PIOMan : un gestionnaire d'entrées-sorties générique

François Trahay

RenPar'18 / SympA'2008 / CFSE'6
Fribourg, Suisse, du 11 au 13 février 2008

PIOMan : un gestionnaire d'entrées-sorties générique

François Trahay – `francois.trahay@labri.fr`

Université Bordeaux 1 / LaBRI
351 cours de la Libération, 33405 TALENCE - France

Résumé
Les mécanismes de communications sont, la plupart du temps, implémentés efficacement et fournissent de bonnes performances lorsque le transfert des données a lieu dans un environnement non perturbé, c'est-à-dire que les ressources (processeur, bus mémoire, cartes réseau) sont disponibles. Mais à l'heure du développement des architectures multi-cœurs et de la multiplication du nombre d'unités de calcul, ces performances sont difficiles à assurer : l'usage de threads à l'intérieur des applications entraîne une détérioration du temps de réaction et donc des performances. Dans cet article, nous présentons un gestionnaire d'entrées-sorties faisant collaborer une bibliothèque de communication et un ordonnanceur de threads afin de conserver une grande réactivité quelque soit le contexte d'exécution. Ce gestionnaire, à la fois générique et portable, permet aux communications de tirer profit du multithreading, notamment en assurant une progression des communications en arrière-plan de façon transparente pour l'application.

Mots-clés : Thread, Communication Hautes Performances, Réactivité

---

1. Introduction

L'architecture type des nœuds des grappes de calcul s'est longtemps résumée à un ou deux processeurs haut de gamme. Depuis quelques années la tendance parmi les grappes les plus puissantes au monde est d'avoir 4 ou 8 unités de calcul (comme le montre le Top500 [1]). Cette augmentation du nombre d'unités de cacul a été rendue possible par le développement des processeurs multi-cœurs. Les constructeurs de microprocesseurs produisent en effet aujourd'hui des processeurs 2-cœurs ou 4-cœurs et prévoient d'intégrer des dizaines de cœurs par puce [2]. Mais bien que l'utilisation de processeurs multi-cœurs dans les grappes de calcul soit un moyen peu onéreux d'en augmenter la puissance de calcul, cela ne va pas sans poser problème.
En effet, l'exploitation des grappes a longtemps reposé sur des interfaces de passage de messages telles que MPI du fait du faible nombre de processeurs par nœud de calcul, la plupart des implémentations MPI se concentrant principalement sur les communications inter-nœud. Chaque cœur d'un nœud exécute alors un processus MPI, rendant les communications intra-nœud peu efficaces. Avec la multiplication du nombre d'unités de calcul par nœud, de plus en plus d'implémentations MPI (MPICH2 [3] ou OpenMPI [4] par exemple) supportent l'utilisation de threads par l'application et les approches hybrides MPI+OpenMP se multiplient [5, 6].
La multiplication des threads dans des environnements auparavant centrés sur les communications inter-nœud est parfois problématique : la gestion des communications peut être difficile à assurer. Dans cet article, nous identifions plusieurs problèmes d'interaction entre threads et

communications, puis nous proposons un modèle faisant collaborer efficacement un ordonnanceur de threads et une bibliothèque de communication.

2. Impact du multithreading sur les communications

L'utilisation de threads dans des applications communicantes entraîne des problèmes de réactivité aux communications : lorsqu'une application se bloque en attendant des données venant du réseau, elle s'attend à être réveillée rapidement après l'arrivée des données. Du fait de l'introduction de threads dans l'application et de la méthode utilisée pour détecter l'arrivée des données, ce réveil peut avoir lieu tardivement. L'attente de ces données peut se faire de deux façons différentes :

Appel système bloquant – Une méthode classique consiste à utiliser un appel système bloquant, attendant que la carte réseau génère une interruption. Cela permet d'attendre l'événement sans monopoliser un des processeurs. En contrepartie, cette méthode souffre d'un surcoût dû au traitement de l'interruption. De plus, le système ne garantit pas que le thread communicant soit ordonnancé dès l'arrivée des données : s'il y a assez de threads actifs pour alimenter tous les processeurs, le thread communicant peut ne pas être ordonnancé immédiatement.

