Gestion des communications centrée sur les accès mémoire à distance

Romain Prou

Gestion des communications centrée sur les accès mémoire à distance

Romain PROU

Tuteur : Alexandre Denis
Master VIP

Année : 2014-2015

Remerciements

Tout d'abord je tiens à remercier toute l'équipe projet de TADaaM pour m'avoir accueilli et pour ces moments de camaraderie partagés durant ce stage.

Je tiens à remercier sincèrement :

— Emmanuel Jeannot, responsable de l'équipe, pour m'avoir permis cette opportunité et de poursuivre ce projet à l'avenir

— Alexandre Denis, mon maître de stage, pour toute l'aide et le support qu'il m'a apporté au cours de ce stage, et pour sa généreuse proposition de continuer l'aventure au sein de l'équipe

Je tiens également à remercier l'université d'Orléans et l'ensemble des professeurs du département d'informatique pour leur formation durant ces cinq années universitaires

Introduction

Présentation grand public

La volonté d'accélérer les calculs afin d'utiliser des programmes de plus en plus complexe a poussé la création de grappes de calculs. Ce sont des machines constitués de plusieurs unités de calculs, nommées nœuds, reliées entre elles par un réseau d'intercommunication, ou réseau HPC. Ces sont des réseaux qui fonctionnent de façon radicalement différente des réseaux usuels, car ils répondent à d'autres impératifs. Ils ont subi plusieurs modifications de fonctionnement au cours des dernières décennies, afin d'améliorer leur performance, et de faciliter leur utilisation. Les bibliothèques de communication, qui servent d'interface entre le programme et le réseau lui même, utilisent aujourd'hui de nombreuses optimisations visant à améliorer l'utilisation des réseaux HPC en prenant en compte la nature de ceux-ci. Récemment est apparu le RDMA, une fonctionnalité permettant à un nœud d'accéder à la mémoire d'un autre nœud directement. La plupart des bibliothèques ne prévoient pas de prendre parti de cette fonctionnalité par le programme directement, ou ne cherchent pas à utiliser pleinement le RDMA pour améliorer la performance de la communication elle même. On cherchera donc ici à savoir : Est-il possible d'améliorer la performance de l'utilisation d'un réseau HPC en utilisant les optimisations existantes sur un modèle de communication nativement RDMA.

Présentation Master Recherche

Les travaux sur les clusters se portent de plus en plus vers l'optimisation des communications que sur l'amélioration de la vitesse de calcul elle même. En effet, avec l'augmentation constante du nombre de cœurs par nœud, les communications se multiplient, alors que les réseaux ne poursuivent pas la même augmentation de leur bande passante. Le passage des bus à InfiniBand permet de réduire le surcoût induit par chaque communication, mais il permet

également l'utilisation du RDMA, une technologie servant à écrire ou lire directement dans la mémoire d'un autre nœud. La plupart des utilisations du RDMA se servent de sa faible latence pour accélérer leurs communications, mais ne cherchent pas à exploiter certaines optimisations que l'on utilise déjà aujourd'hui sur des opérations de *Send/Recv*. On cherchera donc à déterminer si il est possible d'améliorer la performance de l'utilisation d'un réseau HPC en appliquant certaines optimisations sur le flux de communication lui même.

Présentation chercheur

L'ancienne équipe RUNTIME d'Inria développe depuis plusieurs années une interface de communication pour les grappes de calculs nommée New Madeleine. Cette bibliothèque vise à améliorer l'utilisation d'un réseau en optimisant les flux de communications à l'aide de plusieurs méthodes : en agrégeant les messages de l'application pour profiter d'une plus grande bande passante, en divisant efficacement les envois à travers les diverses interfaces disponibles, etc. Depuis qu'InfiniBand s'est imposé comme standard de facto de communication sur les clusters en occupant une grande partie du Top500, et l'émergence des paradigmes de communication one-sided, on peut se demander si une implémentation nativement RDMA d'un tel système pourrait améliorer la performance d'utilisation d'un réseau de grappe de calcul.

Chapitre 1

Travaux connexes

Nous allons examiner les travaux relatifs aux interfaces de communication prenant parti du RDMA, ainsi que les travaux qui seront utiles pour définir un prototype d'interface de communication.

1.1 MVAPICH

La bibliothèque de communication la plus souvent considérée pour se servir efficacement du RDMA sur InfiniBand est MVAPICH[14]. Elle est développée par un laboratoire appelé NOWLAB au sein de l'Ohio State University. L'objectif initial de MVAPICH était d'implémenter efficacement MVPICH en prenant partie du RDMA pour accélérer ses communications[12]. Aujourd'hui, un grand nombre de branches du projet ont vu le jour, pour permettre l'utilisation de plusieurs fonctionnalités : l'usage de PGAS (avec Unified Parallel C) en plus de MPI au sein du programme, RoCE, etc.

L'objectif de MVAPICH est de permettre à l'application d'utiliser les capacités d'InfiniBand au mieux, quel que soit son modèle de communication. Les communications sont optimisées de façon individuelles, mais l'attention n'est pas portée sur le flux dans sa globalité. On cherchera à se détacher de l'aspect matériel pour se concentrer sur les optimisation que l'on pourra apporter au flux de données transitant en RDMA.

1.2 Bibliothèques PGAS

PGAS ou Patitioned Global Address Space est un modèle de programmation parallèle concurrent à MPI. Il part du principe que l'application possède un espace d'adresse global, partitionné par les différents threads qui peuvent

avoir des affinités à certaines parties de celui-ci. Il est utilisé par plusieurs langages, UPC, Chapel et Global Arrays entre autres, et suppose une bibliothèque de communication qui est adaptée. On s'intéressera à deux de ces bibliothèques.

1.2.1 ARMCI

Aggregate Remote Memory Copy Interface [2] est une bibliothèque qui fournit des opérations pour l'accès mémoire distant, des transferts plus complexes (scatter, gather, etc.) et plusieurs utilitaires d'exclusion mutuelle. Son but est de faciliter l'utilisation du paradigme de PGAS sur des architectures diverses et ne considère pas le RDMA mais le RMA(Remote Memory Acces), qui cherche à abstraire la mémoire afin qu'elle soit utilisée comme un espace d'adressage global et optimiser les transferts mémoires[16]. Un système de modules interchangeables dépendant du matériel permet de porter facilement la bibliothèque et de la prérégler. ARMCI est utilisée dans Global Arrays.

1.2.2 GASNet

Global-Address Space Networking est une bibliothèque qui fournit une API destinée à l'implémentation de langages et bibliothèques PGAS de façon indépentante au matériel et au langage utilisé[3]. Le but de celle-ci est d'être performante, hautement portable et expressive. L'approche utilisée par GASNet permet d'avoir une bande passante effective très proche de la bande passante théorique du matériel. Elle fournit notamment des communications collectives optimisées pour le RDMA lorsqu'il est disponible nativement[7] et permet de recouvrir agressivement les coût d'une communication en passant la gestion de celle-ci à l'interface réseau[5]. Les deux bibliothèques cherchent une optimisation des communications singulières, optimiser les transferts mémoires pour ARMCI et fournir des opérations de communication les plus optimisées pour le matériel dans le cas de GASNet. Ces approches ne veulent pas optimiser le flux de communication en lui même, cette tâche est laissée à l'utilisateur. Or c'est l'optimisation de ce flux, dont on se chargera de la gestion plutôt que l'utilisateur, qui va nous intéresser ici.

1.3 LogP

Pour analyser les coûts des optimisations utilisées, il nous faut un modèle qui permette une représentation réaliste des communications sur le ré-

seau. Plusieurs modèles sont utilisés pour représenter les machines HPC. Ils cherchent tous à s'approcher le plus du fonctionnement réel de la machine, sans être trop précis, ce qui risquerait de rendre les calculs inexploitable à cause leur complexité. LogP est un modèle de machine parallèle qui se cherche à la fois à ne pas sur-simplifier et à ne pas être trop spécifique pour être suffisamment universel. Ainsi il permet d'être exploitable pour déterminer une approximation plus juste des coûts d'utilisation du réseau que PRAM par exemple, qui néglige les coûts de transferts des messages.

Initialement[6], le modèle prévoyait l'utilisation de 4 paramètres pour représenter une machine parallèle constituée de plusieurs machines complètes reliées par un réseau d'interconnection : L la latence, représentant le délai maximal encouru par un message transitant d'un hôte à un autre ; o l'overhead, défini comme le temps où un processeur est impliqué dans le transfert d'un message ; g le gap, l'intervalle minimal entre des transmissions ou récéptions consécutives d'un message par le processeur ; P le nombre de processeurs par nœud. Ce modèle est cependant insuffisant, car il ne prends pas en compte le transfert de messages plus lourds et suppose que le coût d'envoi d'un message par le processeur est constant.

Afin de prendre en compte les différences de bande-passantes observées en cas d'envoi de messages plus lourds, une variante du modèle, nommée LogGP fut introduite[1]. Le nouveau paramètre, G, représente le temps de transfert, par octet, des messages plus longs. L'addition de ce nouveau terme permet ainsi de symboliser plus de problèmes pouvant être rencontrés. Ainsi, tandis que G permet de mieux représenter les messages longs, g permet de représenter les messages plus courts, qui sont souvent les initiateurs de communication. Il suppose par contre que l'overhead est constant et non dépendant de la taille du message.

Pour prendre en compte cet aspect, une autre variante peut-être utilisée, parameterized LogP[11]. Ce modèle utilise 5 paramètres, car il sépare o_s l'overhead d'envoi de o_r, celui de réception. De plus, o_s, o_r et g sont maintenant fonction de m la taille du message à envoyer. Ainsi, le fonctionnement réel d'une communication est représentée de façon plus fidèle.

