un message sur un réseau tout en recevant des données d'une autre carte réseau doit être possible.

Des algorithmes *"lock-free"* permettent d'autoriser le traitement de telles actions en parallèles [4]. Toutefois, l'utilisation de tels algorithmes n'est pas toujours possible. D'une manière générale, il faut réduire la portée des *mutex* et identifier plus précisément les sections critiques. Dans NEWMADELEINE, les structures de données partagées sont limitées à deux types de listes :

- Les listes de paquets à optimiser situées dans la couche de collecte (une liste par pair connecté). Ces listes sont accessibles par l'application (lors de l'appel à `nm_sr_isend` par exemple) et par la couche d'optimisation. Un accès concurrent peut donc apparaître si l'application soumet un nouveau paquet alors que la couche d'optimisation en retire un (c'est à dire que NEWMADELEINE forme un nouveau paquet pour la couche de transfert).
- Les listes de paquets à envoyer situées dans la couche de transfert (une liste par carte réseau). Ces listes sont accessibles par la couche d'optimisation quand un nouveau paquet est formé et prêt à envoyer.

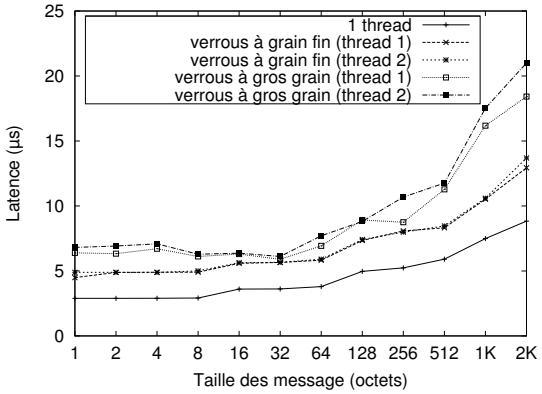

FIG. 5 – Impact des communications concurrentes sur la latence

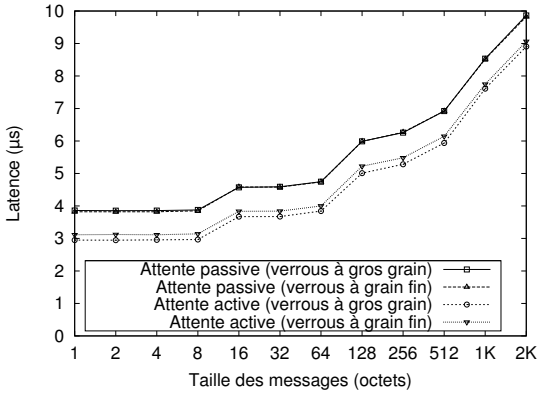

FIG. 6 – Impact des sémaphores sur la latence

NEWMADELEINE accède à une de ces listes lorsque la carte réseau correspondante devient inutilisée : une entrée de la liste est retirée et le paquet est soumis au réseau. Les accès à ces listes doivent donc avoir lieu sous exclusion mutuelle afin d'éviter les problèmes de corruption de données.

Implémenter un mécanisme de protection à grain fin se résume donc à utiliser des verrous séparés pour chacune de ces listes. Comme le montre la figure 3, les listes de chaque carte réseaux utilisent des verrous différents. Les listes de la couche de collecte n'utilisent qu'un *mutex* car la couche d'optimisation itère sur chacune de ses listes et a besoin d'une vision globale des paquets à transmettre.

Afin d'évaluer le surcoût de ce mécanisme, nous avons effectué un test de ping-pong. Les résultats sont présentés dans la figure 4. L'utilisation de verrous à grain fin implique un surcoût constant de 230 ns. Ce surcoût est plus élevé que pour le mécanisme à gros grain dans la mesure où le nombre de *mutex* sur le chemin critique est ici plus élevé.

La figure 5 montre les résultats obtenus pour le test de ping-pong concurrents présenté dans la section 3.1. Le mécanisme à grain fin se comporte mieux que celui à gros grain puisque la bibliothèque de communication est accessible par plusieurs threads simultanément. La latence mesurée est toutefois plus élevée que pour un thread. Cela s'explique par l'usage intensif qui est fait de la carte réseau ainsi que par la contention pour accéder aux *mutex*.