Scrutation – L'autre méthode de détection d'un événement réseau fournie par la plupart des cartes réseau consiste à scruter un emplacement mémoire. Cette méthode ne présente pas de surcoût et permet donc de détecter plus rapidement la complétion d'une communication. Malheureusement, pour assurer une bonne réactivité, la scrutation nécessite d'être effectuée fréquemment, monopolisant ainsi un des processeurs qui pourrait servir à faire progresser les calculs. De plus, le thread effectuant la scrutation doit être ordonnancé constamment pour garantir une bonne réactivité. Si le nombre de threads est plus élevé que le nombre de processeurs, cette garantie est difficile à assurer.

Outre ces problèmes de réactivité, l'introduction de threads peut paradoxalement gêner la progression de communications asynchrones en arrière-plan. La méthode classique pour assurer cette progression consiste à ajouter un thread chargé de faire progresser les communications (par exemple, en répondant aux demandes de *rendez-vous*). L'introduction de threads dans l'application peut empêcher le bon fonctionnement de ce thread de progression : lorsque la machine est surchargée (c'est-à-dire, lorsque le nombre de threads est plus élevé que le nombre de processeurs), celui-ci risque de n'être ordonnancé que rarement, ralentissant ainsi la progression des communications.
Enfin, la multiplication des threads peut entraîner une compétition pour accéder aux cartes réseau : lorsque plusieurs threads tentent de communiquer simultanément en soumettant des requêtes aux cartes réseau, l'accès à ces cartes peut prendre un certain temps, la carte étant occupée à traiter la requête d'un autre thread.

3. PIOMan : un détecteur d'événements réactif

Afin de résoudre les problèmes liés à la cohabitation des threads et des communications, nous proposons de confier la gestion de l'interaction entre threads et communications à un gestionnaire d'entrées-sorties fournissant un service de détection d'événements aux bibliothèques de communication. Ces dernières peuvent ainsi se concentrer sur les optimisations de communications sans se soucier des problèmes liés aux threads.

Ce gestionnaire d'entrées-sorties, nommé PIOMan, est implémenté au sein de la suite logicielle PM2 et interagit avec la bibliothèque de communication NewMadeleine [7] et l'ordonnanceur de threads à deux niveaux Marcel [8].

3.1. Centralisation des requêtes

En centralisant la prise en charge de l'interaction entre threads et communications, il est possible d'avoir une vision globale du système et des requêtes d'entrées-sorties. Les problèmes de compétition pour accéder aux cartes réseau sont donc clairement arbitrés. La soumission d'une requête réseau ne se fait alors que lorsque la carte est disponible et est donc effectuée en un temps miminal. Les possibilités offertes par cette centralisation des requêtes sont multiples. Lorsque la technologie réseau le permet, il peut être intéressant de grouper plusieurs requêtes afin de n'exécuter la méthode de détection qu'une fois pour toutes les requêtes, résumant ainsi la détection des événements réseau à un appel par carte réseau utilisée. Des priorités peuvent aussi être affectées aux requêtes afin de favoriser la détection de données les plus importantes pour l'application.

3.2. Travail avec l'ordonnanceur

Notre gestionnaire d'entrées-sorties interagit avec un ordonnanceur de threads à deux niveaux. Cela permet à PIOMan d'obtenir des informations utiles sur l'état de la machine (nombre de threads prêts, nombre de processeurs, etc.) L'utilisation d'un ordonnanceur de threads utilisateurs permet de contrôler finement l'ordonnancement des threads : sur chaque processeur, un thread de niveau noyau est créé et des threads utilisateurs s'exécutent sur chaque thread noyau. L'ordonnancement des threads s'effectue donc entièrement en espace utilisateur, sans aucune interaction coûteuse avec le système d'exploitation.