1.4 New Madeleine

New Madeleine est une bibliothèque de communication dont le but est d'améliorer les performances des communications sur un cluster[15]. Afin de

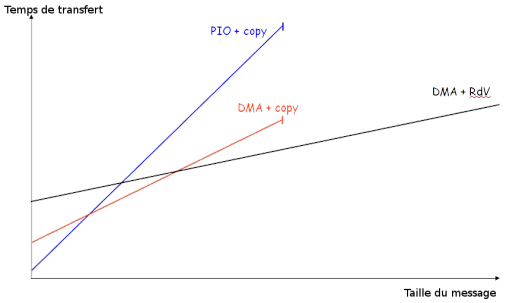

FIGURE 1.1 – Temps de transfert d'un message en fonction de la méthode utilisée

pouvoir utiliser le réseau efficacement, elle sépare les messages de l'application des messages réels à faire transiter. Ainsi, elle permet d'appliquer des optimisations sur le flux de communication lui même

En fonction du type de méthode utilisé pour transférer un message (en utilisant un RDV ou non par exemple), son temps de transfert ne sera pas le même (1.1). On peut alors chercher à déterminer le seuil idéal pour envoyer un message d'une façon ou d'une autre. Mais découpler les messages du programme des messages réels nous permet alors d'optimiser les transferts de façon encore plus efficace.

Avant de pouvoir déterminer la méthode à employer pour l'envoi d'un message donné, il est important de pouvoir déterminer les capacités d'un réseau spécifique. Contrairement à la plupart des bibliothèques de communication qui doivent recevoir un tuning dépendant du matériel avant d'être utilisées, New Madeleine passe par une phase de sampling à son démarrage[4]. Elle peut ainsi déterminer le type de matériel sur lequel le système tourne, notamment les ressources réseau, mais également la bande passante mémoire et la puissance processeur.

Le fonctionnement de New Madeleine est le suivant (1.2) : chaque paquet devant transiter sur le réseau est fourni à New Madeleine ; avant de fournir un paquet à l'interface réseau, la bibliothèque se charge de vérifier si différentes stratégies d'optimisation sont applicables au flux de données actuel en utilisant un ordonnanceur d'optimisation ; les nouveaux paquets ainsi obtenus sont ensuite fournis à un ordonnanceur principal qui se chargera de distribuer les messages sur les différentes interfaces réseau disponibles.

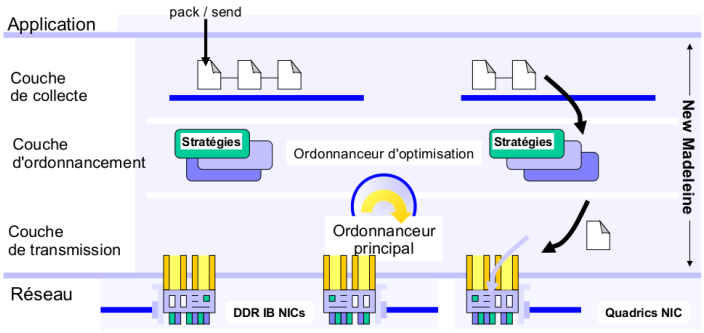

FIGURE 1.2 – Architecture de New Madeleine

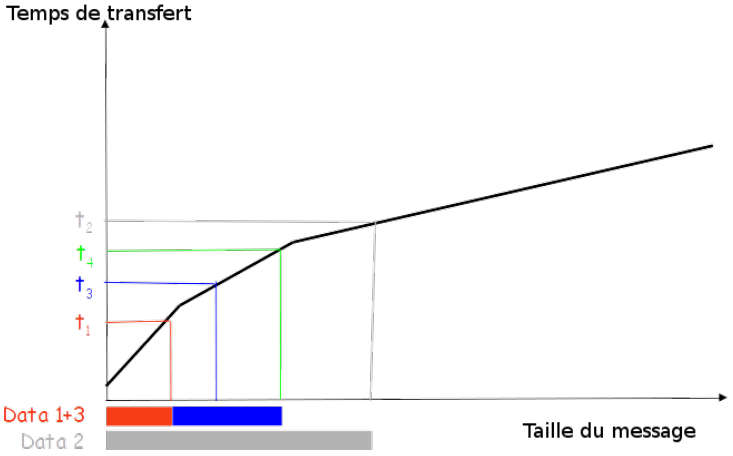

FIGURE 1.3 – Transfert de la somme de deux paquets

On peut alors appliquer plusieurs optimisations pour améliorer la performance de l'utilisation du réseau par une application. L'optimisation la plus utilisée est une aggrégation des paquets en suivant leur numéros de séquence. En fonction de la machine et de la taille des paquets, il est parfois préférable de n'envoyer qu'un large paquet au lieu de plusieurs petits. Si par exemple ici (1.3), le temps de transfert de t_4 auquel on additionne le temps nécessaire pour agréger t_1 et t_3, est plus court que l'addition des temps de transfert de t_1 et t_3, il peut-être judicieux d'effectuer l'envoi de t_2 suivi de l'envoi de t_4. On peut donc envisager de réordonner des paquets avant de les agréger afin d'améliorer leur temps de transfert.

Une autre stratégies possible est le multi-rail, qui se charge de diviser les paquets en fonction des interfaces réseau disponible et également d'équilibrer la division en accord avec la bande passante de chacune des interfaces. En fonction de la topologie des nœuds ainsi que la localité des données, le ratio

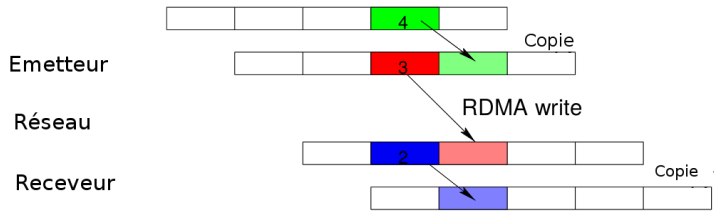

FIGURE 1.4 – Pipeline RDMA

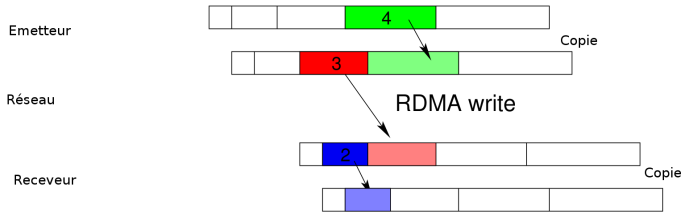

FIGURE 1.5 – Superpipeline RDMA

optimal peut parfois être sensiblement différent. New Madeleine se charge donc de calculer ce ratio optimal afin d'équilibrer au mieux la charge de l'envoi.

La dernière optimisation que nous examinerons est basée sur le RDMA[8]. Elle repose sur l'idée qu'un *RDMA_write* prend plus de temps à réaliser qu'une copie par le processeur pour la même taille de paquet. À la base, un pipeline est utilisé pour effectuer un recouvrement entre le temps de copie du message et son transfert sur le réseau. Les paquets sont donc divisés en chunks (1.4). Cependant, vu que le *RDMA_write* a de manière générale un débit plus faible que le *memcopy*, un superpipeline est utilisé. Celui ci augmente progressivement la taille des chunks à l'intérieur du pipeline en fonction du débit du *memcopy* et du *RDMA_write* (1.5), ainsi le recouvrement est plus efficace.

Dans toutes ces optimisations cependant, on ne prend pas partie du RDMA à proprement parler, mais simplement de sa vitesse de transfert plus rapide dans le cas du superpipeline, et tout simplement pas du tout dans les autre cas.

La bibliothèque fonctionne sur le paradigme traditionnel de MPI. Si l'application utilise un autre paradigme, elle se chargera de convertir ses messages de façon transparente dans un souci d'homogénéité. Les *RDMA_write* seront alors simulés en utilisant des *Send/Recv*. Cette approche n'est tout simple-

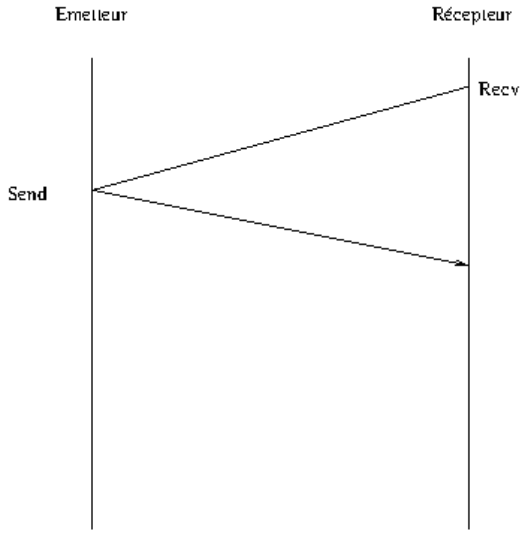

FIGURE 1.6 – Send/Recv

ment optimale lorsque le matériel supporte nativement le RDMA, car elle rajoute alors un niveau d'abstraction inutile. L'utilisation native du RDMA pourrait également à terme résoudre des problèmes aujourd'hui posés par le polling en multi-cœur créant une contention du côté du récepteur grâce à l'absence de réception explicite.

1.5 RDMA et ibverbs

La technologie au centre de ce stage est RDMA, et son implémentation dans le cadre de la bibliothèque d'utilisation d'InfiniBand, ibverbs.

1.5.1 Schémas de communication sur HPC

Le paradigme de communication prédominant dans le HPC est le *Send/Recv*, car il est au cœur de MPI, le principe de programmation parallèle le plus souvent utilisé.

