Improving memory consumption and performance scalability of HPC applications with multi-threaded network communications

Sylvain Didelot

▶ **To cite this version:**

Sylvain Didelot. Improving memory consumption and performance scalability of HPC applications with multi-threaded network communications. Distributed, Parallel, and Cluster Computing [cs.DC]. Université de Versailles-Saint Quentin en Yvelines, 2014. English. NNT : 2014VERS0029 . tel-01132316

HAL Id: tel-01132316

https://theses.hal.science/tel-01132316

Submitted on 17 Mar 2015

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

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

UNIVERSITÉ DE VERSAILLES SAINT-QUENTIN-EN-YVELINES

Université de Versailles Saint-Quentin — École Doctorale des Sciences et Technologies de Versailles

Improving memory consumption and performance scalability of HPC applications with multi-threaded network communications

Amélioration de la consommation mémoire et de l'extensibilité des performances des applications HPC par le multi-threading des communications réseau

THÈSE

présentée et soutenue publiquement le 12 Juin 2014

pour l'obtention du

Doctorat de l'université de Versailles Saint-Quentin

(spécialité informatique)

par

Sylvain Didelot

Composition du jury

Directeur de thèse :	William Jalby	- Professeur, Université de Versailles
Président :	Alain Bui	- Professeur, Université de Versailles
Rapporteurs :	Brice Goglin	- Chargé de Recherche, INRIA
	Torsten Hoefler	- Professeur Assistant, ETH Zürich
Examinateurs :	William Jalby	- Professeur, Université de Versailles
	Alain Bui	- Professeur, Université de Versailles
	Marc Pérache	- Ingénieur de Recherche, CEA/DAM
	Patrick Carribault	- Ingénieur de Recherche, CEA/DAM
Invités :	Jean-Pierre Panziera	- Ingénieur de Recherche, Bull SAS

Exascale computing research

Acknowledgments / Remerciements

Je tiens tout d'abord à remercier mon directeur de thèse, William Jalby, pour la confiance et la liberté qu'il m'a accordé tout au long de mes travaux de recherche. Je remercie également Messieurs Marc Pérache et Patrick Carribault pour m'avoir donné la chance de réaliser cette thèse et pour leur suivi durant ces trois années.

J'adresse aussi mes remerciements à mes rapporteurs, Messieurs Torsten Hoefler et Brice Goglin pour la lecture scrupuleuse de mon manuscrit, leurs conseils avisés et les différentes propositions d'amélioration.

Une thèse, c'est aussi un contexte propice à des rencontres exceptionnelles, le bonheur de partager ses travaux avec des collègues passionnés. Merci tout d'abord à Marc T. qui a toujours su se rendre disponible pour des séances de réflexion dans l'*exa-kitchen* et son travail méticuleux de relecture de mon manuscrit. De plus, son optimisme contagieux est le meilleur encouragement qui soit quand tout va de travers. Adepte de l'optimisation au quotidien, on se rappelle certainement tous de ses discussions autour des billets "SNCF" et des engagements 3 ans des opérateurs mobile. Merci également aux collègues et maintenant amis de l'UVSQ: Asma la picoreuse de graines et aficionada compulsive de chocolat noir. Copine de galère pendant la rédaction, je me souviens des moments où nous cherchions à nous motiver mutuellement pour garder notre rythme d'une page de manuscrit par jour. Eric le franc-comtois bourrin qui allume son barbecue à l'essence et au décapeur thermique. Pablo, chercheur obstiné de "Poivre et Sel". Thomas, expert en un peu près tout et testeur assidu de solutions de *backup* de données. Othman, passionné par la photographie et toujours prêt à partager ses talents culinaires autour d'une savoureuse choucroute au vin d'Alsace ou d'un délicieux kouglof.

Je n'oublie évidemment pas mes amis thésards du CEA. Jean-Baptiste, celui qui m'a fait découvrir le rythme endiablé de la musique de "Papa Pingouin". Jean-Yves Vet, l'homme toujours au courant des bons plans pour se faire de l'argent ou en dépenser le moins. Merci enfin à Sebastien qui a réussi à imposer le `git pull -rebase` au lieu du `git merge` ainsi que pour les après-midi enrichies pas des discussions aussi passionnantes les unes que les autres.

Enfin, je remercie tous les membres du laboratoire Exascale et plus particulièrement Bettina, Augustin, Michel, Alexis, Emmanuel, Cédric et Soad. Un grand merci à François Diakhaté et aux équipes du TGCC pour leur support (et la réparation des nœuds que j'ai pu casser) durant l'utilisation du supercalculateur Curie. Sans oublier les étudiants que j'ai eu le plaisir d'encadrer et qui font partie de la génération 2.0 des thésards MPC : Antoine a qui j'ai cédé la paternité du module SHM et Thomas le nouvel expert réseau.

Je ne pourrais terminer cette section sans remercier ma famille. Merci tout d'abord à mon papa et ma maman qui ont toujours cru en moi et soutenu dans mes choix. Ils m'ont appris la *niaque* d'aller jusqu'au bout de ce que j'entreprends sans jamais baisser les bras. Merci à ma sœur Marina, son mari Cédric, leurs enfants et Kloé. Rien de tel qu'une matinée passée à la piscine ou d'un bon repas autour d'une table toujours bien remplie pour oublier les semaines difficiles. Je tenais également à remercier Gilles, Bernadette et Lucie qui continuent à m'accepter comme leur

"futur gendre" alors que leur fille a été kidnappée par un thésard fou pour habiter en région parisienne.

Enfin, je remercie tout particulièrement ma chère et tendre, Aurélie qui a vécu et partagé mon quotidien tourmenté de thésard. Elle a toujours été l'oreille attentive dont j'avais besoin et la voix raisonnée qui a su me réconforter durant les moments de doute. Bien que difficiles, ces années nous ont pourtant rapproché.

À mes parents et Aurélie.

Abstract:

A recent trend in high performance computing shows a rising number of cores per compute node, while the total amount of memory per compute node remains constant. To scale parallel applications on such large machines, one of the major challenges is to keep a low memory consumption. Moreover, the number of compute nodes clearly increases, which implies more memory to manage network connections.

In shared-memory systems, thread programming typically allows a better memory usage because of data sharing. Multi-threaded applications may however run slower than single-threaded executions since the model requires synchronization points. In this thesis, we develop a multi-threaded communication layer over Infiniband which provides both good performance of communications and a low memory consumption. We target scientific applications parallelized using the Message Passing Interface (MPI) standard in pure mode or combined with a shared memory programming model.

This thesis proposes three contributions. Starting with the observation that network endpoints and communication buffers are critical for the scalability of MPI runtimes, the first contribution proposes three approaches to control their usage. We introduce a scalable and fully-connected virtual topology for connection-oriented high-speed networks. This topology provides a support for on-demand connection protocols and reliably transfers short messages between not-connected processes. In the context of multirail configurations, we then detail a runtime technique which reduces the number of network connections compared to regular implementations and thus with similar performance. We finally present a protocol for dynamically resizing network buffers over the RDMA technology. This protocol enlarges buffer regions until it covers the requirement of the MPI application and reduces or destroys them when the free memory becomes short. The second contribution proposes a runtime optimization to enforce the overlap potential of MPI communications without introducing the overhead of a threaded message progression. Used with scientific applications, we show an improvement of communications up to a factor of 2. The third contribution evaluates the performance of several MPI runtimes running a seismic modeling application in a hybrid context. On large computes nodes up to 128 cores, the introduction of OpenMP in the MPI application saves up to 17% of memory. Compared to a master-only approach, the domain decomposition method allows a concurrent participation of all OpenMP threads to MPI communications. In this case, our multi-threaded communication layer brings a 37% performance improvement compared to the full MPI version.

Keywords: high performance computing, multi-threading, high-speed networks, MPI, NUMA

Résumé:

Problématiques

Depuis l'apparition des premiers super-calculateurs dans les années 60 et jusqu'à aujourd'hui, la course effrénée à la puissance de calcul n'a cessé d'offrir aux scientifiques des machines toujours plus puissantes pour leurs simulations numériques. Le terme "super-calculateurs" et plus généralement le Calcul Haute Performance qui est la science relative aux super-calculateurs remontent aux années 1960. Les premières machines étaient alors équipées d'une unique unité de calcul (le CPU) chargée d'exécuter les instructions d'un programme informatique, et ceci de manière séquentielle. Freinés par la loi Moore qui limite l'utilisation à haute fréquence des processeurs, les fabricants de matériel informatique eurent l'idée dans les années 1970 de rassembler plusieurs unités de calcul. Ces nouvelles machines dites "parallèles" introduisirent cependant de nouvelles problématiques à la conception des applications puisque toutes les unités de calcul devaient à présent travailler de concert sur le même problème numérique. Le "calcul parallèle" était né.

Afin de faciliter le développement des applications parallèles, les grappes de calcul proposent une grande variété de modèles de programmation aux développeurs. D'une part, la programmation à base de *threads* optimise l'exécution sur des machines à mémoire partagée. D'autre part, l'interface de programmation par passage de message MPI permet l'échange de données sur des architectures à mémoire distribuée. Dans ce but, l'interface MPI regroupe un ensemble de fonctions qui permettent les communications pair-à-pair et les communications collectives impliquant plusieurs entités communément appelées "tâches MPI".

Le matériel réseau actuel intègre généralement des puces embarquées qui permettent notamment la prise en charge des communications de manière autonome. Tout d'abord et contrairement aux bibliothèques de communications traditionnellement implémentées dans le noyau (i.e., *sockets* UNIX), les nouvelles interfaces réseau fournissent des mécanismes permettant le court-circuitage du système d'exploitation (SE). Par conséquent, les ralentissements liés au SE sont éliminés et les communications ne requièrent plus l'intervention du CPU. Ensuite, ce nouveau matériel offre bien souvent une technologie RDMA (Remote Direct Memory Access) qui permet à un processus de lire directement la mémoire exposée par un autre processus. Cette aptitude aussi connue sous le nom de *zero-copy* empêche aux données d'être recopiées dans le SE, ce qui ralentirait les communications. La plupart des bibliothèques MPI implémente une classe de protocoles spécialisés appelés `rendezvous`. Cependant, puisqu'une tâche MPI émettrice ne connait pas *a priori* l'adresse du message dans la mémoire du destinataire, les protocoles de `rendezvous` requièrent une synchronisation explicite entre les deux tâches ainsi que l'envoi de plusieurs messages de contrôle. Par conséquent et dans le cadre d'une communication asynchrone, le contrôleur réseau et/ou la pile logicielle doivent traiter ces messages de contrôle de manière transparente.

De nos jours, l'augmentation de la puissance de calcul implique l'assemblage d'un grand nombre d'unités de calcul. Celles-ci sont tout d'abord assemblées sous la forme de "nœuds de calcul" à l'intérieur desquels une mémoire unique est partagée.