3.3. Attente active ou attente passive ?

Lors de la conception d'une bibliothèque de communication moderne, il est nécessaire de comprendre son comportement dans un environnement multi-threadé. L'implémentation des fonctions d'attente (par exemple MPI_Wait) est un problème important lorsque les communications et les threads sont utilisés conjointement. La plupart des bibliothèques de communication implémentent ces fonctions par une attente active : quand un thread attend la fin d'une communication, il scrute le réseau jusqu'à ce que la requête correspondant réussisse.

Bien que ce comportement soit extrêmement efficace dans un environnement non multi-threadé, il devient problématique lorsque plusieurs threads effectuent cette opération en parallèle. La scrutation du réseau est alors faite de manière concurrente et la contention risque de dégrader les performances. Un autre problème concerne ici le gaspillage des ressources processeur : un des processeurs pourrait être utilisé pour ordonnancer un thread de l'application et ainsi faire progresser le calcul.

Du point de vue de l'ordonnanceur de threads, les threads en attente devraient utiliser des primitives bloquantes (sémaphore, conditions, etc.) afin que les autres threads puissent être ordonnancés. Cela permet de réduire le temps total d'exécution des applications qui utilisent beaucoup de threads. Implémenter les fonctions d'attente avec de l'attente active réduit la puissance de calcul exploitée : sur une machine à 4 cœurs, dédier un cœur aux communications peut réduire la puissance de calcul de 25 %.

L'utilisation de primitives bloquantes pour l'implémentation de fonctions d'attente présente toutefois l'inconvénient d'entraîner des changements de contexte coûteux. Sachant que le coût d'un changement

de contexte est du même ordre de grandeur que la latence du réseau, l'utilisation de primitives bloquantes peut fortement altérer les performances des applications sensibles à la latence.
La figure 6 montre l'impact de l'implémentation des fonctions d'attente sur la latence. L'utilisation de primitives bloquantes entraîne un surcoût d'environ 750 ns dues aux changements de contexte. Il est donc important d'éviter les primitives bloquantes lorsque cela est possible.
Une solution consiste à utiliser un algorithme *fixed spin* [6] qui mélange attente active et attente passive : lorsqu'un thread s'apprête à bloquer sur une primitive bloquante, il effectue une attente active pendant une courte période (par exemple 5 µs) avant de se bloquer. Ainsi, le changement de contexte est évité si la communication se termine rapidement. Si l'événement attendu se produit plus tard, le changement de contexte se produit, mais son coût est amorti par la durée de la communication. La durée de cette période de scrutation dépend bien sûr de l'application et du coût d'un changement de contexte.
Afin d'évaluer l'impact que peuvent avoir les primitives bloquantes lorsque l'application utilise plusieurs threads pour communiquer, nous avons exécuté le test de latence multi-threadé de la suite de benchmarks OMB [7]. Ce programme effectue un test de ping-pong avec un émetteur et plusieurs threads de réception. La taille des messages échangés est de 4 octets et la latence moyenne est calculée. La figure 7 montre les résultats obtenus pour NEWMADELEINE – qui utilise à la fois attente active et attente passive – et MVAPICH2 (version 1.2p1) qui n'utilise que de l'attente active. Nous avons également utilisé OPENMPI (version 1.3.3), mais cette implémentation génère des erreurs. La latence moyenne observée pour MVAPICH2 dépend très fortement du nombre de threads de réception. Ce comportement est dû à la contention engendrée par les différents threads qui tentent d'accéder au réseau. À l'inverse, NEWMADELEINE montre des performances constantes quand le nombre de threads de réception augmente : l'utilisation de primitives bloquantes permet de limiter les accès concurrents à la bibliothèque de communication et autorise un ordonnancement des threads équitable, y compris lorsque le nombre de threads est supérieur au nombre de processeurs.