L'interaction avec l'ordonnanceur de threads permet au gestionnaire d'entrées-sorties de donner des indications sur l'ordonnancement, par exemple en précisant les threads de communication dont les données sont arrivées et qu'il serait judicieux d'ordonnancer. De plus, le gestionnaire d'entrées-sorties peut être appelé à certains moments clés (lorsqu'un processeur est inoccupé, lors d'un changement de contexte, lors d'un signal d'horloge, etc.) assurant ainsi un temps de réaction borné aux événements réseau.

Les informations que donnent l'ordonnanceur permettent d'adapter la fréquence de scrutation en fonction de l'activité de la machine. Ainsi, si un des processeurs est inoccupé, il peut être intéressant d'attendre activement un événement réseau. Dans le cas où tous les processeurs sont occupés, une attente active monopoliserait un processeur et empêcherait des calculs de progresser et il convient donc de réduire la fréquence de scrutation.

3.3. Adapter la méthode de détection au contexte

Les méthodes de détection fournies par les interfaces réseau ne sont chacunes optimales que dans un nombre limité de cas. Par exemple, les méthodes de scrutations seront à éviter lorsque tous les processeurs sont surchargés. Il convient donc d'adapter la méthode de détection au contexte d'exécution. Du fait de la connaissance de l'état de la machine, cette adaptation est possible : lorsqu'un des processeurs est inoccupé, utiliser une méthode de scrutation sera vraisemblablement le meilleur choix (le surcoût de ce type de méthode est minimal et une attente active ne dérange alors pas la progression des calculs). À l'inverse, lorsque tous les processeurs sont occupés, une attente active reviendrait à monopoliser un des processeurs et empêcher la progression des calculs. Une solution alternative consisterait à scruter le réseau puis redonner la main à un thread de calcul : l'impact sur les calculs serait alors négligeable. Cette solution souffre néanmoins d'un manque de réactivité : la fréquence de scrutation étant faible (la scru-

tation ne s'effectue ici que lorsque l'ordonnanceur est exécuté, c'est-à-dire à la fin de chaque quantum de temps), le temps de réaction à un événement réseau serait excessivement long (en moyenne un demi quantum de temps). Si l'on souhaite rester réactif sans empêcher les calculs de progresser, il faut utiliser un appel bloquant fourni par l'interface réseau qui soit basé sur des interruptions. Cela ne perturbe pas les calculs tout en assurant un temps de réaction satisfaisant. Toutefois, l'utilisation d'appels bloquants avec un ordonnanceur à deux niveaux est problématique. En effet, un thread effectuant un appel bloquant peut empêcher les autres threads s'exécutant sur le même thread noyau de continuer leur progression. Les appels bloquants sont donc généralement évités sur ce genre de systèmes.
Il est pourtant possible, moyennant quelques précautions, d'utiliser des appels bloquants sans gêner la progression des threads utilisateurs. Avant d'exécuter une fonction potentiellement bloquante, PIOMan réveille un thread noyau supplémentaire chargé de secourir les threads utilisateurs bloqués. Si la fonction se trouve être bloquante, les threads restants pourront donc s'exécuter sur ce thread noyau supplémentaire jusqu'à ce que la carte réseau génère une interruption et réveille le thread de communication. Ce dernier continuera alors son exécution et demandera le rapatriement des autres threads. À l'inverse, si l'appel s'effectue sans bloquer le thread, ce dernier pourra continuer son exécution normalement. Le thread noyau supplémentaire ayant une faible priorité, celui-ci ne sera pas ordonnancé dès son réveil mais après que le thread communicant soit bloqué. Cette technique a l'avantage de ne nécessiter aucune modification du système d'exploitation, contrairement aux *Scheduler Activations* [9].
Outre l'état de la machine, le gestionnaire d'entrées-sorties prend en compte les informations que peut lui donner la bibliothèque de communication. En effet, celle-ci a une plus grande connaissance du temps de complétion des requêtes. Par exemple, lorsqu'une application communique avec MX/Myrinet et utilise TCP/Ethernet comme canal de contrôle, les données de contrôle seront rares et ne nécessiteront pas une réactivité élevée. La bibliothèque de communication peut alors suggérer au gestionnaire d'entrées-sorties d'utiliser un appel bloquant pour ces requêtes, même lorsqu'un processeur est inoccupé. Ainsi, la scrutation ne concernera que les requêtes utilisant le réseau Myrinet et l'application ne sera plus perturbée par la détection d'événements sur le canal de contrôle.