Les nœuds de calcul sont ensuite interconnectés via un réseau haut-débit, distribuant ainsi la mémoire. Les plus grands systèmes pétaflopiques actuellement en production représentent plusieurs dizaines de milliers de nœuds de calcul et les systèmes exaflopiques à venir (10^{18} opérations à virgule flottante par seconde) annoncent repousser cette limite avec plusieurs centaines de milliers de nœuds de calcul [Ama+09] et plus [Ash+10]. En outre, les mêmes sources prévoient une diminution importante de la mémoire par unité de calcul et cela jusqu'à quelques dizaines de mega-bytes par *thread*, soit moins de dix fois la mémoire actuelle par *thread*. L'un des principaux défis pour les bibliothèques MPI visant ces machines massivement parallèles est l'utilisation raisonnable de la mémoire quel que soit le nombre de cœurs utilisés. En effet, si celles-ci allouent trop de mémoire, la taille du problème numérique que la machine permet de résoudre se verra réduite. À l'intérieur d'un nœud de calcul, les bibliothèques MPI *multi-threadées* sont reconnues comme une solution efficace permettant de diminuer la quantité de mémoire tout en partageant des ressources [PCJ09]. Cependant, avec l'accroissement du nombre de cœurs par nœud de calcul s'en suivent un volume plus large de communications réseau et l'augmentation du nombre de connections réseau. Par conséquent, le nombre de *endpoints* réseau (i.e., structures en mémoire qui définissent une connexion réseau) augmente théoriquement d'autant que le nombre de nœuds de calcul. De plus, bien que les récents contrôleurs réseau proposent des mécanismes *zero-copy*, des *buffers* de communication sont toujours requis pour (1) échanger les messages de contrôle des protocoles `rendezvous` et (2) pour améliorer les performances des messages MPI de petites et moyennes tailles. Par conséquent, un réseau plus large implique l'allocation d'un plus grand nombre de ces *buffers* réseau.

Pour finir, bien que les bibliothèques MPI *multi-threadées* permettent de réduire la mémoire sur les nœuds de calcul, le standard MPI requiert tout de même la duplication de certaines données utilisateur. À l'intérieur d'un nœud, cette duplication inutile peut empêcher une application de passer à l'échelle à cause d'une pénurie de mémoire adressable. Une solution à ce problème est de mélanger MPI avec un modèle de programmation à mémoire partagée et ainsi réduire la consommation mémoire dans son ensemble [RHJ09 ; Jow+09]. À première vue la *programmation hybride* semble séduisante, notamment car le standard MPI propose un mode où plusieurs *threads* sont autorisés à communiquer simultanément via MPI. Cependant, c'est sans compter que ce mode, généralement implémenté par les bibliothèques MPI, montre bien souvent de faibles performances à cause d'une mauvaise gestion de la concurrence entre *threads* [TG07].

Contributions

Cette thèse vise à améliorer l'exécution à large échelle de programmes informatiques parallèles sur les grappes de calcul actuelles et à venir. Dans le but de démontrer la pertinence des contributions présentées, cette thèse s'articule autour du paradigme de programmation par passage de messages MPI (Message Passing Interface) et de l'interface de programmation `verbs` pour réseaux Infiniband.

Nous présentons dans un premier temps les réseaux d'interconnexion communément utilisés dans le contexte du calcul haute performance. Par la suite, nous décrivons les principaux défis qui sont à relever lors de la réalisation d'un moteur d'exécution implémentant MPI. Une fois le contexte posé, cette thèse propose trois contributions.

Partant du constat que le nombre de connexions réseau et le volume des *buffers* associés sont critiques pour la mise à l'échelle en mémoire, la première contribution propose des mécanismes permettant de maîtriser l'utilisation de ces ressources. Nous détaillons la réalisation d'une topologie virtuelle entièrement connectée et chargée d'acheminer de manière sûre des communications entre deux processus non connectés tout en gardant un faible nombre de connexions actives. Nous examinons ensuite une technique de partage de connexions dans un contexte agrégeant plusieurs cartes réseau par nœud de calcul. Cette technique vise à exploiter le potentiel de parallélisme offert par ces configurations tout en minimisant l'impact en mémoire des structures réseau. En outre, nous proposons un protocole permettant d'ajuster dynamiquement la taille et le nombre de *buffers* réseau utilisant la technologie RDMA. Le protocole étend les ressources en mémoire de ces *buffers* jusqu'à les faire correspondre au volume de communication que requiert l'application utilisateur. De plus, ces *buffers* sont détruits dans le cas où la mémoire disponible sur les nœuds de calcul devient faible.

La seconde contribution présente le *Collaborative-Polling*, une optimisation qui renforce le potentiel d'asynchronisme des applications MPI et qui ne nécessite pas l'intervention de *threads* de progression. Une tâche MPI inactive peut venir assister l'avancement des communications d'une seconde tâche bloquée alors dans une phase de calcul.

La troisième contribution évalue l'efficacité des implémentations MPI actuelles dans un contexte de programmation hybride. Après une étude de performance menée sur des micro-*benchmarks*, nous mesurons l'apport du modèle de programmation OpenMP dans une application de modélisation sismique réalisée en MPI. Nous présentons notamment une version hybride par décomposition de domaine qui autorise un accès concurrent de tous les threads OpenMP à la bibliothèque MPI. Nous soulignons ensuite les limitations actuelles du standard MPI pour ce type de programmation et expliquons comment le concept de *endpoints* MPI pourrait lever ces limitations.

Mots-Clés: Calcul haute performance, multi-threading, réseaux haut débit, MPI, NUMA

Contents

Part I

Context

Chapter 1

Introduction

"If you were plowing a field, which would you rather use? Two strong oxen or 1,024 chickens?"

Seymour Cray 1925–1996

Over the last decades, numerical simulation has become an essential tool at the center of academic and industrial research programs all over the world. The constant needs in computational power allow the implementation of even more efficient machines, in order to increase the size and the accuracy of numerical problems. As an example, in 2008, the Blue Brain Project[1] has successfully simulated the neocortical column of a rat (10,000 cells) on a supercomputer. More recently, in 2012, the first-ever simulation of the structuring of the entire observable universe was performed, from the Big Bang to the present time[2]. Without the computational power of *supercomputers*, it is certain that all these scientific advances would have never been achieved.

Before 1960s, computers were equipped with a single Central Processing Unit (CPU) and every program instruction was sequentially executed by the same CPU. With the rise of CPU frequency, heat dissipation rapidly began an issue and computer manufacturers have started to combine several execution units into a single *parallel* machine. These new machines however introduced a new complexity to applications since every execution unit has to efficiently work together on the same problem. The developer must explicitly decompose its application into independent sequences of instructions that can be run in parallel and express the interactions between execution units. The *High Performance Computing* was born.

In order to ease the development of parallel applications, supercomputers now expose a variety of *programming models* (i.e., programming interfaces) to developers. On the one hand, *thread programming* optimizes the execution of applications in a context of shared-memory where a unique memory is shared between all CPUs. In contrast to processes that require complex mechanisms for moving data, all threads have the same address space and directly access the data inside a node. On the other hand, the *Message Passing Interface* (MPI) provides an interface to exchange data across distributed-memory systems where each CPU has its own private memory. In practice, each parallel instance of the user application is executed by one *MPI task* running on one CPU. Communications between MPI tasks are then performed using the MPI interface which exhibits a substantial set of functions including one-to-one and collective operations.

With MPI, the *domain decomposition* method is often used to parallelize numerical applications. The domain is first represented as a grid, generally with two or three dimensions. In each timestep, each cell (i.e., element of the grid) is updated using the value of the neighbor cells in a fixed pattern called the *stencil*. As shown

[1] Project continued with the Human Brain Project: http://www.humanbrainproject.eu

[2] Deus: full universe run : http://www.deus-consortium.org

in figure 1.1, the method then splits the domain into subdomains and maps one subdomain to each MPI task with the purpose of processing each subdomain concurrently. In addition to the inner domain, *ghost cells* are introduced on the edges of each subdomain: they locally replicate the neighbor cells which are required to update the inner domain. At each time step, every task exchanges its ghost zones to its neighborhood. The main challenge here is to balance the workload between computational units.

One MPI communication can be decomposed into *processing time* (i.e., time to process message) and *synchronization time* (i.e., idle time while waiting for the beginning of message transmission). While the processing time grows with the amount of data to transfer, synchronization time mainly depends on the *load-imbalance* between MPI tasks. Indeed, if the sender task takes longer to complete its work than the receiver task, the transmission of ghost cells will be delayed and the receiver will consequently waste CPU cycles inside the MPI library due to synchronization overheads. To address domain-decomposition applications that exhibit a non-uniform distribution of work, methods such as *Adaptive Mesh Refinement* (AMR) provide a solution where the grid resolution is different in some subdomains than others. Furthermore, AMR codes generally provide load-balancing protocols where the grid resolution is dynamically reevaluated during the application lifespan: the work-imbalance needs to be detected and the domain to be repartitioned into equivalent sizes.

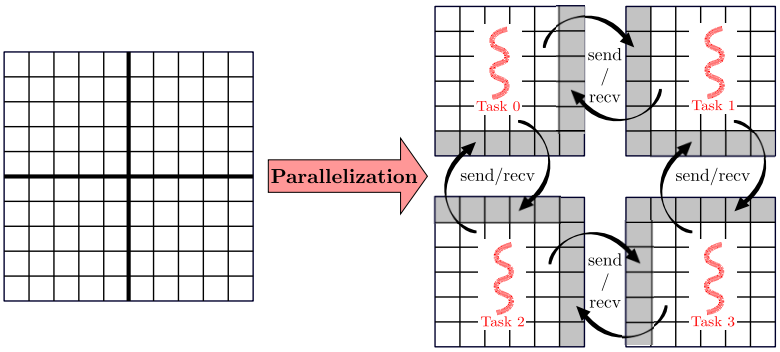


Figure 1.1 – Parallelization using a domain decomposition method

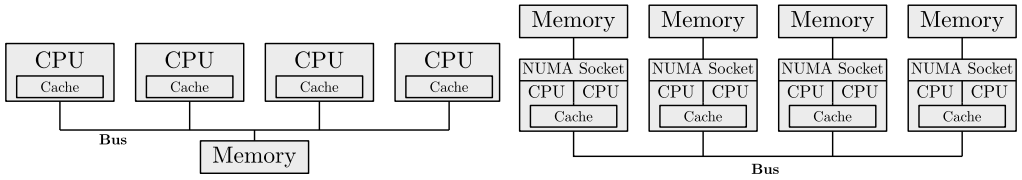
Recent network hardware generally integrates specialized chips that are capable of independently driving complex communication operations. In contrast to regular kernel-based communication libraries such as UNIX sockets, they include mechanisms to minimize the involvement of the operating system (OS) and the network drivers (e.g., *OS-bypass*, *zero-copy*). Consequently, the overhead induced by the OS on the critical path (e.g., memory copies) is reduced and the user applications have a direct access to the network controller. Moreover, the implementation of *OS-bypass* and *zero-copy* mechanisms is facilitated with modern communication protocols like *Remote Direct Memory Accesses* (RDMA) that enable a process to directly access the memory exposed by another process. Most MPI libraries actually implement a specialized class of messaging protocols over zero-copy mechanisms called `rendezvous`. However, since a sender task does not *a priori* know the final address where the data on the receiver side will be copied to, the `rendezvous` protocols require the synchronization of both tasks and the transmission of several control messages. Consequently to an asynchronous progression of MPI communications, the network controller and/or the software stack need to transparently retrieve and

handle control messages related to `rendezvous` protocols.