4. Traitement parallèle des communications

Dans cette section, nous présentons quelques techniques utilisées pour tirer partie efficacement des architectures multi-cœurs modernes. L'exploitation des cœurs inoccupés pour faire progresser les communications ainsi que pour effectuer des traitements coûteux en ressource processeur sont détaillées ici. Puisque les mécanismes de protection à grain fin ont un faible surcoût tout en apportant des gains de performances pour les applications utilisant des threads, seule cette approche est ici considérée.

4.1. Progression des communications

L'augmentation du nombre de cœurs par nœud peut mener à des *"trous"* dans l'ordonnancement : le nombre de threads augmentant, les synchronisations – à la fois inter-nœud et intra-nœud – se font plus fréquentes. Ces trous peuvent être exploités pour faire progresser les communications en arrière plan. Nous proposons donc d'en profiter pour améliorer la transmission de gros messages qui nécessitent un protocole de *rendez-vous*. L'utilisation des cœurs libres pour traiter les *rendez-vous* pendant que l'application calcule permet de recouvrir les communications et le calcul.
L'implémentation de la progression des communications dans NEWMADELEINE est basée sur la détection d'événements réseaux fournie par PIOMAN. Lorsque NEWMADELEINE soumet une requête au réseau, la détection de sa complétion est sous-traitée par PIOMAN : un *callback* correspondant à la fonction interrogeant le réseau est enregistré et PIOMAN prend en charge l'exécution du callback. Lorsque l'ordonnanceur de threads atteint certains point-clés (quand un processeur est inoccupé, lors d'un changement de contexte ou d'un signal d'horloge, etc.), PIOMAN est ordonnancé afin d'exécuter les callbacks des requêtes en cours. Si une communication correspondant à un *rendez-vous* est détectée, la réponse est envoyée. Les applications utilisant des primitives de communication non-bloquantes peuvent donc recouvrir le calcul et les communications de manière transparente.
Afin d'évaluer la progression des communications et le recouvrement, nous avons utilisé un programme de ping-pong alternant communications non-bloquantes et calcul. Le processus émetteur envoie les données avec une primitive non-bloquante (*i.e.* `MPI_Isend`), calcule pendant 100 µs et attend la fin de la communication (*i.e.* `MPI_Wait`). Lorsque le récepteur reçoit les données, un acquittement est renvoyé. On mesure la durée moyenne nécessaire à l'envoi des données pour une taille de messages variable.