3.4. Généricité du détecteur

Afin de ne pas avoir à récrire toutes les fonctions nécessaires au bon fonctionnement de chaque interface réseau supportée par la bibliothèque de communication (les méthodes d'initialisation

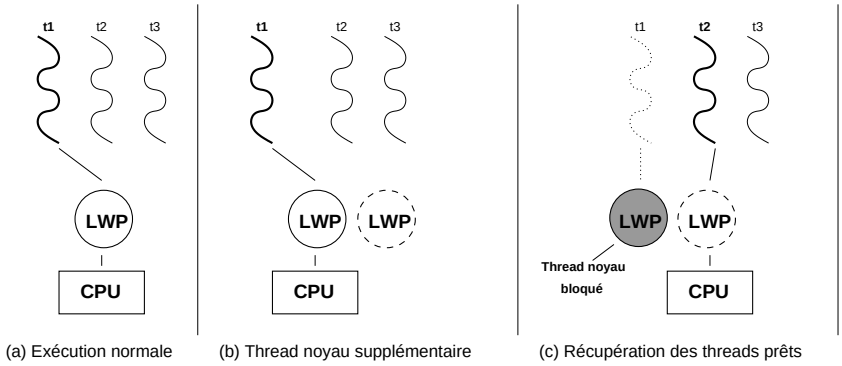

FIG. 1 – Utilisation d'un thread noyau (LWP) supplémentaire pour permettre un appel bloquant.

par exemple), le gestionnaire d'entrées-sorties utilise des << *fonctions de rappel* >> (*callbacks*) : lors de l'initialisation, les fonctions à exécuter pour chaque interface réseau (méthode de scrutation, appel bloquant, fonction permettant de regrouper plusieurs requêtes, etc.) sont fournies au gestionnaire d'entrées-sorties. Lorsque la bibliothèque soumet des requêtes au détecteur d'événements, celui-ci exécute les fonctions correspondant à l'interface réseau jusqu'à la détection de la terminaison des requêtes. Le gestionnaire d'entrées-sorties est donc générique : quand la bibliothèque de communication supporte une nouvelle technologie réseau, il suffit de fournir des << *fonctions de rappels* >> pour pouvoir détecter les événements associés à ce réseau. Cette généricité permet d'utiliser le gestionnaire d'entrées-sorties pour détecter des événements quelconques sans être limité au cadre des événements réseau. Les entrées-sorties sur disques sont bien sûr une application directe du détecteur d'événements, mais l'ordonnanceur de threads lui-même pourrait utiliser ce service pour détecter certains signaux transitant entre les processeurs d'un nœud.

4. Évaluation

L'évaluation de l'implémentation du gestionnaire d'entrées-sorties présenté porte sur trois points. Un test de réactivité mesure la latence dans un environnement threadé. Il s'agit d'un ping-pong au cours duquel une des machines exécute plusieurs threads de calcul. La progression en arrière-plan des communications asynchrones est aussi mesurée. Pour cela, le récepteur poste une réception asynchrone (`irecv`), calcule pendant 50 ms, puis attend la complétion de la requête. En mesurant le temps passé à émettre les données, on évalue la qualité de la progression de la communication. Le surcoût engendré par l'utilisation du gestionnaire d'entrées-sorties est évalué en effectuant un test de latence dans un environnement non perturbé (sans thread de calcul).