Nowadays, the increase of computational power typically involves a large number of computational units that are clustered together. Behind the scene of clustering, subsets of computational units are grouped into shared-memory compute nodes, which are then interconnected using high-speed networks. Today's largest systems aggregate tens of thousands compute nodes and upcoming exaflop systems (10^{18} floating point operations per seconds) are expected to push the limit further with severals hundreds of thousands compute nodes [Ama+09] and even more [Ash+10]. Furthermore, the same sources expect a significant diminution of the amount of per-core memory down to a few dozens of mega-bytes per thread, which is less than ten times the current amount of memory per thread. One of the major challenges for MPI implementations targeting such large systems is to keep a decent memory footprint whatever the number of cores. Indeed, if the runtime allocates too much memory, the size of the numerical problem that can be handled on the machine is much smaller. Inside the compute nodes, thread-based MPI runtimes have been recognized as practical solutions for lowering memory usage because they allow to share runtime resources between MPI tasks [PCJ09]. However, the increasing number of compute nodes typically means a larger volume of data exchanged through the network and a higher number of network connections. As a result, the number of *network endpoints* (structures that define a network entry in memory) theoretically increases in the same order of magnitude as the total number of nodes. Furthermore, although recent network controllers propose zero-copy mechanisms, *communication buffers* are still required (1) to exchange synchronization messages related to the `rendezvous` protocols and (2) to improve the performance of short/medium-sized MPI messages. Consequently, a larger network involves the allocation of more communication buffers.

Finally, although thread-based MPI runtimes are able to reduce the amount of memory on compute nodes, the MPI standard still requires the application to replicate some data across MPI tasks (e.g., ghost cells in figure 1.1). Inside a node, these data are unnecessary replicated and can prevent an application from scaling because of a lack of available memory. One solution to tackle this issue is to mix MPI with a *shared-memory programming model* in order to reduce the overall memory consumption [RHJ09; Jow+09]. At first sight, *hybrid programming* looks appropriate to cluster of shared-memory systems, especially as the MPI standard provides a mode where multiple threads can simultaneously call MPI functions. This mode generally implemented inside mainstream MPI runtimes however often exhibits low performance because of a poor management of thread concurrency [TG07].

This chapter aims at presenting the state-of-the-art of high performance computing. Starting with the observation from the Top500 website [Top] that parallel computing is largely controlled by clusters of multi-processor machines, we first detail how such machines are structured. In the second part, we list and characterize programming models commonly used for parallelizing applications on supercomputers. We then introduce the different challenges to overcome when designing a message passing communication layer targeting current and upcoming clusters of massively multi-core machines. Finally, we detail the contributions of the thesis and present document organization.



(a) SMP **(b)** NUMA

Figure 1.2 – Example of SMP (a) and NUMA (b) architectures

1.1 Overview of Supercomputer Architecture

As predicted by Gordon Moore in 1965 [Moo+65], the processors seem to follow the general rule where the number of transistors on integrated circuits doubles every 18 months. Since 1965 and timelessly over the last decades, the Moore's Law has been continuously applying. With the reduction in component size, CPUs began to integrate powerful optimizations to accelerate the flow of instructions by tracking CPU stalls while waiting for a resource. Among them, (1) pipelining consists in decomposing an instruction into a set of individual data processing elements in order to overlap their execution and (2) out-of-order execution can reorder instructions according to the availability of data and operands instead of the original order of the instructions. Additionally, vector instructions now allow operations to operate on large vectors (up to 512 bits) with a minimum of CPU cycles.

Due to energy issues with high frequency, dissipating heat generated by processors rapidly became a real challenge. To face it, the idea of multiplying the number of computation units emerged in mid 1970s. One representative examples of this trend is the Curie supercomputer composed of 77,184 general purpose execution units which can work together on the same numerical problem. Twice a year, the Top500 project [Top] ranks and details the 500 most powerfull computers in the world. According to the June 2013 Top500 list summarized in table 1.1, *Curie Thin nodes* is the fifth European supercomputer and world-ranked at the 15^{th} position with a computational power of 1.359 PetaFLOPS (1.359×10^{15} FLoating-point Operations Per Second).

Rank	**System**	**Name**	**Site**	**# Cores**	R_{max} **(GFlop/s)**
1	TH-IVB-FEP Cluster	Tianhe-2	National University of Defense Technology	3,120,000	33.863
2	Cray XK7	Titan	Oak Ridge National Laboratory	560,640	17.590
3	10510.0IBM BlueGene/Q	Sequoia	Lawrence Livermore National Laboratory	1,572,864	17.173
4	Fujitsu SPARC64	K computer	RIKEN Advanced Institute for Computational Science	705,024	10.510
5	IBM BlueGene/Q	Mira	Argonne National Laboratory	786,432	8.587
...	...	...	...	...	...
15	Bullx B510	Curie Thin Nodes	CEA/TGCC-GENCI	77,184	1.359

Table 1.1 – Top 5 supercomputers and *Curie Thin nodes* ranked at the 15^{th} position. R_{max} is the maximal performance achieved using the High Performance LINPACK (HPL [Don88]). List extracted from the June 2013 Top500 list

Clustering all processing units into the same chip is however impractical, particularly because keeping the coherency of a unique memory with so many units

would result in poor performance. Instead, Curie's processing units are grouped into compute nodes that share the same memory address space and no coherency is ensured between compute nodes. Inside the compute nodes, SMP systems (see figure 1.2(a)) began to be an important bottleneck with a large number of processors since a single bus accesses a unique memory. To address this issue, Curie enables a non-uniform memory access (NUMA architecture depicted in figure 1.2(b)) and provides a separate memory for each NUMA socket (i.e., group of processes). Every processor however keeps a global vision of the main memory. To handle the case of multiple processors requiring the same data, NUMA systems provide mechanisms to move data between memory banks. Because this operation is costly and increases the time to access a piece of data, the performance gain that an application can expect from NUMA architectures largely depends on data locality. Furthermore, to reduce the distance between the main memory and the processor and enhance memory accesses, a few amount of memory is now engraved directly on the die and commonly called *memory cache*. Memory cache is smaller than the main memory (20 MB on Curie Thin nodes) but because it is integrated on the same chip, it allows a fast access to the cached data. The goal here is to leverage spatial and temporal locality of applications to store least recently used data and reuse those data in the near future. Since the introduction of specialized hardware inside HPC, the architecture of compute nodes is even more complex. In the form of Graphical Processing Units (GPU) and coprocessors (e.g., Intel MIC), they are often connected using PCI Express or equivalent buses and provide a high parallelism potential.

For the purpose of interconnecting compute nodes together, Curie relies on Infiniband, a high-speed and low latency network. To carry the communication volume of the 4,824 compute nodes that compose Curie, the network is hierarchically structured following a 2-level Fat Tree topology [Lei85]. According to the location of the destination node to reach, communications consequently involve a variable number of network switches that affect both latency and bandwidth. Furthermore, in some cases, one network interface may be insufficient for addressing the rise of shared-memory parallelism. With 128 cores per compute node, the Curie Fat nodes supercomputer is an example of this trend since every compute node is equipped with 4 distinct network interfaces.

In a nutshell, one of the most key performance factor on Curie – and more broadly on cluster-based supercomputers – is *data-movement* that is required for exchanging information from one execution unit to another. On the one hand, compute nodes expose a highly-hierarchical memory where data-locality is crucial for rapidly accessing data in order to prevent processor stalls. On the other hand, the inter-node communication networks ensure the transmission between distributed memory address space. According to the Top500 project, two metrics in PetaFLOPS are used to rank supercomputers: the R_{Peak} and the R_{Max} that respectively inform on the cumulative peak performance of every processing unit that the system could theoretically achieve and on the highest score measured using the High Performance LINPACK (HPL [Don88]). Additionally, the ratio between the R_{Max} and the R_{Peak} is also an important metric since it reflects the *efficiency* of the system. For instance on Curie Thin nodes, the maximum computational efficiency achieved on HLP is 81%. In this case, 19% of the computational power is not exploited by the application.

1.2 Programming Models for HPC

Over the years, the architecture of supercomputers has become increasingly complex. Whether inside a compute node or across interconnection networks, the large spectrum of different hardware makes difficult the development of efficient applications. In addition, it would be unrealistic to request such a level of programming expertise to every application developer, especially as scientists from other fields do not necessarily have an advanced background in computer science.

Because supercomputers should be accessible to everyone, they must provide a software stack that abstracts low-level complex operations and exhibits clear and portable interfaces to application developers. Some tools already exist to automatically parallelize a sequential application intended for unicore architectures [Kim+10]. However, the quality of the code produced is occasionally suboptimal due to the need of a complex program analysis and because some factors are only available during execution. In fact, it is often the responsibility of the developer to specify how the application will be executed in parallel. For that purpose, many programming models are available, each with its own special features. In practical terms, developers divide the workload of their application in order to feed each execution unit with computations. Then, according to the chosen programming model, the interactions between running instances of the application are explicitly managed by the programmer.

Since there are plenty of programming models targeting the development of parallel applications, the following section only focuses on the three predominant models. The thread model is intended to leverage the shared memory context of multi-core architectures. The message passing paradigm and the partitioned global address space model are designed to distributed memory systems where each task has only access to its private memory, inaccessible to other tasks.

1.2.1 Shared-Memory Systems

The most explicit way for expressing shared memory parallelism consists in directly manipulating threads, or *light-weight processes*, which share the same address space of the underlying UNIX process. The PThread interface (for POSIX Threads) allows for developing portable multi-threaded applications on systems which are compliant with the POSIX standards. The programmer has a total control on thread creation/destruction and, in particular, it must explicitly manage synchronizations to avoid race conditions and keep memory consistent. A large set of functions is available for this usage among which mutexes guarantee an exclusive access to a shared resource.

To keep an independent execution flow, each thread has a private execution context (call stack, processor registers such as Instruction Pointer). It allows the Operating System (OS) scheduler to put a thread to sleep and switch to another thread. This context switching may for example occur when a thread enters a blocking call or reaches the end of its quantum of time allowed by a preemptive scheduling. Despite the light-weight aspect of threads, creating them and managing context switches are not trivial operations. Furthermore, the overhead due to thread management is actually magnified by the fact that thread scheduling is often achieved by the OS.

The PThread standard is however barely used because taking full advantages

of this model is painful on large applications. Parallel programmers often resort to higher level interfaces like OpenMP [Ope13] which abstracts low-level thread operations such as thread managing (e.g., create, join) and provide a user-friendly interface for sharing work (e.g., distribute `for`-loop iterations among multiple threads).

Finally, task parallel programming models like Cilk [Blu+96] or Intel TBB [Intd] offer a way for application programmers to expose the parallelism by identifying the operations that can concurrently be executed in parallel. The runtime then decides during execution how to distribute the work to the execution units. These programming models are well suited for nested parallelism in recursive like in recursive divide-and-conquer algorithms.

1.2.2 Distributed Memory Systems

Partitioned Global Address Space (PGAS [EGS07]) is a parallel programming model that assumes a user-level global address space that is logically partitioned such that a portion of it is local to each process. Compared to the explicit message passing interfaces such as the Message Passing Interface detailed in the next section, PGAS languages allow any process to have a direct access to the shared data. Furthermore, since PGAS languages exclusively communicate using one-sided communications, they are well suited for networks that support a direct access to the memory of remote processes.

However, because the compiler is responsible for the code generation related to communications, the lack of efficient communication optimizations can result in poor performance. This statement is especially true for applications that use fine-grain communications where accessing a remote memory region is orders of magnitude slower compared to operations on the local memory [CIY05]. The two main PGAS languages are Co-array Fortran (CAF [NR98]) and Unified Parallel C (UPC [EgCD03]).