Message Passing Interface est un paradigme de programmation parallèle dont le premier standard a été créé en 1994. Son utilisation a été popularisée depuis grâce à son adoption par virtuellement tous les constructeurs. Il leur permet d'avoir une base concrète sur laquelle se tabler pour faire leur interface d'utilisation du matériel.

Au centre de MPI, on trouve le *Send/Recv*, qui permet d'échanger des données entre deux hôtes. C'est l'opération la plus basique pour permettre le

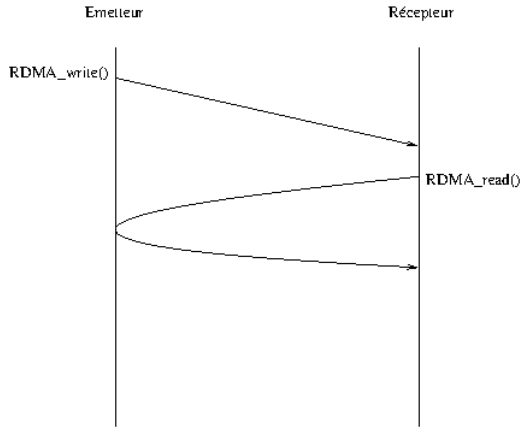

FIGURE 1.7 – *RDMA_write* et *RDMA_read*

transfert de données. Elle suppose une réception explicite de la part de l'hôte qui doit récupérer des données sous la forme d'un *Receive*. L'émetteur reçoit alors la requête, ce qui lui permet de savoir qu'il peut commencer le transfert des données. Il fait alors un *Send* pour envoyer le message au récepteur(1.6).

Pour améliorer le recouvrement des communications, il est possible d'utiliser des communications dites "non bloquantes". Lorsque le récepteur effectue un *Receive*, il ne peut pas poursuivre son exécution tant qu'il n'a pas reçu les données. Le récepteur peut donc, dans le cas d'une communication non-bloquante, fournir un buffer qui sera utilisé lorsque l'émetteur sera prêt à émettre, et vérifier par la suite si le transfert a bien été effectué avant d'utiliser les données.

Le RDMA ou Remote Direct Memory Access, est un autre type de modèle de communication[18], qui permet d'écrire ou de lire directement dans la mémoire d'un hôte distant. La création du standard pour le RDMA a été motivé par plusieurs groupes industriels, notamment le RDMA Consortium et le DAT Collaborative. Certaines de ces entreprises sont également responsables de la création d'InfiniBand.

A l'époque de la création du standard, il existait beaucoup d'inquiétudes face à l'engorgement imminent des communicateurs. En effet, avec l'accélération constante de la bande passante, le temps de copie de la part du processeur pour mettre les données à disposition de la carte réseau devenait de moins en moins négligeable, menaçant de devenir un jour le limiteur de la vitesse de la communication. Afin de résoudre ce problème, il fallait trouver un moyen de transmettre des données en s'affranchissant du processeur.

Le RDMA permet donc de copier des données directement d'un hôte en

n'utilisant qu'au minimum le processeur. Ainsi, il initie un transfert en signifiant à la carte les zones mémoires concernées. C'est la carte qui se chargera de copier les données d'un coté à l'autre du réseau. La gestion de la mémoire à distance est faite en utilisant des zones mémoire bloquées dont les adresses sont connues par l'adaptateur. Dans le cas d'un *RDMA_write*, un bloc de données est écrit dans une zone spécifiée, alors que qu'un *RDMA_read* se chargera de récupérer les données dans la mémoire d'un hôte distant pour venir les écrire en local(1.7).

L'utilisation de RDMA, bien que marginale, commence à gagner de le popularité avec l'utilisation de plus en plus fréquente de protocoles tel que PGAS et des communications one-sided spécifiées dans les standard MPI les plus récents. Plusieurs technologies implémentent le RDMA. En effet, il est à la base du développement d'InfiniBand, mais également iWARP et RoCE.

1.5.2 Le cas d'Infiniband

La technologie de réseau d'inter-connection la plus représentée aujourd'hui dans le monde du HPC est InfiniBand. C'est donc sur elle que nous nous concentrerons.

Le standard d'InfiniBand a été crée par le groupe industriel InfiniBand Trade Association[10], constituée de géants tels que IBM, Mellanox, Oracle, HP, Cray, ainsi qu'Intel. Le standard visait à la base à créer une technologie de communication pour remplacer le PCI, Ethernet, les réseaux pour clusters et Fibre Channel. Dû à l'abandon de Microsoft sur le développement ainsi que la priorité d'Intel à développer PCI Express, son application fut cependant moindre.

Depuis la création du standard, il finit par s'implanter dans le monde du HPC. En 2014 il représentait 51.4% des types de réseaux utilisés dans le TOP500. InfiniBand est donc devenue une technologie majoritaire dans le monde du HPC.

Afin d'utiliser une carte InfiniBand, il faut se servir d'une interface appelée *ibverbs*. La bibliothèque prévoit deux sémantiques différentes, le mode par canaux, pour utiliser des *Send/Recv* et le mode mémoire, pour le RDMA[13].

Le mode par canaux prévoit l'allocation d'un canal de communication entre deux cartes InfiniBand à la manière de TCP, avant de pouvoir échanger des *Send/Recv* entre les deux machines ainsi connectées. Il est toutefois nécessaire d'avoir échangé les adresses des cartes à l'aide d'un mécanisme out of band. Les deux nœuds impliqués dans la communication doivent créer

une *Queue Pair* ou QP, une structure de donnée contenant deux files, une destinée à recevoir les requêtes d'envoi et l'autre pour celles de réception. Le récepteur doit poster une requête de réception (*Receive*)contenant l'adresse du buffer dans lequel doivent être placées les données. Lorsque l'émetteur poste une requête d'envoi (*Send*), la carte réseau transmet les données à la carte distante, qui se chargera de consommer une requête de la file de réception et de placer les données à l'adresse spécifiée. Si le processeur n'est pas impliqué dans la copie en elle même, cette pratique suppose une réception explicite, ainsi qu'un polling pour vérifier si les données ont été transmises.

Dans une communication en mode mémoire, il n'est pas nécessaire d'allouer de canal. Un échange d'adresse est toutefois également requis à l'initiation d'une connexion ; il est en général effectué au démarrage de la bibliothèque de communication. Divers objets sont ensuite utilisés pour effectuer une opération.

Pour que la carte puisse accéder directement aux buffers sans avoir besoin de faire une interruption, il est nécessaire que celle-ci possède une copie d'une partie de la TLB, et que les adresses de ceux- ci ne changent pas. Il faut donc les bloquer en utilisant une structure spécifique, la *Memory Region* ou MR. Tous les buffers doivent être contenus à l'intérieur, et celle-ci a une taille limitée par le type de carte utilisé.

Pour effectuer des requêtes d'opérations RDMA, comme pour le *Send/Recv*, il est impératif d'avoir instancié une QP. Celle ci recevra les requêtes d'envoi, et de réception dans le cas d'une réception explicite. Une réflexion explicite est nécessaire dans le cas d'un *RDMA_write* avec des données immédiates, afin de pouvoir spécifier où les placer. Dans le cas de cette étude, nous n'en utiliserons donc pas.

Dans les QP, il faut alors poster des requêtes, sous la forme de *Work Request* ou WR afin d'effectuer des opération. C'est dans la WR qu'on spécifie le type d'opération ainsi que les adresses de lecture ou d'écriture. On utilise pour ça une *Scatter/Gather Element List* ou SGE List. Il est possible de spécifier plusieurs SGE par opération, pour lire des buffers non contigus et les concaténer avant d'écrire dans le cas d'un *RDMA_write*, ou écrire les données lues par un *RDMA_read* à l'intérieur de plusieurs buffers en local. A l'aide de ces structures qui permettent une certaine flexibilité des opérations RDMA, on pourra implémenter plusieurs optimisations en vue d'améliorer la performance de l'utilisation du réseau.

Chapitre 2

Contribution

Pour essayer d'améliorer l'utilisation d'un réseau nativement capable de communications en RDMA, nous allons examiner les coûts de différents types de communication afin de pouvoir établir un concept de protocole permettant d'utiliser le réseau plus efficacement.

2.1 Analyse théorique des performances

2.1.1 La latence

On désigne habituellement la latence comme le temps de transfert d'un message vide. Celle ci dépend cependant de la définition de l'origine et de la destination du message. On utilisera deux abstractions afin de définir deux types de temps de transfert.

Lorsque le message est émis par l'application pour être traité par une autre application sur l'hôte récepteur, on parlera de latence soft. Celle ci ne compte pas seulement le temps de transfert "sur le fil" à proprement parler, mais également le temps de traitement par les différentes couches du protocole de communication.

Si le message en revanche n'est pas supposé passer à travers toutes les couches du protocole, sa latence sera plus faible. On supposera ici que dans le cas d'un RDMA Read, le message de retour, contenant les données à proprement parler, ne passe que sur la couche matérielle du protocole de communication. On parlera donc de latence hard.

2.1.2 Overhead

Lors d'un envoi, les différents acteurs de la communication ont un temps de traitement, parfois répartis sur plusieurs sous-acteurs, au cours duquel ils sont occupés à traiter le transfert à proprement parler. C'est ce que LogP désigne comme l'overhead. On séparera l'overhead que l'on considérera en plusieurs catégories : tout d'abord l'overhead d'émission et l'overhead de réception, tel que l'on peut voir dans parametrized LogP, mais on distinguera également l'overhead CPU.