Tous ces tests ont été effectués sur la même plate-forme matérielle : une grappe de bi-Opteron dual core reliés par les réseaux Myri-10G et GigaEthernet. Les résultats présentés concernent soit MX/Myri-10G, soit TCP/Ethernet mais des résultats similaires peuvent être obtenus pour différents réseaux (Myrinet, Ethernet et Infiniband). Pour cette évaluation, PIOMan gère la détection des événements pour la bibliothèque de communication NewMadeleine [7] et utilise l'ordonnanceur de threads utilisateurs Marcel [8]. Nous évaluons ici trois versions différentes :

- Une version monoprocesseur : Marcel ne gère ici qu'un thread noyau sur lequel tous les threads s'exécutent. PIOMan ne peut alors pas utiliser d'appels bloquants (qui nécessitent un thread noyau supplémentaire).
- Une version multiprocesseur : Marcel utilise un thread noyau par processeur disponible. PIOMan peut alors utiliser les appels bloquants et les méthodes de scrutation en fonction de l'état de la machine.
- Une version de NewMadeleine sans threads n'utilisant pas PIOMan.

4.1. Évaluation de la réactivité

La réactivité fournie par PIOMan est évaluée de la façon suivante : deux machines exécutent un ping-pong en concurrence avec des threads de calcul. On mesure le temps nécessaire aux machines pour s'échanger 4 octets en fonction du nombre de threads par processeur. Les résultats obtenus pour cette expérience sont regroupés dans la Figure 2.

Ces résultats montrent que le temps de réaction de NewMadeleine augmente quand on ajoute des threads de calcul. Le thread communicant doit en effet être ordonnancé pour détecter l'arrivée de données, ce qui arrive plus rarement quand le nombre de threads augmente. En utilisant la version de PIOMan basée sur la scrutation, on observe un temps de réaction relativement

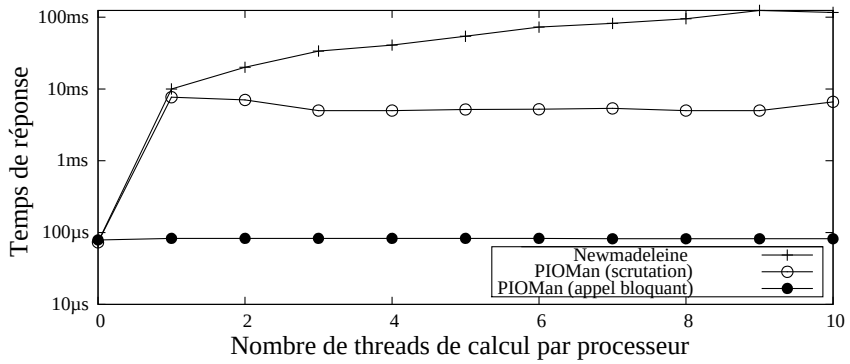

FIG. 2 – Temps de réponse observés en fonction du nombre de threads de calcul par processeur pour TCP/Ethernet.

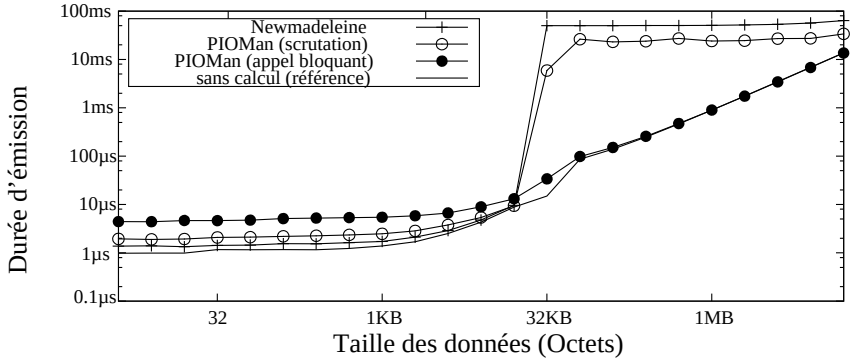

FIG. 3 – Temps d'émission mesurés pour MX/Myrinet.