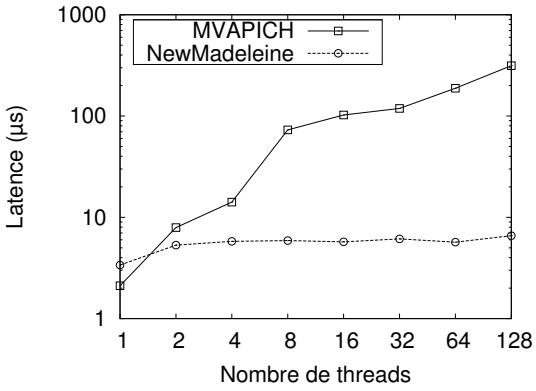

FIG. 7 – Impact des threads sur la latence

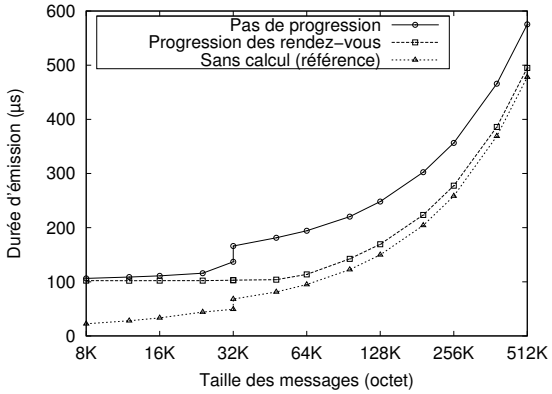

FIG. 8 – Test de progression des *rendez-vous*

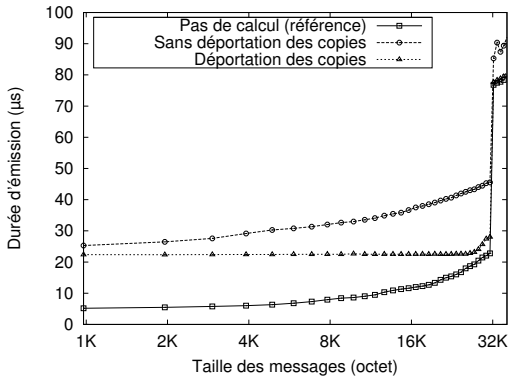

FIG. 9 – Test de recouvrement pour des messages de petite taille

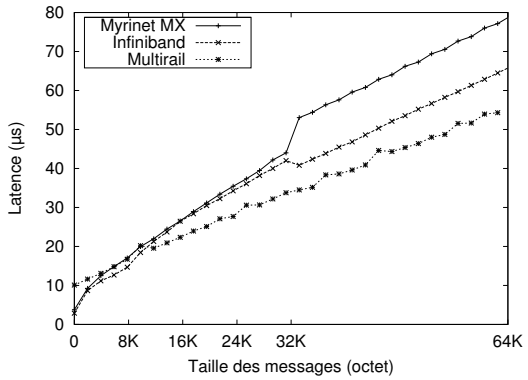

FIG. 10 – Test de latence avec InfiniBand et Myrinet MX

Les résultats obtenus pour ce programme sont présentés figure 8. On remarque que si aucune progression des communication n'est effectuée, le temps d'émission correspond à la somme du temps de calcul (100 µs) et de la communication. Le protocole de *rendez-vous* est ici en cause : lorsque l'application appelle la primitive non-bloquante (`MPI_Isend`), seule la demande de *rendez-vous* est envoyée. L'application commence ensuite son calcul et ne détecte la réponse au *rendez-vous* que lorsqu'elle atteint la primitive d'attente (`MPI_Wait`). Le transfert des données n'a donc lieu qu'après le calcul. Si la progression des communications est activée, le temps d'émission correspond à $\max(\mathtt{calcul}, \mathtt{communication})$. Lorsque le processus récepteur envoie l'acquittement du *rendez-vous*, un des cœurs libres du processus émetteur détecte la communication et peut initier le transfert de données pendant que l'application calcule. La communication et le calcul sont alors recouverts.

4.2. Équilibrage de charge

Les cœurs inutilisés peuvent également être exploités pour répartir la charge de calcul de la bibliothèque de communication. La transmission des messages de petite taille est un processus coûteux car cela nécessite une copie du message à envoyer vers la mémoire de la carte réseau. Cette copie est effectuée par le processeur – il serait trop coûteux d'utiliser un transfert DMA pour ces tailles de message – qui est donc monopolisé pendant une période relativement longue. Nous proposons donc d'exporter cette copie mémoire sur un cœur inactif afin d'équilibrer la charge et de réduire l'impact des communications sur l'application.

L'implémentation de ce mécanisme dans NEWMADELEINE se déroule comme ceci : lorsque l'application

dépose un message à envoyer, les informations nécessaires à l'envoi sont enregistrées (taille des données, destinataire, adresse du message, etc.) et l'application peut retourner au calcul rapidement. Si un cœur est inactif, ce message est détecté et la soumission des données au réseau – et donc la copie mémoire – peut avoir lieu. Si aucun cœur n'est disponible, le transfert des données a lieu lorsque l'application se bloque en attente de la fin de la communication. Ainsi le transfert de données peut être recouvert par le calcul ou, au pire, retardé.

Afin d'évaluer la déportation de la soumission de messages de petite taille, le programme décrit dans la section 4.1 a été utilisé. La durée du calcul est réduite à 20 µs afin d'observer le comportement de NEWMADELEINE quand la durée du transfert réseau devient équivalente au temps de calcul. Les résultats obtenus sont présentés figure 9. Lorsque la soumission des messages n'est pas déportée, la durée d'émission correspond à la somme de la durée de calcul et de la communication. Si un cœur inutilisé prend en charge la soumission des messages, la durée d'émission est réduite à $max(calcul, communication) + surcoût$. Le surcoût observé – environ 3 µs– est dû à la synchronisation entre les différents cœurs. Le recouvrement du calcul et des communications est donc également possible pour les messages nécessitant une copie mémoire.