Lors d'un envoi, le processeur n'effectue que très peu de calculs, c'est la carte qui se charge en très grande partie d'effectuer le transfert à proprement parler. Nous négligerons donc ce temps par la suite, afin de se concentrer sur le temps d'occupation de la carte.

Pour envoyer des données sur le fil, la carte doit lire les divers paramètres qui lui ont été fournis puis effectuer le transfert à proprement parler. Durant ce temps, la carte n'est pas disponible pour effectuer un autre envoi. Il est fonction de la taille du message. On le mesure en observant le délai entre le processing de la requête d'envoi et l'acquittement d'envoi. On notera l'overhead d'émission pour un message de taille m, $O_s(m)$

Dans le cas du RDMA, où la réception n'est pas active, celui ci est virtuellement inexistant. On parlera d'overhead de réception dans le cas d'un envoi en *Send/Recv* classique, mais le concept n'apparaîtra pas dans les analyses de coût de communications RDMA.

2.1.3 Gap

Lors d'une communication en continu, il existe un temps entre le traitement d'un message et le suivant. C'est cet intervalle que LogP désigne comme étant le gap. Dans le cas d'une communication sur un réseau "classique", c'est le processeur qui est sujet à cet intervalle.

Ici, le processeur n'étant que l'initiateur de la communication, c'est la carte qui possède un gap entre le processing d'une requête et la suivante. On pourrait penser que celui-ci serait dépendant de la taille du message envoyé précédemment, mais comme on verra par la suite, ce n'est pas le cas. On considérera le gap comme étant une constante, dépendante uniquement du type de carte utilisé.

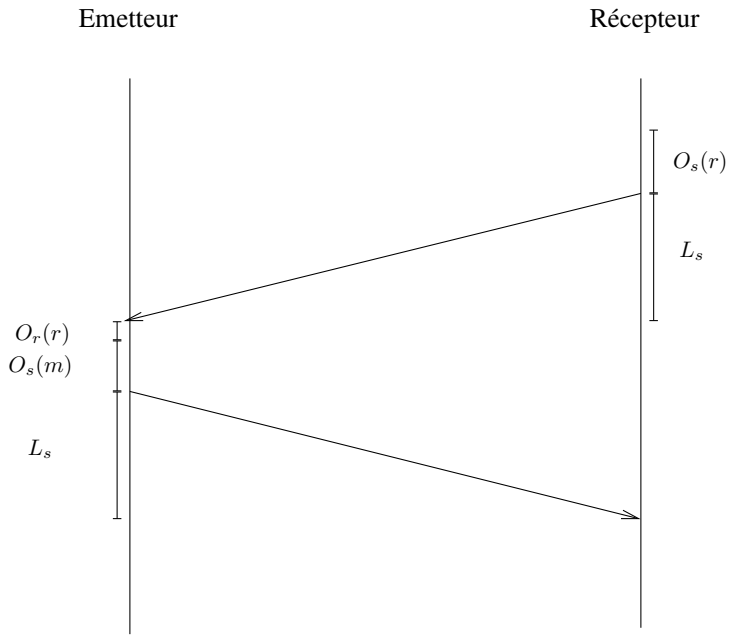

FIGURE 2.1 – Send/Recv

2.2 Différents types d'envoi

2.2.1 Send/Recv

Ce mode de communication étant le plus commun, l'API d'IB fournit un moyen de l'utiliser. Il faut alors utiliser des communications sur des canaux. Cela implique d'établir un canal de communication pour pouvoir communiquer entre deux hôtes. Les structures mises-en place en revanche ne changent pas, il est toujours nécessaire d'utiliser des queues d'envoi et de réception ainsi qu'une queue de complétion afin de pouvoir consulter l'état d'une communication.

Outre le surcoût induit par l'établissement d'un canal de communication, l'utilisation d'un receive explicite impose des contraintes sur le coût de ces opérations. En effet, utiliser un receive explicite demande à l'hôte de réception d'entamer la communication, ce qui veut parfois dire avoir une congestion sur l'émetteur, mais cela induit également un polling de la part du récepteur, soit un surcoût qui peut parfois devenir non négligeable.

Outre l'échange d'adresses, nécessaire à tous les types de communication, et les divers surcoûts induits par l'établissement d'un canal de communication

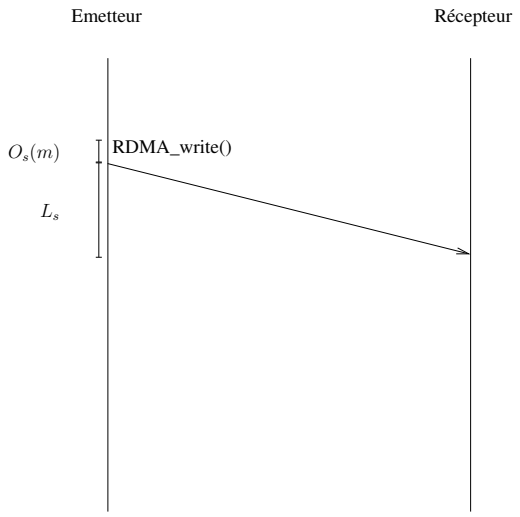

FIGURE 2.2 – Eager Send

au début de celle-ci qui sont de l'ordre de quelques latences, l'utilisation du *Send/Recv* introduit également un coût de traitement de la carte réceptrice, ainsi que le cout de propagation de la requête de réception. On notera la taille du message de la requête de réception r, et le message en lui même m. Le coût est donc de : $2L_s + O_s(r) + O_r(r) + O_s(m) + O_r(m)$. Pour envoyer n messages à la suite, de taille m_1 à m_n, on a donc un cout minimum de : $O_s(r) + O_r(r) + \sum_{i=1}^{n} O_s(m_i) + (n-1)g + 2L_s$. Quel que soit le nombre de messages à envoyer, il existe donc un facteur constant : $O_s(r)+O_r(r)+2L_s$. En revanche, $\sum_{i=1}^{n} O_s(m_i)+(n-1)g$ est une composante linéaire qui est dépendante du nombre de messages à envoyer, mais également de leur taille.

2.2.2 Eager Send

Le eager send en RDMA (2.2) est la forme la plus simple de communication. Lorsque l'émetteur doit envoyer un message quelconque, il le fait dans un endroit de la mémoire du récepteur qui lui est dédié. Bien sûr cela pose des questions vis-à-vis du contrôle de flux ainsi que vis à vis de la notification du récepteur que nous exposerons par la suite.

En utilisant ce mode de communication, on s'affranchit de réception explicite, et ce faisant d'une partie non négligeable du temps de transfert d'un message. Bien sûr ce type de communication n'utilise pas de canaux, ce qui théoriquement réduit considérablement le coût de l'établissement d'un

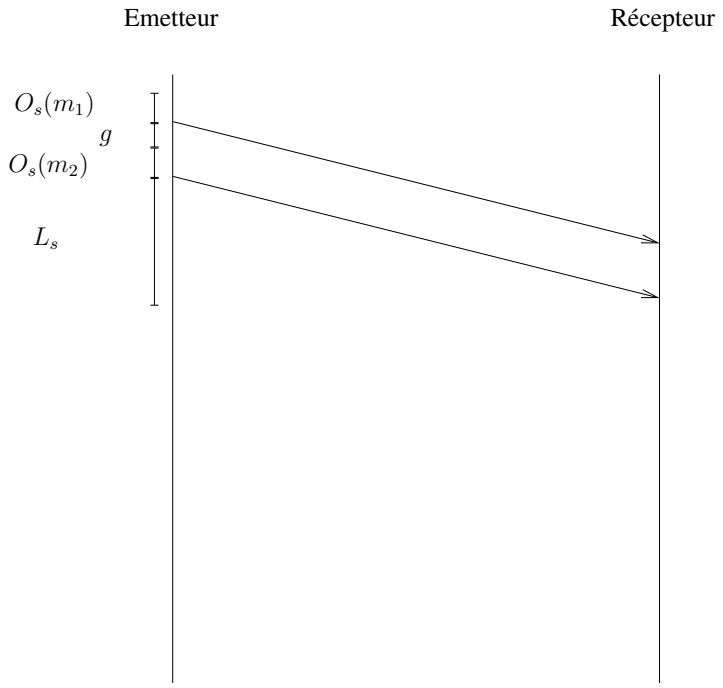

FIGURE 2.3 – Eager Send de deux messages consécutifs

échange de données.

Le eager send suppose donc un envoi simple d'un hôte à l'autre, soit un cot pour un message de longueur m de : $O_s(m)+L_s$ En supposant que l'overhead d'émission d'un message en RDMA est au plus égale à l'overhead d'émission du même message en mode Send/Recv, l'économie est visible. Mais on peut aussi vouloir envoyer plusieurs messages à la suite (2.3). Pour un envoi de n messages de tailles m_1 à m_n, on a alors : $\sum_{i=1}^{n} O_s(m_i) + (n-1)g + L_s$. L'unique partie constante est donc L_s, soit une unique latence. On retrouve cependant la même partie linéaire dépendante du nombre de messages à envoyer et de leur taille.

Il est néanmoins nécessaire de mentionner qu'un tel envoi nécessite des coûts annexes additionnels ainsi que la mise en place de structures adaptées. Afin de pouvoir utiliser un buffer dédié, il doit être créé au préalable, ainsi qu'un mécanisme de contrôle de flux pour éviter l'écrasement des données pur et simple. L'allocation du buffer elle même peut-être effectuée une unique fois au démarrage du protocole de communication, en revanche, le contrôle de flux engendre un surcoût qui doit être pris en compte.