constant correspondant à la moitié d'un quantum de temps. PIOMan ne peut effectuer la scrutation que lorsque l'ordonnanceur reprend la main, c'est-à-dire à la fin de chaque quantum de temps. Le temps de réaction est donc, en moyenne, d'un demi quantum. Lorsque PIOMan utilise un appel bloquant, on observe un temps de réaction constant proche du temps nécessaire à détecter un événement réseau lorsque les processeurs ne sont pas chargés. Cette méthode souffre néanmoins d'un surcoût, montrant ainsi l'utilité de pouvoir choisir la méthode de détection en fonction du contexte d'exécution.

Émetteur	*Récepteur*

`get_time(t1);`
`nmad_send(...);`
`get_time(t2);`
	`nmad_irecv(...);`
	`compute();`
	`  /* calcul pendant env. 50ms */`
	`nmad_rwait(...);`

TAB. 1 – Programme d'évaluation de la progression de communications asynchrones.

	NewMadeleine	PIOMan (scrutation)	PIOMan (appel bloquant)
Latence	3,00 µs	3,80 µs	4,59 µs

TAB. 2 – Durée d'un transfert de 4 octets.

4.2. Évaluation de la progression en arrière-plan des communications asynchrones

Afin d'évaluer l'efficacité de la progression des communications asynchrones, nous avons utilisé le programme décrit dans la Table 1. Ce programme tente de recouvrir une communication par du calcul. La Figure 3 montre les résultats obtenus sur MX/Myrinet. Pour les messages de petite taille, on observe une durée d'émission proche de la latence du réseau. Pour les messages de taille plus conséquente, au-delà du seuil de *rendez-vous*, les comportements diffèrent :

NewMadeleine – La progression du *rendez-vous* en arrière-plan n'est pas assurée par NewMadeleine. L'émetteur est donc bloqué jusqu'à ce que le récepteur atteigne l'instruction `nmad_rwait`.

PIOMan - scrutation – La version monoprocesseur de PIOMan fait progresser le *rendez-vous* lorsque l'ordonnanceur est exécuté, c'est-à-dire lors d'une interruption d'horloge qui arrive après 10 ms. Le temps d'émission est donc borné à 10 ms et non à la durée du calcul intervenant du côté du récepteur.

PIOMan - appel bloquant – La version multiprocesseur de PIOMan exécute un appel bloquant pour attendre la demande de *rendez-vous* du côté du récepteur et continue le calcul sur un thread noyau supplémentaire. Le *rendez-vous* progresse donc sans être gêné par le calcul.

PIOMan est donc capable de faire progresser les communications asynchrones en arrière-plan, permettant ainsi le recouvrement des transferts de données par des phases de calcul.

4.3. Évaluation du surcoût de PIOMan

Pour évaluer le surcoût engendré par l'utilisation de PIOMan, nous avons effectué un ping-pong dont les résultats sont présentés Table 2. La version monoprocesseur de PIOMan entraîne un surcoût d'environ 0,8 µs. Ce surcoût est principalement dû à l'introduction de la possibilité d'utiliser des threads : il est en effet nécessaire de gérer une contention et les primitives de synchronisation sont indispensables. Une partie du surcoût est liée à l'utilisation même de PIOMan qui doit maintenir une liste des requêtes en cours et des threads en attente de communication.
La version multiprocesseur de PIOMan engendre un surcoût d'environ 1,6 µs, soit 0,8 µs de plus que la version monoprocesseur. Le réveil du thread noyau supplémentaire lors d'un appel bloquant en est la principale cause. À ce réveil s'ajoute la complexification des primitives de synchronisation qui deviennent plus coûteuses.
Le surcoût logiciel dû à l'utilisation de PIOMan est donc conséquent et devra être réduit dans le futur. Ce surcoût est néanmoins relativement faible si l'on considère le temps perdu par NewMadeleine lorsque les processeurs sont tous occupés par des threads de calcul.