In the distributed memory programming model, the message passing paradigm is undoubtedly the most prevalent model for developing applications aimed to be run on a large number of cores. In the message-passing model, processes executing in parallel have separate address spaces and communications occur when a portion of one process address space is copied into another process address space.

1.2.3 The Message Passing Interface

Historically, the message-passing paradigm has been popularized by PVM (Parallel Virtual Machine [Sun90]), an open-source and portable project publicly released in 1989. Before PVM, application developers had to settle for proprietary communication libraries provided by parallel computer vendors.

Thereafter, the Message Passing Interface (MPI [MPI93]) replaced PVM. MPI is a message-passing standard proposed in 1993 by the MPI forum and now available for C, C++ and Fortran. The standard provides a set of portable messaging interfaces that could be implemented on any machine, independently of the underlying node and network hardware. Nowadays, MPI is the *de-facto* programming model for developing parallel applications. A large number of parallel codes have been ported to MPI and some libraries natively support MPI such as Intel MKL [Inta] or ScaLAPACK [Tre89]. The key to success has probably been a high-portability of code and performance. At first glance dedicated to distributed memory systems, MPI nevertheless provides good performance on shared memory systems.

Since the first revision of the standard, MPI provides a two-sided point-to-point communication model where communicating pairs of processes call *Send* and *Recv* functions to transmit a message. In addition, the standard exposes a variety of powerful and efficient collective operations.

1.2.3.1 Point-to-point Communications

The basic communication model of MPI is point-to-point communication, in which a piece of data is exchanged between two tasks. Point-to-point send routines specify the address of the data, its size, the destination and a message identifier as well. Conversely, receive routines specify the address of the remote buffer, its size, the emitter and the message identifier expected. Once a message is received, the MPI library walks through the list of pending receive requests and determines which receive buffer matches the description of the incoming message. Additionally, MPI provides two modes for point-to-point communications. The *blocking* mode blocks the MPI task until it is safe to re-use the communication buffer. The *non-blocking* alternative returns control to the application as soon as the MPI request is internally registered, leaving the opportunity for the messaging library to progress the communication in background. Both sender and receiver then wait or poll for the completion of operation by calling an appropriate library function such as `MPI_Wait` or `MPI_Test`.

1.2.3.2 Collective Communications

Another important concept is blocking collective communications that involves a pre-defined group of MPI tasks called *communicator*. Collective communications are categorized into three groups:

1. **Data movement operations** are used to rearrange data among MPI tasks. They include the broadcast operation and many elaborate scattering and gathering operations.

2. **Collective computation operations** such as minimum, maximum, sum and logical OR using a reduce operation.

3. **Synchronization operations** such as barriers that block the calling task until all processes in the communicator have reached this routine.

1.2.3.3 From MPI-1 to MPI-3

Since MPI-2, the standard supports I/O and one-sided messaging where processes perform remote accesses to exposed regions of memory. In this model, MPI tasks collectively expose a window of memory and remote tasks may passively or actively access data. Concerning the active target mode, the remote task explicitly synchronizes its window between accesses. In the passive target mode, the origin task may update the target window without involving the remote task.

MPI is actively developed and the third version of the standard has been released in 2012. Among the available enhancements, the standard now introduces the possibility for collective communications to be non-blocking and to progress asynchronously. Additionally, the standard extends the one-sided communication operations of MPI-2.

1.2.3.4 Message Passing Interface Implementations

During the last twenty years, a large number of MPI libraries have emerged. On the one hand, several open-source MPI libraries such as Open MPI [GWS06], MPICH [Gro+96] (and its successor MPICH2 [Labb]) are freely available and efficiently leverage the widespread underlying hardware, whether the MPI tasks are communicating on the same compute node or on two different compute nodes. Nowadays, MPICH and Open MPI are certainly the predominant MPI runtimes and numerous MPI runtimes derivate their codes from these two projects. Moreover, these runtimes are recognized as stable (because they are intensively evaluated by a large community) and feature-complete with a support of the latest MPI standard. On the other hand, some libraries provide an optimized support for specific hardware and architectures. MVAPICH2 [Hua+06], runtime derived from MPICH2 and developed at the Ohio State University provides an inter-node communication layer tuned for Infiniband, 10GigE/iWARP and RoCE (RDMA over Converged Ethernet) networks. MPICH-GM and MPICH-MX [Inc] are both MPI implementations on top of Myricom GM and MX interconnects. It is now commonplace that parallel computer vendors provide their own MPI runtimes, tuned for the underlying hardware they set up. Intel MPI [Intc], BullxMPI [Bul], IBM Platform MPI [IBM] and Cray MPI are some examples. Finally, a few projects aim at including new software features to MPI. Fault Tolerant MPI (FT-MPI [FD00]) and MPICH-V [Bos+] are both projects focusing on reliability of MPI applications. FT-MPI provides a process-level fault tolerance at the MPI API level. In case of failure, a notification is transmitted to the application. The application can make the decision to abort the job, respawn the dead process, shrink/resize the application to remove the missing processes or create holes in the communicator. During job execution, MPICH-V processes periodically emit an "alive" message to a "dispatcher" which monitors the communication. If it detects a potential failure, the dispatcher launches a new instance of the dead process. If a failure happens during a communication, both FT-MPI and MPICH-V runtimes ensure that in-flight messages will be either canceled or received after the process restart.

The point of mutual interest of previously cited MPI runtimes is that they are all process-based, meaning that every MPI task is a UNIX process. Because the MPI standard does not restrict MPI tasks to be processes, a few number of MPI libraries have early emphasized the potential of encapsulating MPI tasks into threads. The main reason concerns performance aspects as thread-based MPI runtimes only require one memory copy for intra-node communications. A large number of thread-based MPI runtimes has thus emerged among them AMPI [HLK04], AzequiaMPI [RGM11], FG-MPI [KW12], MPC [PCJ09], TOMPI [Dem97], TMPI [TY01], USFMPI. More recently, Friedley *et al.* proposed Hybrid MPI (HMPI) [Fri+13], a hybrid MPI runtime that proposes MPI tasks as processes but enables one memory copy mechanism for intra-node communications. Finally, several process-based MPI runtimes now rely on kernel modules (KNEM [GM12], LiMIC2 [Jin+07]) to mimic the one-copy mechanism of thread-based MPI runtimes.

In the next paragraph, we detail MPC, a framework that implements a thread-based MPI runtime.

MPC: The MultiProcessor Computing Framework

The MPC framework depicted in figure 1.3 aims at improving the scalability and performance of applications running on large clusters of multi-processor/multi-core NUMA nodes[3]. It provides a unified runtime and exposes to the user its own implementation of the POSIX threads, OpenMP 2.5 and MPI 1.3. MPC relies on its own lightweight two-level and non-preemptive MxN thread scheduler where N user-level threads are scheduled on top of M kernel threads (depicted as *Virtual Processors* in the figure). This processor virtualization brings a total control over scheduling and a fast context switching between threads. This point is attractive to efficiently oversubscribe CPU cores with several MPI tasks.

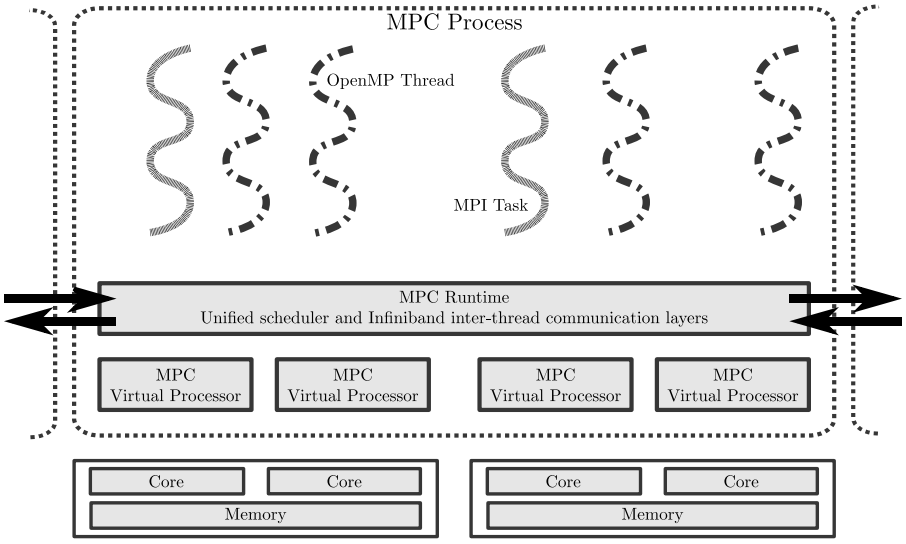


Figure 1.3 – MPC runtime overview

More precisely on the MPI implementation, MPC is a thread-based runtime where MPI tasks are encapsulated inside POSIX threads. Additionally to a full support of the standard, MPC provides the `MPI_THREAD_MULTIPLE` level of thread-safety from MPI 2.0. In a few words, this level allows multiple threads to simultaneously call MPI functions. To deal with global variables and ensure the correctness of MPI applications on top of a thread-based MPI runtime, MPC includes a compiler based on the GNU Computer Collection (GCC) that enables auto-privatization of end-user global variables to Thread Local Storage (TLS) variables. MPC also extends the regular TLS mechanism to nested hybrid MPI/OpenMP codes where the user may choose the level at which a variable must be shared or privatized [CPJ11]. As an example, a variable can be privatized at an MPI level and shared across the OpenMP tasks owning the same MPI task.

At the time of this writing, the inter-process communication layer of MPC supports IP-based networks and accesses high-speed networks over Infiniband.

1.2.4 Discussion

The message passing programming model can be considered as a low-level approach but portable since users keep a fine-grain control on how the work is distributed and what are the interactions between computing entities. Since each entity works

[3]The MPC framework is freely available at `http://mpc.sourceforge.net`

on its own private data, the model naturally ensures locality of memory accesses. Furthermore, there is an implementation of the MPI standard for pretty much all systems, making MPI portable in terms of performance and by far the most adopted standard for parallel applications.

1.3 MPI Challenges

In the following section we describe what we believe are the six more important challenges an MPI runtime shall overcome for current and upcoming large-scale parallel systems: (1) the high performance of communications, (2) the runtime scalability and reliability, (3) the independent progression of messages, (4) the runtime memory consumption, (5) the efficient support of hybrid programming and (6) the locality of data.

1.3.1 High Performance of Communications

MPI runtimes are a predominant link which connects the user application to supercomputers: they supervise process spawning/finalization and manage communications between tasks. One of the main relevant criteria for MPI runtimes is the high-performance of communications while most efficiently utilizing the underlying hardware. With the evolution of hardware, MPI runtimes must continuously be reconsidered for maximizing the performance potential of the underlying system. As an example, mainstream runtimes like MPICH2 provide highly optimized intra-socket intra-node MPI communications [BMG06; GM12]. On the other side, inter-node communications of MVAPICH2 over Mellanox ConnectX-3 FDR Infiniband HCAs deliver latencies close to 1 microseconds and unidirectional bandwidths up to 6,000 MB/s on the same architecture[4].

Designing an efficient communication layer for MPI runtime usually requires the consideration of two decisive points for performance. First, the runtime should provide a support for recent network protocols in order to improve network communications of end-user applications. On recent NICs, RDMA is by far the protocol that brings both the lower latency and the fastest communications. Second, the MPI library should optimize the critical path and hunt idle CPU cycles. With thread-based MPI runtimes, this implies, for example, to design efficient algorithms in order to minimize synchronizations due to resource sharing.