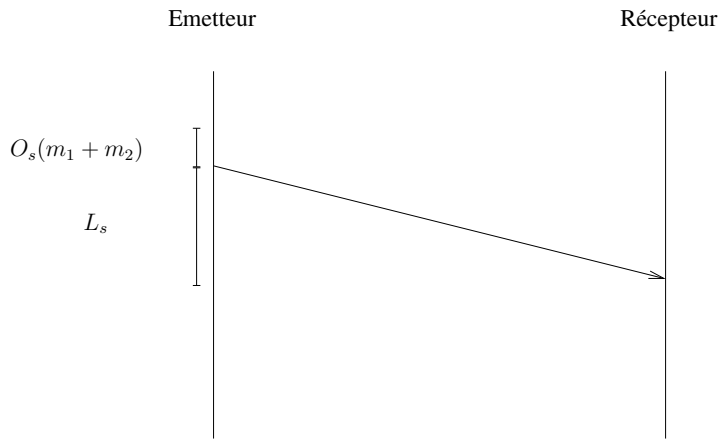

FIGURE 2.4 – Gather de deux messages

2.2.3 Gather coté émetteur

Il est possible d'amortir certains coûts pour une communication de plusieurs messages en ne formant qu'un unique message à être divisé par le récepteur. L'API d'infiniband fournit un moyen efficace permettant de concaténer différents messages à envoyer en un seul message lors du *RDMA_ write*.

Afin d'envoyer les mêmes n messages que précédemment le coût théorique est donc de $O_s(\sum_{i=0}^{n} m_i) + L_s$(2.4). On économise les gaps car l'envoi de ces messages n'est qu'une unique requête. Le seul coût constant reste d'une unique latence, comme précédemment. En revanche, la partie linéaire est modifiée, elle ne dépend plus du nombre de messages, mais de la somme de la taille de tous les messages. Il est donc nécessaire de déterminer si $O_s(\sum_{i=0}^{n} m_i)$ est inférieur à $\sum_{i=1}^{n} O_s(m_i)$ pour vérifier si le coût de l'envoi agrégé est inférieur à un eager send, sans le coût d'agrégation lui même.

Le message réceptionné est en revanche virtuellement illisible si le récepteur ne sait pas à l'avance quel est le découpage à effectuer sur ces données. Un moyen pour informer le récepteur de ce découpage doit donc être devisé. Ce découpage peut lui même produit un surcoût CPU si les données ne sont pas consommées in situ.

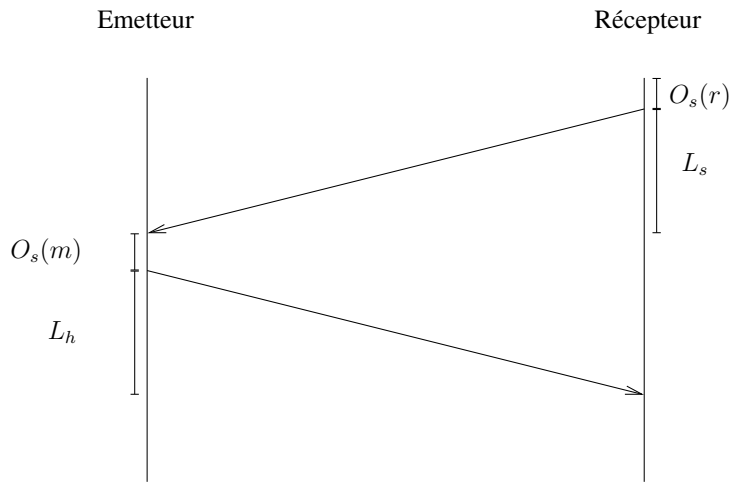

FIGURE 2.5 – RDMA_Read

2.2.4 Scatter coté réception

Pour résoudre le problème du découpage du coté de la réception, il est envisageable d'utiliser le récepteur pour engager la communication. En effet IB nous fournit un moyen simple de pouvoir découper un bloc contigu récupéré par *RDMA_read* et de le placer aux adresses correspondantes.

L'utilisation d'un *RDMA_read*(2.5) implique par contre de subir un délai d'une latence hard de plus, le message devant effectuer un "aller-retour" sur le fil. On appellera r la longueur de la requête envoyée dans un RDMA read servant à récupérer un message de longueur m. Le coût de récupération d'un tel message est de $O_s(r) + L_s + O_s(m) + L_h$. Pour n messages, on utilisera un header r' qui sera d'une taille supérieure à r mais inférieure à $\sum_{i=1}^{n} r_i$ si la requête r_i est celle qui sert à récupérer le i-ème message. Le coup de récupération d'un bloc de messages est de $O_s(r') + L_s + O_s(\sum_{i=0}^{n} m_i) + L_h$. La partie constante du coût est plus importante : $O_s(r') + L_s + L_h$, c'est le coût de l'envoi de la requête initiale. La partie linéaire est la même que celle du gather :$O_s(\sum_{i=0}^{n} m_i)$. Il convient de noter également qu'une économie en temps de processing des données par rapport au scatter n'est pas montrée ici, car si le scatter ne consomme pas les données directement dans le buffer, c'est le processeur qui sera chargé de copier les données à leur emplacement final, alors que dans le cas d'un scatter, c'est la carte hca qui sera chargée de les placer.

Si le récepteur initie l'échange de données, il lui est impossible de garantir la disponibilité de celles-ci. L'émetteur doit donc d'une façon ou d'une autre qu'il est prêt à transmettre mais également spécifier le découpage du bloc qui doit être récupéré. On utilisera un message supplémentaire de l'émetteur au récepteur pour convoyer ces informations. Cette pratique induit cependant un surcoût non négligeable sur la communication.

2.3 Adresses de réception

Les deux types d'envois agrégés ont néanmoins besoin de spécifier la localisation des données à transmettre, que ce soit pour pouvoir les récupérer dans un *RDMA_write*, ou pour savoir où les replacer après avoir fini un RDMA read. Nous allons donc mettre en place des systèmes adaptés.

2.3.1 Gather

Comme énoncé précédemment, pour que le récepteur puisse exploiter les données qui lui ont été envoyées par un gather, il est impératif qu'un entête soit affixé aux données afin de spécifier le découpage de celles-ci. Une structure possible de cet entête consiste à encoder le nombre de segments différents, puis des couples (taille, numéro de séquence). De cette façon, il est possible de savoir quel message de l'application correspond à un segment donné. Il est également envisageable d'utiliser des offsets vers des adresses spécifiques en lieu de numéro de séquence si l'application spécifie les localisations des données sur l'hôte de réception.

Pour ajouter la propagation de cette entête au coût d'un envoi de type gather on doit considérer une longueur d'entête pour un unique segment, h, qui est ensuite multipliée par n, le nombre de segments au total. On compte également c, la longueur de l'encodage de n dans l'entête. On obtient alors $O_s(\sum_{i=1}^{n} m_i + (h * n) + c) + L_s$. Le coût de la partie linéaire augmente donc, d'un facteur non dépendant de la taille des messages, mais uniquement au nombre de segments agrégés.

2.3.2 Scatter

Afin de pouvoir savoir quelles sont les données qui sont prêtes, ainsi que leur emplacement, il faut que l'émetteur dans un scatter envoie une structure contenant ces informations au récepteur avant que celui ci ne puisse effectuer un *RDMA_read*. Il est envisageable que cette structure soit similaire

à l'entête décrit pour le gather, et de la même façon utiliser des numéros de séquence ou des offsets mémoire en fonction de l'hôte chargé de gérer les adresses où les différents segments doivent être réceptionnés. On notera que si des numéros de séquence sont utilisés, cela implique que le récepteur effectue un traitement pour déterminer où placer les données en fonction d'instructions qui lui ont été fournis au préalable par l'application. On ne considérera pas ce coût, car il n'est pas dépendant de la communication elle même.

On utilisera un message de longueur $h * n + c$ pour le demi rendez-vous, h étant la longueur d'encodage d'un segment singulier et c la taille d'encodage de n. On ajoute alors le coup d'un eager send de ce message au coût du scatter déterminé précédemment : $O_s(h * n + c) + O_s(r') + 2L_s + O_s(\sum_{i=0}^{n} m_i) + L_h$. La partie constante du coût total d'un envoi est donc augmentée d'une latence soft. On remarquera aussi que le terme ajouté est dépendant du nombre de segments et non de leur taille, comme pour le gather. Les deux envois agrégés sont maintenant capables de transmettre la localisation des données qu'ils doivent traiter.

2.4 Contrôle de flux

Ces différents types d'envois peuvent également souffrir de problème de contention, car le RDMA ne permet pas de vérifier si une donnée a déjà été consommée. Il nous faut donc examiner des solutions de contrôle de flux.

2.4.1 Eager Send

Afin d'être sûr de ne pas écraser de données à chaque envoi, il est impératif de contrôler le flux d'un eager send. Un premier système envisageable consiste à utiliser des flags d'occupations, qui sont échangés en fonction de l'état actuel du buffer(2.6). En revanche, un tel système ajoute des coûts de communications qui ne sont pas négligeables. Si le flag est codé en mémoire juste avant le début du buffer de données, il est possible de le modifier en même temps que l'envoi se fait. En revanche, l'acquittement doit impérativement passer via un message du récepteur vers l'émetteur, et coûte donc au moins une latence supplémentaire. De plus, ce système signifie qu'un unique message peut-être envoyé à la fois, car il doit être consommé (ou déplacé) avant que le message suivant ne puisse être envoyé.