5. Conclusion et perspectives

L'apparition des architectures multi-cœurs et leur développement dans les grappes de calcul a rendu plus délicate la gestion des communications. L'exploitation de ce type d'architectures passe en effet de plus en plus par l'utilisation de threads au niveau de l'application, comme le montre le nombre croissant d'approches hybrides MPI+OpenMP. Cette introduction de threads

empêche pourtant une détection efficace des événements réseau. Après avoir montré les raisons de certaines de ces perturbations, nous avons présenté une nouvelle approche consistant à centraliser l'ensemble des requêtes réseau au sein d'un élément logiciel interagissant avec la bibliothèque de communication et l'ordonnanceur de threads afin d'avoir une connaissance globale de l'état de la machine et des communications. Ce modèle a été implémenté au sein de la suite logicielle PM2 et les expérimentations ont montré que la réactivité aux événements de communication est assurée sans empêcher les threads de calcul de progresser.

Le système de décision permettant de choisir la méthode de détection à utiliser est pour l'instant simpliste et risque de ne pas être adapté à certaines d'applications. Nous comptons donc analyser le comportement de ce type d'applications et créer des stratégies décisionelles adaptées. Ces stratégies pourront être implémentées sous forme de *fonctions de rappel* (*callbacks*). Par ailleurs, l'évaluation de PIOMan a montré son efficacité sur des programmes synthétiques et il reste à évaluer le système avec des applications réelles afin de vérifier que les améliorations apportées ont bien un effet sur le temps d'exécution du programme. Afin de mener à bien cette évaluation, une intégration de PIOMan dans une implémentation MPI telle que MPICH/NewMadeleine ou YAMPII/PadicoTM est nécessaire. Enfin, nous comptons étudier la possibilité d'exporter les *entrées-sorties programmées* (PIO) sur un processeur inoccupé afin d'accélérer la soumission des requêtes réseau concernant de petites données.

Bibliographie

1. Dongarra, J.J. and Meuer, H.W. and Strohmaier, E. and others : TOP500 Supercomputer Sites `http://www.top500.org/`.
2. Intel : Intel Tera-scale Computing Research Program (2007) `http://techresearch.intel.com/articles/Tera-Scale/`.
3. ANL, MCS Division : MPICH-2 Home Page (2007) `http://www.mcs.anl.gov/mpi/mpich/`.
4. Graham, R.L., *et al.* : Open MPI : A high-performance, heterogeneous MPI. In : Proceedings, Fifth International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks, Barcelona, Spain (September 2006)
5. Smith, L., Bull, M. : Development of mixed mode mpi / openmp applications. Scientific Programming **9**(2-3/2001) 83–98 Presented at Workshop on OpenMP Applications and Tools (WOMPAT 2000), San Diego, Calif., July 6-7, 2000.
6. Viet, T., Yoshinaga, T., Ogawa, Y., Abderazek, B., Sowa, M. : Optimization for Hybrid MPI-OpenMP Programs on a Cluster of SMP PCs. Japan-Tunisia Workshop on Computer Systems and Information Technology (2004)
7. Aumage, O., Brunet, E., Furmento, N., Namyst, R. : 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, Long Beach, California, USA (March 2007) Also available as LaBRI Report 1421-07 and INRIA RR-6085.
8. Runtime Team, LaBRI-Inria Futurs : Marcel : A POSIX-compliant thread library for hierarchical multiprocessor machines (2007) `http://runtime.futurs.inria.fr/marcel/`.
9. Anderson, T.E., Bershad, B.N., Lazowska, E.D., Levy, H.M. : Scheduler activations : effective kernel support for the user-level management of parallelism. ACM Trans. Comput. Syst. **10**(1) (1992) 53–79