1.3.2 Scalability and Reliability

HPC trends over the last 20 years show a continuously rise in the number of cores. According to the June 2013 Top500 project [Top], with 3,120,000 cores, the Tianhe-2 supercomputer from National University of Defense Technology in China is the first system with three million or more cores.

To run MPI applications on such large machines, the runtime should provide equivalent performance of communications regardless of the number of cores. Parallel applications are however subject to phenomena that are insignificant or absent with a few number of tasks but that become critical for scalability on one million cores.

[4] Benchmark results extracted from MVAPICH2's wesite and available at http://mvapich.cse.ohio-state.edu/

With the high level of nowadays supercomputer hierarchy, one common optimization to provide efficient communications is to design algorithms that fit the underlying hardware layout. First, runtimes should implement NUMA-aware algorithms to limit the saturation of NUMA interconnects in NUMA architectures. Second, algorithms such as collective communications should carefully consider the network topologies (e.g., multi-dimensional torus, tree-like topologies, hypercubes) of current and upcoming machines.

Moreover, large-scale applications are more exposed to hardware failures (e.g., dead link, NIC out of service) and software anomalies (e.g., process crash, resource exhaustion). New techniques should consequently be designed for dynamically diagnosing the source of the error and restoring lost services.

1.3.3 Independent Message Progression

Nowadays trends in high-speed network cards are to improve raw performance while offloading communication operations to the NIC. One challenge for optimizing MPI runtimes is to provide independent progression of communications, which aims at (1) minimizing the time inside the communication library and (2) releasing the host CPU to the application.

Independent progression of communications actually focuses on two aspects. First, runtimes should provide an efficient support of hardware capabilities related to independent message progression. This approach however leads to expensive hardware since the NIC should integrate a programmable micro-controller or should fully support the MPI standard in hardware. Second, the software stack should provide some mechanisms for asynchronously progressing MPI messages. One common strategy to achieve this goal is to rely on progression threads, which however have two severe implications. If the progression and the computation threads share the same core, the high number of context-switches may generate an overhead. Otherwise, if some cores are dedicated to progression threads, less cores are available for computations.

1.3.4 Memory Consumption

The most up-to-date studies on plausible exascale system architectures highlight a large growth in CPU numbers with a factor of more than 2,000 compared to Tianhe-2, the current world's fastest supercomputer [Ama+09; Ash+10]. On the contrary, the same sources expect a significant drop to less than 50 MB of memory per core where 1 GB is commonplace in today's supercomputers like Curie. Furthermore, one promising hardware candidate for exascale machines is the Intel MIC architecture depicted in figure 1.4. The 5100 family provides a single chip up to 1 teraflops double-precision performance and integrates 60 x86 cores for a total of 240 threads and 8 GB of memory [Inte]. As a result, this architecture represents less than 33 MB of memory per thread.

To run MPI applications on exascale machines, one of the most challenging points to consider for scalability is memory footprint at all levels, from the runtime to the end-user code [Tha+10]. As regards MPI runtimes, efforts should address structures that linearly increase with the number of tasks inside the communication. Alternatively, the behavior of the runtime should adapt to the end-user application and dynamically adjust its memory consumption according to the free memory. If

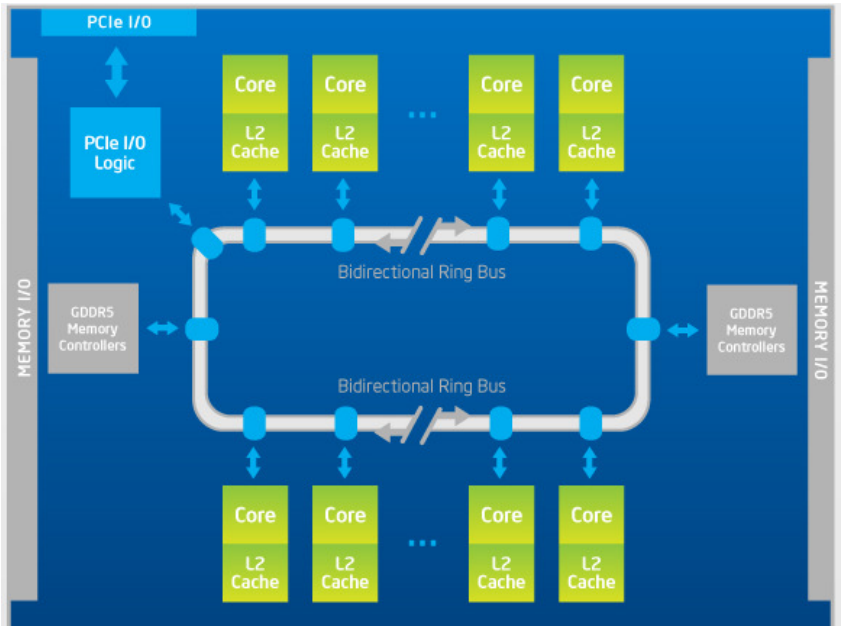


Figure 1.4 – Intel Many Integrated Core (MIC) architecture block diagram (courtesy of Intel)

a compute node runs out of memory, users would certainly better admit a slower application than a job that fails and needs to be restarted from the beginning.

With upcoming memory restrictions of future exascale machines, thread-based MPI runtimes are a relevant solution because they provide an efficient way to decrease the overall memory footprint. First, threaded MPI runtimes allows for sharing common structures across MPI tasks such as buffers and inter-node communication endpoints. Most MPI runtimes implement the `rendezvous` protocols over *bufferless* zero-copy transfers. However, since a sender MPI task does not *a priori* know the final address at the receiver side, `rendezvous` protocols require the synchronization of both tasks and the transmission of several control messages. As a result of these control messages, the communication latency over these protocols is high. Consequently, short and medium-sized messages are generally transmitted using the regular Send/Receive semantics which involves network buffers. Second, threaded MPI tasks do no require extra-memory for intra-node communications while process-based MPI runtimes usually allocate extra buffers inside a shared memory segment [BMG06]. Recently, HMPI [Fri+13] proposed a mechanism to enable zero-copy for intra-node messages inside a process-based MPI runtime. Although this design decreases the number of shared-memory buffers, at the time of writing, it does not allow network resources to be shared among MPI tasks.

Designing MPI tasks as threads is however not a silver bullet to low-memory systems. Because sharing resources in a multi-threaded environment requires synchronizations, thread-based MPI runtimes make developments more complex and often exhibit performance overheads.

1.3.5 Hybrid Programming

Exascale machines are expected to have many more cores per node than today, but the amount of per-core memory is likely to decrease. Applications want however to address more and more memory and the amount of memory available to an MPI task

is insufficient to solve emerging problems. One answer to that concern is to combine message-passing and shared-memory programming models, often referred to as hybrid programming. With this approach, MPI is dedicated to move data between different compute nodes and some shared-memory model is used for parallelizing data within the node. Candidates for X are the following:

- OpenMP, Cilk and TBB [Intd]
- PGAS languages such as UPC or CoArray Fortran. Some research groups have started to investigate advantages of combining MPI+UPC [Din+10]. MVAPICH2 has been extended to unfify both UPC and MPI runtimes [Jos+10].
- CUDA [Nvi08]/OpenCL [Mun+09]. In that regard, a version of MVAPICH2 (MVAPICH2-GPU) is available and focuses on Hybrid MPI+CUDA[Wan+11]

Since the first revision of the standard, MPI makes hybrid programming possible and defines four levels of thread-safety for multi-threaded applications. This mechanism allows the runtime to avoid more thread safety than the user actually needs.

The highest level of thread-safety is `MPI_THREAD_MULTIPLE`. With this level, multiple threads may concurrently call MPI without any restriction, which is particularly useful for hybrid codes. Although it is often implemented, the related work has proven the poor efficiency of `MPI_THREAD_MULTIPLE` inside regular MPI runtimes [TG07]. Indeed, multiple threads accessing the runtime at the same time requires the implementation to use locks which turn out to be expensive and complicate developments. Aware of this concern, application developers often avoid multi-threaded accesses to MPI and rather rely on a master-only approach: MPI is initialized without thread support and communications are all handled by the same thread. This solution however increases the sequential portion of the application and, according to the Amdahl's Law, considerably reduces the application parallelism.

To conclude on hybrid programming, one of the most anticipated challenges for MPI runtimes is certainly the ability to allow several threads to efficiently share internals from the runtime.

1.3.6 Data Locality

Today's compute node architectures are cache coherent NUMA where the memory is hierarchically organized. As an example of this trend, systems with the proprietary Bull Coherent Switch (BCS[5]) exhibit two different levels of NUMAness for a total of 16 NUMA nodes.

Figure 1.5 shows the consequence of non-uniform memory accesses over the memory bandwidth considering a 128 MB memory copy on a 128-core architecture implementing the BCS. As we can see, the NUMA interconnect significantly affects the memory accesses and communication passing by the BCS are slowed down by a factor of 2. NUMA effects are however not ineluctable and can be mitigated by maximizing local memory accesses and avoiding latency and bandwidth penalties induced by remote accesses. In addition, since the BCS is shared across the whole

[5] see http://www.bull.com

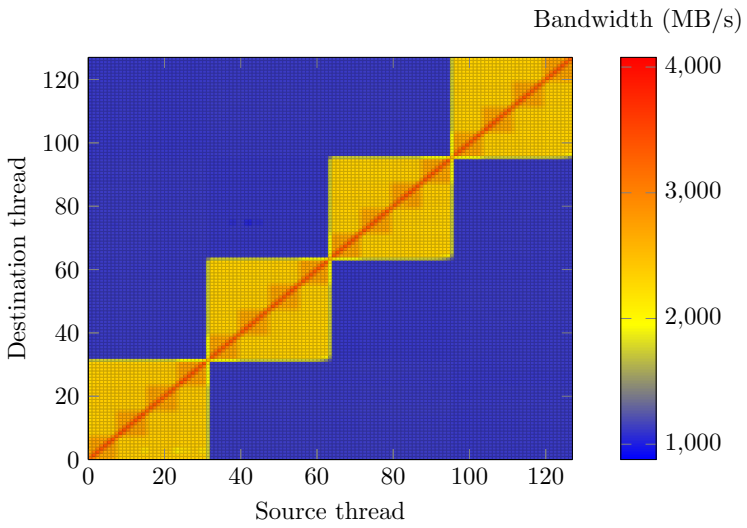


Figure 1.5 – NUMA effects between processes on an architecture implementing the BCS. Efficiency of a memory copy (128 MB) in MB/s according to the memory affinity between the 128 physical cores

compute node, a large memory traffic may cause a bottleneck. Even worse, Infiniband communications may also be subjected to this bottleneck effect since they may pass through the BCS as well.

These locality concerns involve just as much application developers (i.e., hybrid programming) than runtime designers. Because the MPI standard has been designed for distributed memory architectures, it slightly suffers from data locality. Implementing MPI tasks as threads however requires to carefully considering the locality of the runtime internals in order to minimize non-local data accesses.

1.4 Dissertation Contributions

The following thesis investigates the memory scalability issues and performance concerns in high-speed networks for the Message Passing Interface (MPI) inside a multi-threaded context.

The first contribution targets the memory usage of network endpoints and communication buffers in MPI runtimes. Starting with the observation that these resources are critical for achieving large-scale executions of parallel applications, we present three new approaches to reduce their usage in memory.