4.3. Gestion du multirail

Outre le recouvrement des communications par le calcul, l'équilibrage de charge dans NEWMADELEINE peut être exploité pour réduire la durée de transmission des données. NEWMADELEINE est capable d'exploiter efficacement plusieurs cartes réseaux afin d'augmenter le débit utile : en "coupant" les messages de grandes tailles et en transférant les parties résultantes *via* plusieurs réseaux, la durée de transmission d'un message peut être fortement réduite [3]. Toutefois, l'application de cette méthode aux messages ne nécessitant pas de *rendez-vous* est délicate. Les messages de petites tailles étant soumis au réseau *via* une copie mémoire, appliquer la même méthode que pour les gros messages résulterait en une sérialisation des transferts et ne réduirait donc pas la durée de la communication.

Nous proposons d'exploiter les multiples cœurs de calcul des machines modernes afin d'outrepasser cette limitation. En utilisant le mécanisme de déportation des copies mémoire présenté dans la section 4.2, il est possible d'exploiter plusieurs réseaux pour transférer les messages de taille moyenne et donc de réduire leur durée de transmission.

L'utilisation de plusieurs cartes réseaux par NEWMADELEINE se passe comme suit : l'état de chaque réseau disponible est évalué afin de déterminer s'il est inutilisé et, s'il est occupé, dans combien de temps il deviendra disponible. L'estimation de la date de libération du réseau est basé sur un échantillonnage des réseaux à l'initialisation de l'application [3]. Les capacités des réseaux ainsi que leur charge – on considère que seul NEWMADELEINE exploite les liens réseaux – étant connus, il est possible d'estimer la durée des transferts réseau et donc la date à laquelle les cartes réseau seront à nouveau disponibles.

En se basant sur l'état des cartes, un nouveau paquet est formé et découpé de façon à maximiser l'usage du réseau. L'échantillonnage des réseaux permet de calculer un *ratio* de telle sorte que le temps de transfert du message est minimum : si plusieurs réseaux sont disponibles, le message est découpé en fonction des capacités des réseaux (un réseau lent émettra moins de données qu'un réseau rapide). Si certaines cartes réseaux sont occupées, une pénalité correspondant à leur date de libération leur est appliqué lors du calcul du *ratio*. Ainsi, une carte ayant presque fini une communication sera prise en compte lors du calcul du *ratio*.

Une fois le paquet découpé, il est soumis aux réseaux : les morceaux nécessitant un protocole de *rendez-vous* déclenchent les communications correspondantes. Les morceaux pouvant être envoyés directement sont déportés sur d'autres cœurs de calcul. Ainsi, la soumission des différents morceaux de paquet est effectuée en parallèle, limitant le surcoût de ce mécanisme.

Afin d'évaluer les performances de cette technique, nous avons effectué un test de ping-pong utilisant les réseaux InfiniBand et Myrinet simultanément. Les résultats obtenus sont reportés figure 10. Le seuil de *rendez-vous* étant fixé à 32 ko, et les performances des deux réseaux étant similaires pour les petites tailles de messages, les paquets sont ici coupés en deux partie de même taille. L'utilisation des deux réseaux simultanément permet de réduire la latence. On observe que la latence obtenue pour une taille N correspond à $max(latence(MX, N/2), latence(InfiniBand, N/2) + surcoût$. Malgré le surcoût important – dû à la déportation de la soumission des paquets – l'utilisation de plusieurs réseaux permet de réduire la latence pour les messages de taille moyenne (ici, à partir de 16 ko).