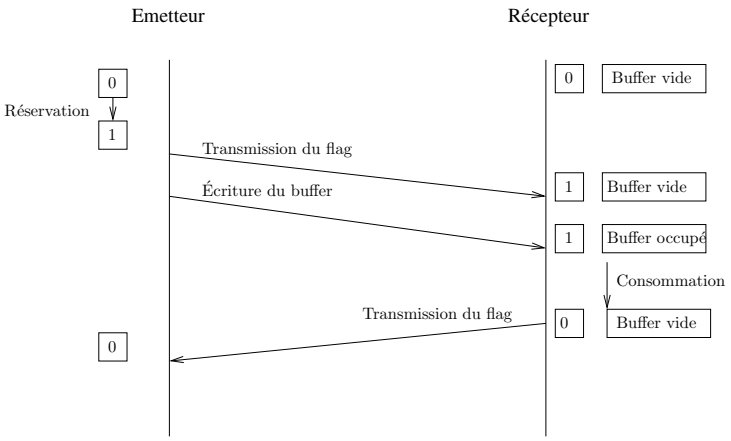

FIGURE 2.6 – Utiliser un flag pour contrôler le flux

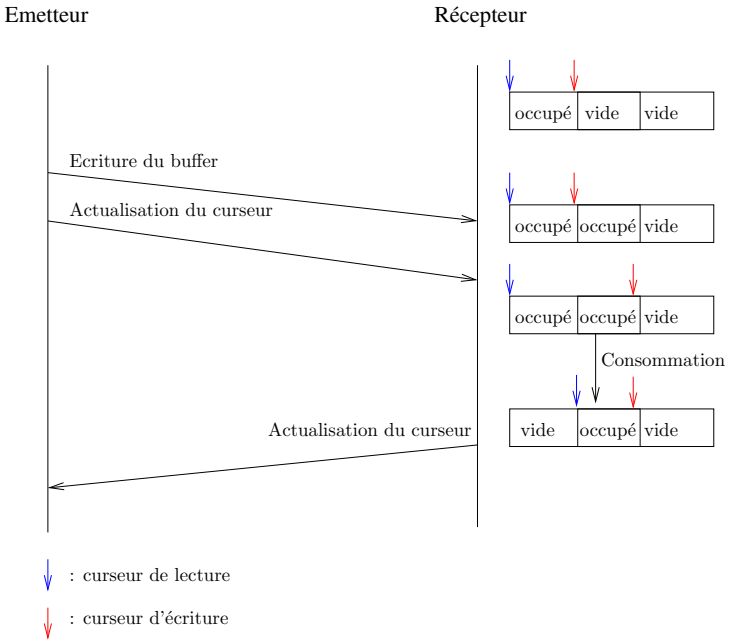

FIGURE 2.7 – Buffer circulaire

Ce système est améliorable en utilisant un buffer d'écriture divisé en différents blocs avec un fonctionnement circulaire(2.7). Lorsque le bloc est utilisé, l'émetteur doit signifier au récepteur là où son curseur d'écriture se trouve. Le récepteur doit également transmettre où son curseur de consommation se trouve, ce qui permet aux deux hôtes de savoir quels sont les blocs utilisés et utilisables. L'astuce précédente pour écrire le flag juste avant le buffer de données n'est plus utilisable aussi facilement, il faut rajouter une zone entre chaque bloc à cet effet, mais cela implique que le récepteur doit effectuer un parcours de la totalité de ces inter-blocs afin d'obtenir l'état actuel du buffer. La pratique devient alors dépendante du nombre de blocs et son efficacité par rapport à un envoi de la valeur du curseur dans un autre message devient discutable. Pour améliorer le nombre de messages à utiliser pour signifier l'utilisation en écriture des buffers dans le cas d'un envoi en rafale, il est également possible pour l'émetteur de modifier la valeur de son curseur avec un pas équivalent au minimum entre le nombre de messages en attente d'envoi et le nombre de blocs libres.

Les messages de modification de valeurs de flags peuvent être considérés de taille quasi-nulle. On obtient alors le coût suivant pour un échange total d'un unique message : $O_s(0)+O_s(m)+2L_s$. En supposant que le temps de consommation de chaque message est négligeable, on obtient un surcoût significatif sur le coût minimal avec cette technique : $\sum_{i=1}^{n} O_s(m_i) + (2n)L_s + nO_s(0)$, soit une augmentation de $(2n-1) * L_s + nO_s(0)$. Le temps de consommation lui même n'étant pas négligeable, cette technique n'est bien évidemment pas viable. En utilisant le système d'écriture par blocs, on peut éventuellement avoir un recouvrement des messages servant à spécifier le curseur de lecture de la part de la réception. Si on considère également que le pas de modification du curseur est équivalent au nombre de messages à envoyer (meilleur des cas), on obtient alors le coût suivant :$O_s(0) + \sum_{i=1}^{n} O_s(m_i) + ng + 2L_s$. Le surcoût n'est plus que de $O_s(0)+ng+L_s$, qui dépend du nombre de messages à envoyer. C'est un coût minimal dans un cas idéal cependant, qui suppose qu'il y a au moins autant de blocs libres que de messages à envoyer. Si ce n'est pas le cas, il convient de rajouter les coûts d'envoi des messages de libération de blocs (qui ne sont plus recouverts car la carte d'émission est en attente), ainsi que le délai d'attente de consommation des données, qui est tout simplement indéterminable. Pour éviter d'obtenir ces désagréments, on doit s'assurer que le temps nécessaire au récepteur pour consommer un message de taille m soit inférieur à $O_s(m)$ ou sinon d'assurer que le nombre de blocs soit toujours supérieur au nombres de messages qui sont en attente,

ce qui implique d'avoir déjà analysé le trafic de l'application précédemment.

2.4.2 La gestion des RDV

Afin de pouvoir utiliser des RDV, il est possible d'utiliser une structure de buffer circulaire similaire à celle décrite au dessus. La taille du buffer nécessaire est cependant bien moindre, la taille des structures à propager est bien moindre. Pou avoir plusieurs zones de mémoire à récupérer par le récepteur, il est nécessaire d'ajouter également à chaque structure l'adresse du début du buffer à récupérer. Une taille de blocs suffisante doit être utilisée afin de permettre l'utilisation d'un nombre adéquat de segments, et ainsi ne pas rendre l'utilisation du scatter inutile. En effet le cas échéant causerait une congestion qui non dépendante de la vitesse de récupération des données, mais simplement dûe à l'overhead nécessaire pour encoder le début de la zone mémoire et le nombre de segments par rapport à la taille de l'encodage de la localisation.

Les messages d'acquittement de la lecture des blocs peuvent être omis, ils sont soit tout simplement recouverts, soit ils peuvent être envoyés de façon suffisamment sporadique pour que leur coût soit négligeable par rapport à celui des transmissions "utiles". En revanche, il est nécessaire de considérer le coup de modification du curseur d'écriture par l'émetteur, car il doit être effectué à chaque rendez-vous. L'alternative est d'utiliser encore une fois des interstices entre les blocs pour signifier de l'état d'occupation du bloc suivant, ce dont l'efficacité est à déterminer. Pour un échange complet sans utiliser cette technique on obtient le coût minimal suivant : $O_s(0) + g + O_s(h * n + c) + O_s(r') + 2L_s + O_s(\sum_{i=0}^{n} m_i) + L_h$. Le surcoût induit est de $O_s(0) + g$, qui est constant. En utilisant ces divers systèmes, on peut maintenant garantir que la transmission de données n'écrasera pas d'informations non traitées, quel que soit le type d'envoi.

2.5 Concept du protocole

Il est maintenant possible de deviser un concept de protocole afin d'utiliser nativement les trois types d'envoi par RDMA. Le type d'envoi utilisé peut ainsi être choisi en fonction de divers critères, la taille du message par exemple, mais également les nécessités de l'application.

Intuitivement, l'envoi par gather semble plus rapide dans la majorité des cas. En revanche, il crée une surcharge au niveau du récepteur, et le coût de l'agrégation n'est pas forcément négligeable du coté de l'émetteur en fonction de l'implémentation utilisée. Pour évaluer si une salve de message sera envoyée via Eager send ou gather, il faut examiner si $O_s(0) + \sum_{i=1}^{n} O_s(m_i) + ng + L_s < O_s(\sum_{i=1}^{n} m_i + hn + c)$. Si le nombre de messages n'est pas élevé, le coût créé par l'ajout des headers pourra être plus élevé que les divers gaps ajoutés par le eager send. Pour évaluer l'efficacité d'un eager send par rapport à un scatter, on évaluera alors $\sum_{i=1}^{n} O_s(m_i) + (n-1)g < O_s(h * n + c) + O_s(r') + O_s(\sum_{i=1}^{n} m_i) + L_h$. Et pour déterminer si le scatter sera plus rapide que le gather : $O_s(h * n + c) + O_s(r') + L_s + O_s(\sum_{i=1}^{n} m_i) + L_h + O_s(0) + g < O_s(sum_{i=1}^{n} m_i + hn + c)$. Il faut encore une fois déterminer si $O_s(a + b) < O_s(a) + O_s(b)$. Le plus avantageux entre les deux dépend également du matériel et des besoins de l'application, en fonction des coûts induits par les manipulations mémoire diverses. On peut donc envisager d'avoir une certaine flexibilité de configuration qui serait modifiable manuellement afin de pouvoir profiter pleinement des bienfaits d'un type d'envoi ou d'un autre.

Pour déterminer le résultat de ces évaluations, il est impératif de tester la capacité du réseau. Pour ce faire, on cherchera à déterminer les différentes valeurs du parametrized LogP au démarrage du protocole. Les test suivants sont nécessaires : une mesure du gap par saturation du lien, un ping-pong pour avoir la latence moyenne et la mesure du délai de processing des requêtes en fonction de la taille du message. Il est alors possible d'examiner les valeurs des différents coûts pour décider des seuils de taille de chacun des envois. Il est également possible de sauter cette étape afin d'utiliser une configuration manuelle des seuils, ou même utiliser les numéros de séquence des messages pour décider.