- We expose a **scalable and fully-connected virtual topology for connection-oriented high-speed networks**. This work aims at providing a convenient, reliable and high-speed transportation protocol to exchange data between processes without connecting them and for any underlying network.

- We design a novel approach for leveraging **multi-rail configurations in a multi-threaded context** with network endpoint sharing. The method is new since it provides similar results as the related work but reduces the number of network structures required for communicating.

- We propose a **protocol for dynamically resizing network buffers over the RDMA technology**. Network buffers over RDMA is a relevant solution for optimizing latency and bandwidth of MPI messages. This protocol however

induces the allocation of a large amount of memory, which would limit their usage. The contribution develops an approach for dynamically readjusting the amount of RDMA buffers allocated by the runtime according to (1) the volume of data that the MPI application communicates and (2) the free memory in the compute nodes.

The second contribution named Collaborative-Polling allows an **efficient auto-adaptive overlapping of communications by computations**. This runtime approach enforces the overlap potential of MPI applications without introducing the overhead of a state-of-the-art threaded message progression. While waiting for messages, the idle MPI tasks can progress and handle outstanding messages for other MPI tasks running on the same compute node. This contribution led to one publication in a scientific conference [Did+12] and another in a journal [Did+13].

The third contribution evaluates the **performance of MPI runtimes in the context of hybrid programming**. Using micro-benchmarks, the contribution highlights an overhead inside mainstream MPI runtimes due to multi-threaded accesses to the runtime's internals. Focusing on a modelling seismic application parallelized with MPI, the contribution evaluates the introduction of OpenMP to the application. Moreover, it presents a domain decomposition method which allows a concurrent participation of all OpenMP threads to MPI communications. With this method, experiments demonstrate the relevance of using an MPI runtime that efficiently supports multi-threading. Furthermore, this thesis highlights the limitations induced by the MPI standard on hybrid applications and discusses how they could be addressed with the concept of MPI endpoints.

Finally, the last contribution involves the **conception of a multi-threaded communication layer over Infiniband** that integrates the functionalities previously mentioned into a mainstream runtime interfacing several programming models.

1.5 Document Organization

Chapter 2 describes past and current high-speed networks that compose shared-memory clusters. In particular, it presents the `verbs` Application Programming Interface (API) that is used to implement communication layers over RDMA-capable networks like Infiniband. With the observation that network endpoints and buffers are two dominant factors for memory scalability of communication libraries, Chapter 3 proposes three techniques for controlling their usage. In Chapter 4, the thesis investigates regular threaded message progression and introduces a runtime optimization that enforces the asynchronousity of MPI communications. Chapter 5 then explores hybrid programming and demonstrates the needs of a thread-safe MPI runtime through a scientific application. It also shows the limitations of the current MPI standard for supporting this model. Finally, Chapter 6 concludes on the work achieved in this thesis and proposes some outlooks that emerge from this work.

Chapter 2

Interconnection Networks for High Performance Computing

"Never underestimate the bandwidth of a station wagon full of tapes hurtling down the highway."

Andrew S. Tanenbaum,
Computer Networks, 4th ed., p. 91

2.1 Introduction to High-Speed Networks

Since high-speed networks are governed by electrical limitations more severe than the scaling performance of semiconductors, they need to radically change every few years in order to maintain a balance with the CPU performance. From the beginning of clusters made of commodity computers to nowadays HPC centers, networks have evolved, not only in terms of performance, but also in terms of capabilities natively supported by the hardware.

In the following section, we present high-speed networks for interconnecting HPC clusters. We first describe the overhead of regular communication libraries implemented in the kernel and detail some capabilities of modern networks for improving performance. We then describe several low-level interfaces and high-level communication libraries commonly used to program interconnection networks. More precisely, we emphasize the Infiniband network and introduce the mechanisms provided by the `verbs` Application Programming Interface (API) from OpenFabrics. Finally, we introduce the experimental platforms that we used to evaluate the contributions.

2.1.1 Kernel Level Messaging Libraries

Traditional communication libraries reside in the kernel space. In other ways, to access the network structures, the kernel exposes a regular client/server socket API to end-applications.

TCP/IP is one example of a kernel level protocol. Known as the Internet Protocol, it was initiated in the 70's and it is the most widespread communication protocol for interconnecting computers on a network. It uses four independent stacked layers of general functionalities that provide a high portability from both a hardware and software perspective. While TCP/IP perfectly fits the requirements of an heterogeneous and decentralized network like Internet, it is not suitable for HPC mainly because the software stack lacks to deliver both low network latencies and high network bandwidth.

These inefficiencies originate from different levels [Foo+03]. First and as depicted in figure 2.1, left part, the TCP/IP stack is generally implemented inside

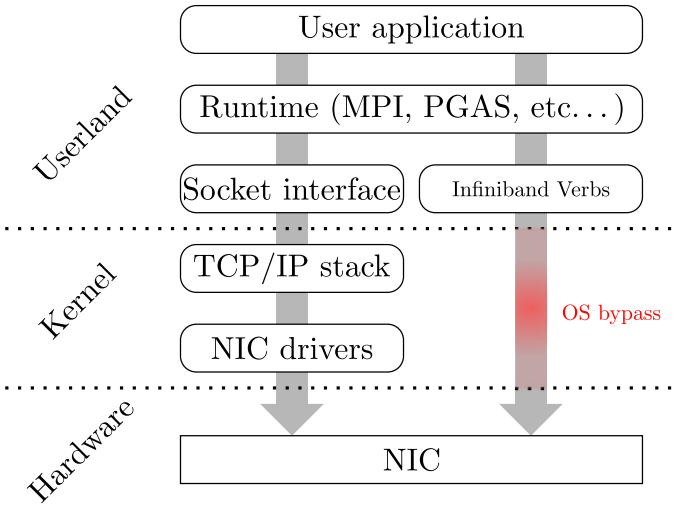


Figure 2.1 – Comparison of two communication libraries. The regular TCP/IP stack (on the left) involves the OS during communications. Modern interconnects such as Infiniband (on the right) support technologies to bypass the OS.

the OS. Thus, reading/writing through a UNIX socket induces OS overhead, socket and protocol processing, which inevitably increases latency. Second, sending and receiving data generally involves a memory copy from the user space to the restricted kernel space. It is mainly because the Network Interface Controllers (NIC, i.e., component that connects the compute node to the network) do not have the information where the received data are placed in memory. This causes the memory caches to be flushed with data from communications. Furthermore, a significant part of the network protocol (e.g., message fragmentation, Direct Memory Access initiations for data movements) is driven by the host processor, which avoids asynchronous progression of communications while the CPU is busy.

2.1.2 Facilities of Modern Interconnects

On the strength of earlier experience gained from TCP/IP sockets, interconnect vendors have started to develop new technologies for improving high-speed networks. One of the major innovations is the arrival of embedded processors in the NICs, allowing complex operations to be offloaded from the CPU to the network card. NICs can now drive themselves communications and deliver data directly to the application, bypassing the OS and avoiding any involvement from the CPU.

2.1.2.1 OS Bypass and Zero-Copy

On regular platforms, the OS keeps an exclusive access to computer devices and network operations must be handled by a *device driver*. This intermediate operation is required because of security concerns. Indeed, a direct access to a device could corrupt the system integrity and violate the resource partitioning: any program could intercept the data from another program passing through the device. As a result of regular networks that involve the OS, user-space applications suffer from high overheads while accessing network devices (see section 2.1.1).

When network bandwidth began to approach the bandwidth of memory copies, the buffering of network data internally performed by the OS rapidly became an issue for performance. This concern led to the development of zero-copy message-passing

protocols in which intermediate messages copies are eliminated to avoid the loss of bandwidth. A typical zero-copy protocol requires the network card to generate an interruption for the CPU when a message becomes ready. The interrupt handler then manages the message transfer into the final virtual address of the right application. This approach however increases latency because the time required by the OS to start the interrupt handler is fairly significant. Additionally, the CPU remains busy during the interruption handler and cannot perform useful computations.

To avoid these overheads, some modern NICs are assisted by a processor that can be programmed to implement part of a message-passing protocol. As an example on conventional TCP networks, it is possible to program the NIC to take in charge the whole protocol stack, including segmentation, checksum, sequence number calculation and acknowledgement of packets without requiring even a single cycle from the CPU. Because this technique does not need to involve the OS on message transfers, it is frequently called "OS bypass". This technology is depicted in figure 2.1, right part.

To support OS bypass with communication offload and achieve highly efficient data transmissions, a few networks like Infiniband and the BlueGene/Q network rely on RDMA operations: a process can directly read from (RDMA *get*) and write to (RDMA *put*) the memory exposed by another process and without requiring its participation. This communication model however raises two issues: first, the network card manipulates virtual addresses and only the OS knows the virtual to physical addresses mapping. Second, the OS may swap virtual pages at any times. To tackle these concerns, the network card must query the OS for the physical address of each virtual page to be transmitted. In addition, to prevent page swapping, virtual pages must be registered as "unmovable" in the OS.

2.1.2.2 Connectionless

Regular sockets API that use TCP/IP are connection-oriented. It requires the OS to allocate a few amount of memory to maintain the state of each active connection and store internal buffers. In fully-connected systems (i.e., all processes are connected together), the number of connections rapidly increases with the number of processes, exhibiting an $\mathcal{O}(n^2)$ memory usage (for a system with n processes). As a consequence, maintaining a state per connection imposes a limit on the application scalability. Some modern interconnects like Infiniband or the BlueGene/Q network provide connectionless facilities. In contrast to the TCP/IP stack, processes do not need to explicitly establish a connection to communicate.

2.1.2.3 Independent Progress and Application Bypass

Many protocols that support OS bypass still require the application to actively participate in the messaging protocol to ensure a proper message progression. Independent progress enables a transparent data progression and completion independently of entering or not the network progression function. This enhancement is particularly relevant for some current parallel programming languages such as MPI where complex messaging protocols are involved. First, nearly all MPI runtimes support a set of specialized protocols dedicated to large messages using RDMA and called `rendezvous`: before starting data transmission, sender and receiver tasks synchronize on the destination buffer where the message will be copied at the receiver side. In practice, the sender emits a `rendezvous` request to the receiver, which in turn

returns to the sender the address in memory of the destination buffer. If the receiver replies in late to the `rendezvous` request, the delivery of the message may be delayed. Consequently, offloading the reply process to the NIC may accelerate MPI message progressions, and so reduce message latency. Second, research groups have started to emphasize the benefits of offloading a portion of the MPI collective communications, especially as the third version of the standard enables collective operations to be non-blocking [BP03; Gup+03; Kan+11; Kan+12].

The capability to offload MPI operations to recent NICs may however lower communication performance. As an example, because processors embedded with NICs are slower than host CPU, the time to execute long operations (such as process long queues of posted receives and unexpected messages) may be degraded [UB04; Bri+05]. Moreover, offload capabilities are often specific to a particular network and often require dedicated hardware (i.e.,: the collective offload from Mellanox involves network switches that are compatible with this technology).

Nowadays, most of the networks provide an hardware or software facility to ensure a transparent message progression. Other networks such Infiniband do not support a fully independent progress but rely on interrupts to simulate this capability.

2.1.3 Overview of Interconnects for HPC

When designing HPC systems, a preliminary study should highlight what aspects can widely influence application performance. With the growth in compute nodes number, one of the key tenants of performance is the network. It includes different aspects, from the latency to the network throughput, including the capabilities supported and the topology. Undervaluing any of the previous points would be critical, once the supercomputer has gone into production: an inappropriate network would for example lead to link or switch congestion which finally would provide poor performance of end-applications. At the opposite, overvaluing a network would considerably increase the total cost of the machine.