5. Conclusion

Le déploiement massif de processeurs multi-cœurs dans les grappes de calcul change radicalement la façon de programmer ces architectures. L'approche classique qui consiste à lancer un processus MPI par cœur souffre de problèmes de scalabilité et l'utilisation de threads devient obligatoire. Ce changement de paradigme ne va pas sans poser problèmes aux bibliothèques de communication qui doivent aujourd'hui gérer les accès concurrents efficacement tout en exploitant les multiples cartes réseau présentes dans les grappes modernes.
Après avoir présenté et évalué différentes stratégies permettant aux bibliothèques de communication de supporter l'usage de threads par l'application, nous avons décrit des approches pour exploiter efficacement les processeurs multi-cœurs modernes. En collaborant avec l'ordonnanceur de threads, il est possible d'assurer une progression des communications et d'utiliser les cœurs de calcul inutilisés pour équilibrer les traitements des flux de communication. Les évaluations ont montré que les mécanismes implémentés permettent de recouvrir le calcul et les communications. L'utilisation de plusieurs réseaux simultanément permet également de réduire la latence perçue par l'application.
Tous les mécanismes décrits dans cet article ont été implémentés dans la bibliothèque de communication NEWMADELEINE et dans le gestionnaire d'entrées-sorties PIOMAN. Des travaux sont en cours pour intégrer NEWMADELEINE et PIOMAN dans MPICH2 [1] ainsi que dans YAMPI [5]. Dans un futur proche, nous souhaitons élargir les mécanismes d'optimisation des messages et d'exploitation des architectures multi-cœurs afin de créer une plate-forme gérant à la fois les communications et les entrées-sorties distantes. Cette plate-forme, basée sur la bibliothèque d'entrées-sorties PGAS [8], sur NEWMADELEINE et sur PIOMAN, permettrait de fournir une brique de base pour des bibliothèques de haut niveau comme MPICH2 ou PVFS.

Bibliographie

1. ANL, MCS Division. MPICH-2 Home Page, 2007. `http://www.mcs.anl.gov/research/projects/mpich2/`.
2. O. Aumage, E. Brunet, N. Furmento, and R. Namyst. NewMadeleine : a Fast Communication Scheduling Engine for High Performance Networks. In *CAC 2007 : Workshop on Communication Architecture for Clusters, held in conjunction with IPDPS 2007*.
3. O. Aumage, E. Brunet, G. Mercier, and R. Namyst. High-performance multi-rail support with the newmadeleine communication library. In *HCW 2007 : the Sixteenth International Heterogeneity in Computing Workshop*, 2007.
4. P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur. Toward Efficient Support for Multithreaded MPI Communication. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface : 15th European PVM/MPI Users' Group Meeting, 2008*.
5. Y. Ishikawa. YAMPII Official Home Page. `http://www.il.is.s.u-tokyo.ac.jp/yampii`.
6. A.R. Karlin, K. Li, M.S. Manasse, and S. Owicki. Empirical studies of competitve spinning for a shared-memory multiprocessor. *ACM SIGOPS Operating Systems Review*, 25(5) :41–55, 1991.
7. J. Liu, B. Chandrasekaran, J. Wu, W. Jiang, S. Kini, W. Yu, D. Buntinas, P. Wyckoff, and DK Panda. Performance comparison of MPI implementations over InfiniBand, Myrinet and Quadrics. In *Supercomputing, 2003 ACM/IEEE Conference*, pages 58–58, 2003.
8. K. Ohta, H. Matsuba, and Y. Ishikawa. Gather-Arrange-Scatter : Node-Level Request Reordering for Parallel File Systems on Multi-Core Clusters. In *2008 IEEE International Conference on Cluster Computing*, pages 336–341, 2008.
9. Runtime Team, LaBRI-Inria Bordeaux — Sud-Ouest. Marcel : A POSIX-compliant thread library for hierarchical multiprocessor machines, 2007. `http://runtime.bordeaux.inria.fr/marcel/`.
10. L. Smith and M. Bull. Development of mixed mode MPI/OpenMP applications. *Scientific Programming*, 9(2) :83–98, 2001.
11. F. Trahay, A. Denis, O. Aumage, and R. Namyst. Improving Reactivity and Communication Overlap in MPI using a Generic I/O Manager. In *EuroPVM/MPI*. Springer, 2007.
12. T. Viet, T. Yoshinaga, Y. Ogawa, B.A. Abderazek, and M. Sowa. Optimization for Hybrid MPI-OpenMP Programs on a Cluster of SMP PCs. *Japan-Tunisia Workshop on Computer Systems and Information Technology*, 2004.