Il est nécessaire de considérer la structure mémoire demandée par ce protocole (2.8). La zone bloquée n'ayant pas une taille infinie, il faut atteindre un équilibre vis à vis de la taille allouée aux différents buffers.

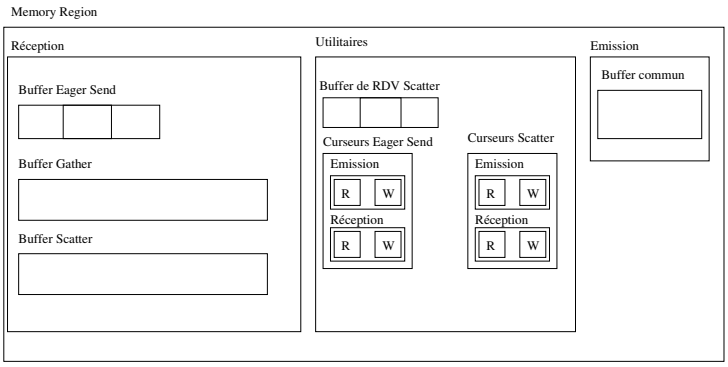

FIGURE 2.8 – Structure de la MR du protocole

Chapitre 3

Implémentation

Pour tester la validité du protocole conceptualisé, il faut pouvoir effectuer les mesures nécessaires sur le réseau, ainsi que vérifier la validité des coûts en fonction des différentes tailles de messages. La plupart des mesures ont été effectuées sur les machines expérimentales de l'équipe RUNTIME, les daltons en particulier william0/1 (carte FDR) et jack0/1(carte QDR) et joe0/1 (carte DDR), les autres ont été faites sur des nœuds "fourmi" de PlaFRIM([17]).

3.1 Mesure des paramètres du modèle

3.1.1 Mesure de l'overhead

Par souci de complétion, on commencera par mesurer l'overhead cpu induit par l'envoi d'une requête RDMA. Il faut donc mesurer le temps de processing de l'instruction.
Le temps est quasi-constant, et ne dépends pas du type de carte utilisé, le plus gros coût est dû aux *memcopy* à la volée qui sont parfois cachés, notamment en cas d'envoi inline. On a donc fait varier la taille du message et mesuré l'overhead cpu (3.1). On observe que même si la courbe semble chaotique, les déviations sont de l'ordre du dixième de micro-seconde, et ne semblent pas réellement influencées par la taille du message.

Pour déterminer l'overhead d'envoi, il faut mesurer le temps entre la fin du processing de la requête par le processeur et la complétion de l'envoi. L'implémentation de la bibliothèque IB envoie un event de complétion dans une queue spéciale, il faut donc mesurer le temps entre la fin de l'instruction de requête et l'event de complétion. On fait ensuite varier la taille du message pour avoir l'overhead en fonction de celle-ci (3.2). On voit aisément que bien

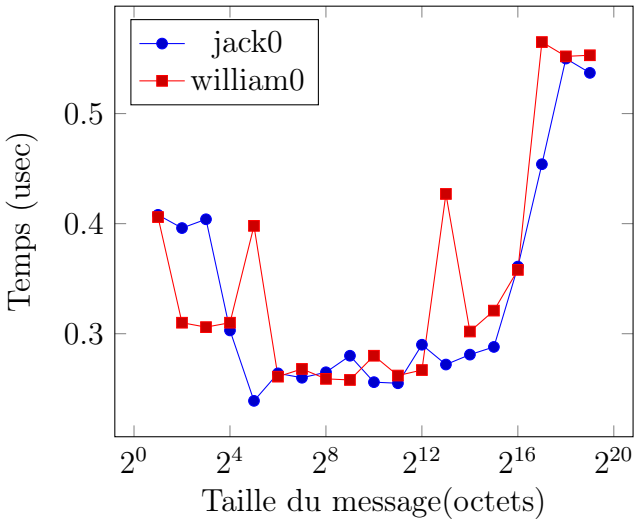

FIGURE 3.1 – Comparatif des temps d'overhead cpu

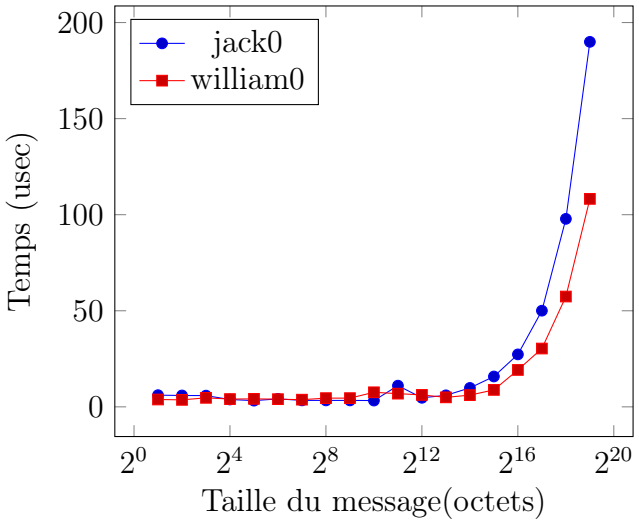

FIGURE 3.2 – Comparatif des temps d'overhead d'envoi

que le coût d'envoi des messages augmente avec sa taille, cette augmentation n'est pas linéaire.

3.1.2 Mesure du gap

La mesure du gap sur un réseau correspond au temps minimal entre deux envois consécutifs. Il couvre d'habitude également le temps d'envoi par la carte. La raison pour cela est qu'il arrive, pour des longs messages, que O_s et O_r se recouvrent. Ici cependant, l'overhead de réception étant inexistant, nous considérerons le gap comme étant le downtime de la carte entre deux envois. Pour mesurer le gap, on tachera de saturer le lien avec des messages vides et diviser le temps obtenu, en soustrayant $O_s(0)$. Après la convergence, on obtient une approximation du gap par type de carte.

Pour toutes les machines testées(jack0, fourmi), cette valeur tourne autour de 0.22μs, avec un écart type de moins de 0.03μs. On peut affirmer que cette valeur est constante et ne varie pas en fonction de la machine.

3.1.3 Mesure de la latence

Même si la latence d'un réseau est une valeur constante, il nous faut la mesurer, car celle-ci peut varier en fonction du type de carte utilisée.

Pour la mesurer, on effectue un ping-pong, auquel on prendra soin de retirer le coût d'envoi du message initial et d'un message vide. On obtient alors la valeur de $2L_s$, qu'il convient de mesurer plusieurs fois pour obtenir une approximation moyenne. Pour Joe0(3.3), on obtient une valeur d'environ 1μs. William0(3.4), donne une valeur d'environ 0.66μs. Avec Fourmi002(3.5) on obtient 1.04μs. Ce sont des valeurs consistantes avec les chiffres fournis par les constructeurs([9])

3.2 Validation du protocole

Pour valider ce concept de protocole, on doit montrer que les diverses optimisations proposées peuvent avoir un impact positif sur le coût global des communications d'une application. Au vu des temps obtenus pour l'overhead en fonction de la taille des messages (3.2), on peut affirmer que $O_s(a+b) < O_s(a)+O_s(b)$, au moins jusqu'à un certain seuil, au delà duquel le coût d'envoi par message devient colossal. Ce seuil dépend du type de carte utilisée(3.6) : alors que les coûts d'envois sont assez similaires pour des petits messages, on