In the next section, we describe the characteristics of widespread past and present networks for interconnecting HPC supercomputers. We categorized them into (1) system-on-a-chip networks that integrate networking components directly on the same chip as CPUs and RAM and (2) off-chip networks that involve independent interconnection switches to relay communications. Furthermore, we describe the fundamental capabilities of each network and define some user interfaces that are available to program them.

2.1.3.1 System-on-a-Chip Networks

IBM's Blue Gene/Q network [Che+12] and Cray's Gemini [Inc10] are two proprietary networks that integrate a System-on-a-Chip (SoC) connected to a multidimensional torus network (3D for Cray, 5D for IBM). Both networks are reliable, support RDMA operations and connectionless communications.

To program Cray's Gemini interconnects, two communication libraries are exposed to end-users. On the one hand, the Gemini Network Interface (GNI [Inc11]) directly exposes the communications capabilities of the network to the user-space software. This interface is commonly chosen for developing MPI implementations [PGB11]. On the other hand, the Distributed Memory APPlication (DMAPP) API implements a logically shared, distributed memory library for interfacing the

PGAS languages detailed in section 1.2.2. The Blue Gene/Q network is capable of 2 GB/s unidirectional bandwidth and natively supports collective operations such as barrier, broadcast, reduce and allreduce over the same physical torus. Furthermore, since a physical core is composed of four hardware threads, MPI libraries can dedicate one or two threads per core to communications in order to enable independent progress. The Parallel Active Messaging Interface (PAMI [Kum+12]) provides a common interface to MPI runtimes and other programming paradigms such as PGAS for accessing IBM's network.

2.1.3.2 Off-Chip Fabric-Based Interconnection Networks

Ethernet

Ethernet was commercially introduced in 1980 as a standard for interconnecting computers in Local Area Networks (LAN). Nowadays, it equips a large part of the world's Top500 supercomputers [Top] (43% according to the list of June 2013) . Behind its success, Ethernet is a cost-effective and easy-to-deploy technology with (for the 40Gbit version) a competitive peak bandwidth.

Programming Ethernet cards usually relies on UNIX sockets and TCP/IP since this protocol is natively supported for reliable communications on modern OS. In order to minimize the involvement of the CPU into communications, some techniques propose to offload expensive TCP communication tasks directly to the network card. For this purpose, a few top-of-the-range network cards are certified TCP Offload Engine (TOE) where the hardware takes partially or totally in charge the TCP protocol. The adoption of this technology is however limited since it raises several concerns for the operating system (eg: security, complexity of integration[1]).

To provide low latency and high throughput, the Internet Wide Area RDMA Protocol (iWarp [Hil+03]) introduced in 2002 delivers RDMA services over standard, unmodified IP networks. Besides a support of a broad range of network characteristics, iWarp's usage is very limited due to implementation challenges and is mainly limited to long-distance TCP connections. During the last few years, there has been an increasing focus on a new standard called RDMA over Converged Enhanced Ethernet (RoCE). RoCE provides a superior solution compared to iWarp and allows for performing native Infiniband communications over lossless (i.e., Data Center Bridging (DCB) capable switches) and non-lossless Ethernet links. Alternatively, a "pure" software implementation of RoCE is available for ordinary Ethernet NICs [KKB14].

Finally, in 2013, Cisco announced the Userspace NIC (usNIC) technology that allows low-latency ($\simeq$ 2 us MPI ping-pong latency) on Ethernet links with OS-bypass support[2].

QsNet and Myrinet

QsNet^{II} [Bee+03] interconnect released by Quadrics and Myrinet [Bod+95] introduced in 1994 by Myricom are two switch-based interconnects that involve individual network switches for relaying communications. Furthermore, both provide a support for RDMA transfers and propose to connect the compute nodes using a fat-tree topology.

[1] Linux TOE workgroup: http://www.linuxfoundation.org/collaborate/workgroups/networking/toe

[2] Jeff Squyres' EuroMPI'13 usNIC presentation: http://blogs.cisco.com/performance/eurompi13-cisco-slides/

QsNet^{II} consists of two hardware parts: (1) a programmable network card called Elan-4 and (2) a communication switch called Elite-4. To program Elan-4 cards, Quadrics proposes the Tports interface which exposes basic mechanisms for point-to-point message passing such as MPI [Yu+05]. The interface allows a process to execute a programmable thread in the network card. This innovative approach provides facilities to implement complex message passing protocols in hardware and enables independent progress. Furthermore, the NIC supports a memory management unit (MMU) to transparently register virtual pages to be transmitted in the OS.

The last Myrinet device named Myri-10G provides a bandwidth of 10 GB/s and natively handles two network modes: the MX (*Myrinet eXpress*) protocol mainly dedicated to HPC applications and the standard Ethernet protocol for interfacing an already existing network such as a storage network. The high performance of the MX protocol makes it a good candidate for HPC. The latency falls to 2 μs and the bandwidth can reach 1.25 GB/s in each direction (for single lane (x1) PCI-Express). Furthermore, the high-level programming interface covers up memory registering/unregistering which are internally managed by the MX library and executed on the host CPU. Finally, the basic communication primitives of MX-10G are non-blocking send and receive operations, which can directly be used in the implementation of MPI communication primitives.

In spite of the well-featured hardware of QsNet and Myrinet, their HPC market share has never stopped to decline in the past ten years. Quadrics has stopped its activities in 2007 and Myricom has gone from the market before being acquired by CSP in 2013. Currently, Myrinet only equips 3 of the world's Top500 supercomputers and Quadrics is not ranked anymore.

Infiniband

Infiniband is a server and storage interconnection standard initiated in 1999 by a consortium regrouping computer science companies such as Compaq, IBM, Hewlett-Packard, Intel, Microsoft and Sun. The Infiniband Architecture (IBA [Inf]) released by the Infiniband Trade Association (IBTA) specifies a set of abstract definitions called Infiniband `verbs` that provides a software interface for programming over the end nodes. Because of a lack of standardization, manufacturers have began to implement their own interface but they were incompatible with each other. It was not until the availability of the OpenFabrics project that a low-level portable programming interface was provided. The adoption of the Infiniband network has been strengthened by the availability of high-level socket-based interfaces such as SDP (Socket Direct Protocol) or IPoIB (IP over Infiniband). The latter solution allows for encapsulating the TCP/IP protocol inside Infiniband network frame, ensuring a backward compatibility of applications.

In the HPC context, the `verbs` are used because the highest performance can be achieved with this interface. Indeed, the `verbs` interface is very low-level and the responsibility of communication protocols, buffer and connection management are left to the developer. Especially, the `verbs` interface exposes many capabilities supported by the network hardware like one-sided RDMA operations, OS bypass, atomic operations and network reliability. The volume 2 of the architecture specifications offers a wide range of both copper and optical cables that implementers can aggregate in units of 4 or 12, called 4X and 12X. In addition, the Infiniband specifications define five speed grades from SDR (Single Data Rate) to EDR (Enhanced

Data Rate). The last generation Infiniband Host Channel Adapters (HCA, i.e., network hardware that connects to Infiniband network) from Mellanox based on the FDR (Fourteen Data Rate) standard aggregates 4 links running at 14.0625 Gb/s. It results in an effective unidirectional bandwidth of 54.54 Gb/s.

In summary, Infiniband is a very low latency network and provides a bandwidth generally higher than current other equipments [Vie+12]. Moreover, according to the Top500 list from June 2013, it now equips more than 41% of the world's fastest supercomputers.

2.1.4 Programming Infiniband

During the last decade, Infiniband has been successfully succeeding in providing a valuable network that is suited to HPC supercomputers. It now equips some of the largest computing centers around the world. In France, the Tera-100 supercomputer from CEA and the Curie supercomputer owned by GENCI use Infiniband networks. To develop applications for Infiniband, a large spectrum of user-level interfaces is available from the highest communication libraries to the lowest-level with Infiniband `verbs`.

2.1.4.1 High-Level Communication Libraries

The Common Communication Interface (CCI [Atc+11]) and the Lawrence Berkeley National Lab's GASNet [Bon02] project are two high-level communication libraries that support various interconnects including Infiniband. CCI intends to create a simple, high-level communication interface while providing scalability, resiliency and performance. It uses an active message style [Eic+92] for small/control messages and RDMA transfers with a zero-copy mechanism for large data movements. Furthermore, buffer management (allocation and recycling) is operated by the CCI library itself. The interface yet supports the UDP/TCP sockets, the Cray's Gemini network and Infiniband networks through the `verbs` interface. The authors also provide a proof-of-concept for the Portals interface and the MX protocol.

GASnet allows a standard application interface to be implemented over a wide variety of standard and high-performance networks. It aims at being a portable, language-independent, high-performance, one-sided communication interface that provides an abstraction of the network and operating system for the implementation of the PGAS languages detailed in section 1.2.2. Currently, GASNet supports execution on many networks amongst Myrinet, Quadrics and Infiniband thanks to the OpenFabrics `verbs` interface described in the next section.

2.1.4.2 Low-Level Programming Interfaces

Low-level programming interfaces are generally restricted to a few interconnection networks but allow a direct access to the capabilities proposed by the hardware. In the following section, we introduce four low-level programming interfaces for Infiniband: Portals, uDAPL, `verbs` and Mellanox VAPI/MXM which are two proprietary extensions to the regular `verbs`.

Portals

Cray/Sandia's Portals [Bri+] is an interface for message passing between nodes of a parallel computing system. Originally developed for Cray's networks, it has

evolved into an interface that can be efficiently implemented for different operating systems and networking hardware. Although focusing on the MPI standard, it is flexible enough to support a variety of higher-level data movement layers based on one-sided operations. To improve scalability, Portals is connectionless. It supports reliable, ordered delivery of messages without explicit point-to-point connection establishment between pairs of processes. It combines both channel and memory semantics respectively with regular two-sided Send/Receive and RDMA operations. To the best of our knowledge, only the Portals 4 Reference Implementation [Laba] implements Portals 4 over Infiniband `verbs`.

uDAPL

Direct Access Transport (DAT) defines a transport-independent and platform-independent set of APIs that benefits from the RDMA capabilities of modern interconnects. The Direct Access Programming Library (DAPL) was released by DAT Collaborative and defines both user level (uDAPL [Len+03]) and kernel-level (kDAPL)[3] APIs. An open source reference implementation of the uDAPL v1.0 interface is available and currently supports Infiniband. DAT however targets other systems than Infiniband and provides a solution for programming future RDMA-based interconnect. Nowadays, some well-known MPI libraries attempt to leverage the portability of uDAPL and provide a compatible layer, such as Intel MPI, Open MPI and MVAPICH2.

Infiniband `verbs` and Mellanox VAPI/MXM

To enhance portability across equipments from different vendors, the OpenFabrics Alliance evolved the `verbs` specification from the IBA standard into a complete open-source API, which is called the OpenFabrics verbs (initially known as OpenIB `verbs`). The OF `verbs` are included in the OpenFabrics Entreprise Distribution (OFED), an open-source software stack which is available for many Linux and Windows distributions. It includes a support for legacy 10 GB Ethernet, iWARP for Ethernet, RDMA over Converged Ethernet (RoCE) and 10/20/40 GB Infiniband. The OpenFabrics `verbs` interface is currently the most widely used API for writing applications over Infiniband networks.