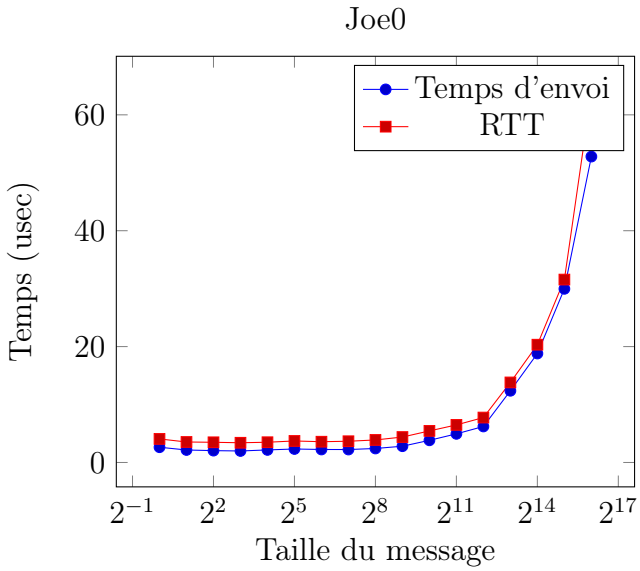

FIGURE 3.3 – Temps d'envoi et de RTT sur Joe0

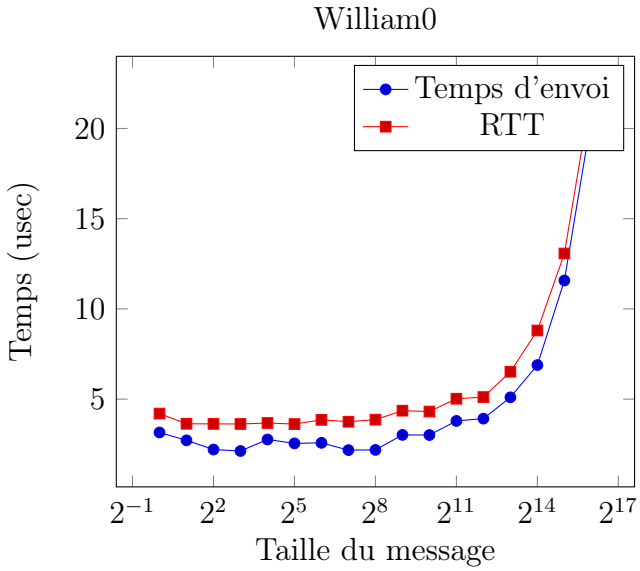

FIGURE 3.4 – Temps d'envoi et de RTT sur William0

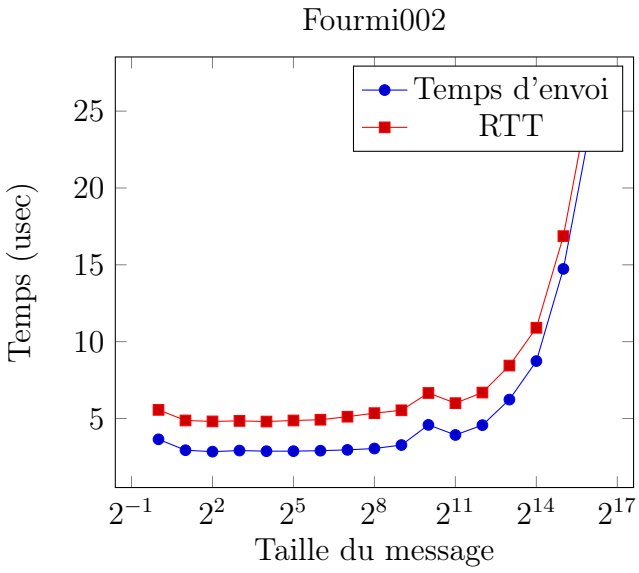

FIGURE 3.5 – Temps d'envoi et de RTT sur Fourmi002

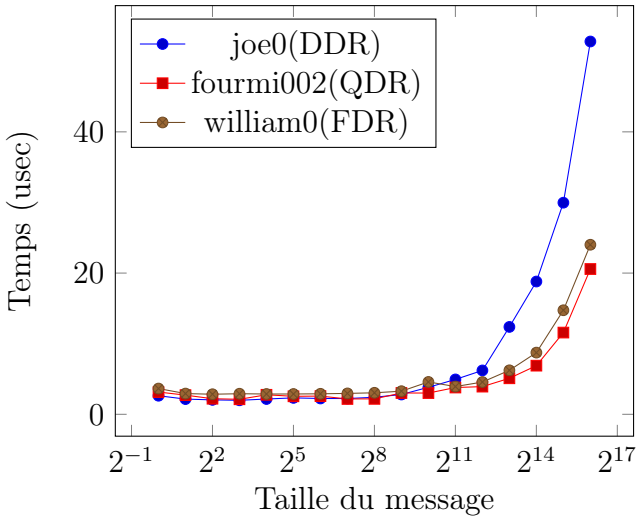

FIGURE 3.6 – Comparaison des temps d'envoi des différentes cartes

voit assez facilement que Joe0, qui utilise une carte DDR, voit son overhead s'envoler bien plus rapidement que les autres, aux alentours de 2^{13} octets. Passé cette valeur, l'envoi d'un message deux fois plus gros demande deux fois plus de temps, l'agrégation n'est donc plus rentable. C'est donc sur des messages de petite et moyenne taille que $O_s(\sum_{i=0}^{n} m_i)) < \sum_{i=0}^{n} O_s(m_i)$. Il faut cependant ne pas oublier le coût induit par les manipulations mémoires qui sont souvent nécessaires pour agréger les messages, qui rendent inutiles l'agrégation des plus petits messages : il est plus élevé que le gain effectué autrement, soit $\sum_{i=0}^{n} O_s(m_i) + O_s(0) + ng - (O_s(\sum_{i=0}^{n} m_i)(h * n) + c)$. Pour ce qui est de l'utilisation du scatter en revanche, les headers étant de la même taille, le gather sera toujours plus rapide à transmettre, car si $O_s(a + b) < O_s(a) + O_s(b)$, alors $O_s(\sum_{i=1}^{n} m_i + hn + c) < O_s(\sum_{i=1}^{n} m_i) + O_s(h * n + c)$. Il faut néanmoins encore une fois considérer les besoins de l'application, et si le surcoût engendré est inférieur à celui induit par la manipulation des données par le récepteur. L'impact de l'utilisation d'un tel protocole serait donc bel et bien positif. En revanche, il faudrait maintenant développer une implémentation complète afin de pouvoir mesurer son influence réelle sur la performance d'utilisation d'un réseau et ainsi le comparer avec différentes bibliothèques de communications.

Conclusion et travaux futurs

3.3 Conclusion

Nous avons montré qu'il était possible d'approcher le RDMA non seulement comme nouvelle manière de propager les messages, mais également comme outil pour accélérer le flux de communication en utilisant les optimisations déjà implémentées sur des communications MPI classiques. A l'aide d'un modèle de représentation des machines HPC, nous nous sommes efforcés de déterminer les effets de divers types d'envoi sur un réseau implémentant le RDMA nativement et enfin définir un concept de protocole permettant l'exploitation des gains envisageables. Afin de valider ce concept et de montrer qu'une mesure simple des différents paramètres de coûts était possible, nous avons mesuré plusieurs caractéristiques du réseau sur divers types de cartes InfiniBand Nous avons ensuite utilisé ces résultats ainsi que les coûts théoriques déterminés précédemment pour mettre en évidence les situations pouvant bénéficier de l'utilisation du protocole. Il apparaît que le concept proposé permet d'améliorer l'utilisation d'un réseau HPC implémentant le RDMA, en revanche il est encore nécessaire d'en concevoir un prototype fonctionnel afin d'avoir une réelle idée de l'impact de ces optimisations dans des cas concrets.

3.4 Travaux futurs

Après avoir implémenté le prototype, il restera encore beaucoup de questions ouvertes. Il faudrait que ce protocole puisse communiquer entre plusieurs hôtes afin qu'il puisse être utilisé dans des situations réalistes, mais également qu'il supporte le multi-threading. Paralléliser certains de ses systèmes pourrait également présenter une opportunité d'amélioration de performance tout en posant un tout nouveau set de problème. Les systèmes eux mêmes peuvent être également complexifiés afin de permettre des économies de messages, pour le contrôle de flux ainsi que pour la transmission des

adresses. L'aspect de la contention du coté du récepteur et de l'exploitation des processeurs manycore représentent aussi une piste de développement prometteuse pour l'avenir, avant de pouvoir éventuellement intégrer le concept à l'intérieur d'une bibliothèque de communication .

Bibliographie

[1] Albert Alexandrov, Mihai F Ionescu, Klaus E Schauser, and Chris Scheiman. Loggp : Incorporating long messages into the logp model for parallel computation. *Journal of parallel and distributed computing*, 44(1) :71–79, 1997.

[2] The ARMCI project website. `http://hpc.pnl.gov/armci/`.

[3] Dan Bonachea. Gasnet specification, v1.1. Technical Report UCB/CSD-02-1207, EECS Department, University of California, Berkeley, Oct 2002.

[4] Élisabeth Brunet, François Trahay, Alexandre Denis, and Raymond Namyst. A sampling-based approach for communication libraries autotuning. In *IEEE International Conference on Cluster Computing*, Austin, United States, September 2011.

[5] Bell C., D. Bonachea, R. Nishtala, and K. Yelick. Optimizing bandwidth limited problems using one-sided communication and overlap. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 10 pp.–, April 2006.

[6] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten Von Eicken. *LogP : Towards a realistic model of parallel computation*, volume 28. ACM, 1993.

[7] D. Bonachea, P. Hargrove, M. Welcome and K. Yelick. GASNet Collectives Poster at SuperComputing 2006, Nov 2006.

[8] Alexandre Denis. A High-Performance Superpipeline Protocol for InfiniBand. In E. Jeannot, R. Namyst, and J. Roman, editors, *Euro-Par 2011*, volume 6853 of *Lecture Notes in Computer Science*, pages 276–287, Bordeaux, France, August 2011. Springer.

[9] InfiniBand performance benchmark on the Mellanox website. `http://www.mellanox.com/page/performance_infiniband`.

[10] Introduction to InfiniBand for End Users. `https://cw.infinibandta.org/document/dl/7268`.

[11] Thilo Kielmann, Henri E Bal, and Kees Verstoep. Fast measurement of logp parameters for message passing platforms. In *Parallel and Distributed Processing*, pages 1176–1183. Springer, 2000.

[12] Jiuxing Liu, Jiesheng Wu, and Dhabaleswar K Panda. High performance rdma-based mpi implementation over infiniband. *International Journal of Parallel Programming*, 32(3) :167–198, 2004.

[13] Mellanox IB-Verbs API Manual. `http://nuweb12.neu.edu/rc/wp-content/uploads/2013/09/MellanoxVerbsAPI.pdf`.

[14] The MVAPICH Project website. `http://mvapich.cse.ohio-state.edu/overview/`.

[15] New Madeleine project website. `http://pm2.gforge.inria.fr/newmadeleine/`.

[16] Jarek Nieplocha, Vinod Tipparaju, Manojkumar Krishnan, and Dhabaleswar K Panda. High performance remote memory access communication : The armci approach. *International Journal of High Performance Computing Applications*, 20(2) :233–253, 2006.

[17] Site de la plateforme d'expérimentation PlaFRIM. `https://plafrim.bordeaux.inria.fr/`.

[18] Renato Recio. A Tutotrial of the RDMA Model. `http://www.hpcwire.com/2006/09/15/a_tutorial_of_the_rdma_model-1/`.