The Mellanox IB-Verbs API (VAPI [Tec13b]) and its successor, the Mellanox Messaging (MXM [Tec14]) are interfaces that implement the regular `verbs` from the Infiniband standard plus additional features related to the recent HCAs that support the ConnectX CORE-*Direct* technology. For example, they provide an interface for offloading collective operations [Tec11] and propose two additional transport protocols: the eXtended Reliable Connection (XRC) and the Dynamically Connected Transport (DCT [Tec13a]). The both interfaces will be discussed in section 2.2.5.

2.1.5 Discussion

Table 2.1 summarizes the hardware capabilities of high-speed networks previously presented and their system share according to the Top500 list as well. Because of its dominant position in HPC with 41% of the 500 world's fastest supercomputers, the following thesis focuses on Infiniband networks. Furthermore, from the various APIs and high-level libraries available for programming Infiniband, the thesis uses the OpenFabrics `verbs`. Indeed, this low-level interface provides the most fine-grain

[3] uDAPL reference implementation: http://sourceforce.net/projects/dapl

Table 2.1 – Comparison between capabilities of high-speed interconnects for HPC and their system share

Network		Memory Access	Connection-less	MPI matching in hardware	Collectives support	System Share[1]
Ethernet	iWarp, RoCE	RDMA	Partial, unreliable[2]	No	Unreliable multicast[2] (`verbs`) + collective offload (Mellanox VAPI/MXM)	43.0 %
Ethernet	usNIC	DMA	Unreliable[2] Reliable planned	No	No	
QsNet		RDMA	Yes	Hardware support	Collective offload	0.0 %
Myrinet		DMA	No, unreliable network	No	No	0.6 %
BG/Q Net.		RDMA	Yes, reliable network	Software helper thread (PAMI)	Supported in hardware	4.4 %
Gemini		RDMA	Yes, reliable network	No	No	3.2 %
Infiniband		RDMA	Partial, unreliable[2]	No	Unreliable multicast[2] (`verbs`) + collective offload (Mellanox VAPI/MXM)	41.0 %

[1] According to the Top500 list from June 13
[2] "Unreliable" means that the network hardware does not guarantee the reliable delivery of data to the destination.

control over the HCA for developing complex communication protocols. Moreover, `verbs` interface gives the lowest latencies through a direct access to low-level operations such as RDMA transfers, memory registration and buffer management. Finally, the OpenFabrics Alliance well-documents the `verbs` interface, which facilitates developments.

2.2 Infiniband Overview

In the following section, we describe the structures involved in Infiniband developments over the `verbs` API. As a first step we introduce the available communication semantics as well as various transport modes for delivering messages. Second, we detail the memory registration process and mechanisms involved in message completioning. To finish, we present some hardware optimizations and enhancements from the Infiniband specifications to reduce the memory consumption of Infiniband resources.

2.2.1 Communication Semantics

Infiniband supports two types of communication semantics which allow for developing a wide range of applications. The two-sided channel semantics involves *send* and *receive* methods while the memory semantics enables zero-copy data transfers with RDMA *put* and *get* operations.

2.2.1.1 Send-Receive

With the channel semantics, after the sender posted the send request to the `verbs` internals, the network card then drives the data transfer to the corresponding memory region at the receiver side. Infiniband requires the receiver to post a buffer in the remote communication endpoint before the sender initiates the communication. This requirement is actually prevalent in most high-performance networks like Myrinet [Bod+95] and Quadrics [Pet+01]. Compared to the memory semantics, the sender has no control over where the data will reside in the remote process. Furthermore, the channel semantics implements a control flow mechanism that prevents the sender to complete until a request has been posted by the receiver. At the receiver side, the buffer size must be equal or greater than the one posted at the sender side since the size is not checked prior to the transfer. In case of reliable transports where the reliability of data is guaranteed during transmission, if the receive buffer is undersized, both send and receive communication endpoints enter into an error state and the connection has to be reinitialized.

2.2.1.2 RDMA *Put*/*Get* & Atomic Operations

With the memory semantics, RDMA enables the network adapter to transfer data directly from or to the memory of a remote process. The caller specifies the remote virtual address to acess as well as a local memory address to copy. For security concerns, the remote host must provide appropriate permissions to its memory prior to the RDMA transfer. By default, the remote host is not informed that an RDMA transfer is completed. The user may request the HCA to generate a notification on the remote host as soon as the RDMA operation completes. Furthermore, this notification can contain a 16-byte immediate data to be set by the local host before posting the RDMA request.

Two atomic operations are also available and extend the regular RDMA *put*/*get* operations. The *compare-and-swap* operation atomically compares a value pointed by an address with a given value. If both values match, the specified value is stored at the provided address and the old value is returned to the caller. The *fetch-and-add* operation atomically adds the value pointed by an address with a given number. The previous value is returned to the caller.

2.2.2 Queue Pairs and Infiniband Transport Modes

To exchange messages, HCAs communicate through a logical connection endpoint composed of a send and a receive queue, both referred to as Queue Pair (QP). As a comparison, this is roughly equivalent to a UNIX socket. To communicate data, the user posts a Send Request (SR) to the available send QP. A SR defines how much and which data will be sent to which remote QP. It also defines the protocol to use (Send/Receive or RDMA) and the target address that will be used on the remote process for RDMA operations. Optionally, the user may require the hardware to signal the application when a SR completes. At the receiver side, the remote QP associated to the send QP must have previously queued an Receive Request (RR) to retrieve the data. A RR defines a buffer where data are being received for non-RDMA operations.

The Infiniband specifications provide four different transport modes which are detailed in table 2.2. The following thesis however only focuses on Reliable Con-

Operation	Unreliable Datagram (UD)	Unreliable Connection (UC)	Reliable Connection (RC)	Reliable Datagram (RD)
Send (with immediate)	X	X	X	X
Receive	X	X	X	X
RDMA Write (with immediate)		X	X	X
RDMA Read			X	X
Atomic: fetch-and-add/ compare-and-swap			X	X
Max message size	MTU	2 GB	2 GB	2 GB

Table 2.2 – Capabilities of Infiniband transport modes (courtesy of Mellanox's RDMA Aware Networks Programming User Manual).

nection (RC) and Unreliable Datagram (UD) transport modes. Indeed, since the Infiniband specifications do not require Reliable Datagram (RD) to be supported by any HCA, it is (as far as we know) not available in current hardware. Concerning Unreliable Connection (UC), it is roughly a mix between RC and UD, where the connection is not reliable and a QP is associated with only one other QP.

RC is actually the most commonly used transport mode for developing applications over Infiniband: a QP is associated with only one other QP and messages using this transport are reliably transmitted and delivered to the receiver. It is roughly comparable to TCP. In addition, hardware control flow guarantees that packets are delivered in order regardless the semantics, channel or memory. Because RC is connection-oriented, it requires a connection to be explicitly established between two QPs prior to any communication. At first glance, developers could be encouraged to use an IP-based network such as Ethernet or IPoIB – most of the time supplementary to Infiniband networks for cluster administration – for interconnecting QPs. However, as described in section 2.1.1, TCP-based networks induce several overheads and this solution may rise performance issues on large-scale applications. To assist the developer with QP management, the IBA community provides the RDMA Communication Management (RDMA CM). RDMA CM exposes an interface close to socket programming and includes the protocols and mechanisms used to establish, maintain and tear down QP connections. A few MPI runtimes such MVAPICH already leverage RDMA CM and enable a fast QP interconnection mechanism on top of this interface [YGP06].

Only recently, applications have started to emphasize the advantages of UD for large-scale applications over Infiniband. Where RC requires a separate QP for each connection, one QP is enough with UD to transmit and receive packets to/from any other QP. As a consequence, UD aims at providing a near-constant memory-footprint of network structures, regardless the number of processes. However, UD requires much more software infrastructure that makes the communication layer more complex and usually degrades the performance. First, like UDP connections, ordering and delivering are not guaranteed by the hardware. Applications that require reliability have consequently to implement their own reliability protocol. Second, packets cannot exceed the size represented by the Message Transfert Unit (MTU) and must be manually split, operation normally handled by the software with RC.

2.2.3 Memory Registration

Compared to networks such as Quadrics, Infiniband does not provide hardware-assisted facilities for registering memory, meaning that all data used for communication must be manually pinned by the user [Tez+98]. As depicted in figure 2.2, memory registration is a four-step operation that (1) prevents virtual pages from being swapped out by the operating system and (2) allows the Host Channel Adapter (HCA) to get the corresponding physical to virtual page mapping. The procedure is summarized as follows:

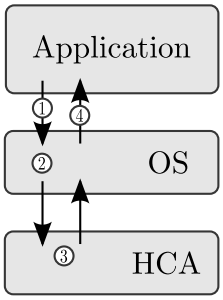


Figure 2.2 – Infiniband memory pinning process

1. The application requests a memory registration. It sends to the OS the virtual address and the length of the contiguous data.
2. During the registration, the OS checks the permissions of the memory region to register. Then, it pins the region into physical memory and translates virtual to physical mapping.
3. The physical address table is transmitted and written to the network adapter. A handle is issued by the network card. It is composed of a pair of keys: the remote and local key (r_key, l_key). Local keys are used by the local HCA to access local memory, for example during a receive operation. Remote keys are required by the remote HCA to allow a remote process to access the main memory during RDMA operations.
4. Finally, the handle is returned to the application.

To conclude on memory registration, it is an expensive operation which introduces a significant overhead in communications. Furthermore, its cost is proportional to the number of pages to register and depends on whether the pages are present in physical memory on not [Mie+06].

2.2.4 Completion and Event Handling Mechanisms

Infiniband is an event-based network: when a previously posted Send Request (SR) or Receive Request (RR) completes, a notification can be posted by the HCA into an event-queue called the Completion Queue (CQ). A CQ can service send queues, receive queues or both. Additionally, a single CQ can be associated to multiple QPs. As a result of a completed event, a Completion Queue Entry (CQE) is filled and provides a brief description of the event which has been generated. It includes for example the status of the transfer and, in case of failure, the error code from the faulty transaction.

Because Infiniband does not support independent progression, it is the user's responsibility to regularly poll the CQ to check for completed messages. If the CQ encounters an overrun, it is shut down and an asynchronous event is generated by the HCA.

To check the completion of messages, the `verbs` interface makes available two strategies: the *polling-based* and the *event-based*. The polling-based synchronous

strategy requires the user to manually and periodically poll the network for retrieving messages. This method however raises two severe issues. First, because the CQ is checked whether it contains outstanding CQEs or not, it may waste useful CPU cycles. Second, if the CPU is busy on computations for a long time, the CQ may fill up in the background and lead to a fatal overrun. To address these issues, the `verbs` interface provides an event-based asynchronous message progression. The application may create an additional thread which remains blocked until an incoming message becomes ready in the HCA. When it happens, an event is generated to the application and the progression thread is scheduled to receive the message. The majority of MPI implementations usually support the two strategies and let the user decides which one to use. In a fully-subscribed context where the progression-thread and the MPI task share the computational unit, a threaded message progression may not be recommended. It is partially due to the overhead that is generated by a high number of context-switches (see section 4.2.1). Consequently, MPI runtimes often combine the two approaches: CQs are polled entering MPI functions and only messages that have low latency constraints are generating events [Sur+06a; Kum+08].

2.2.5 Memory-Friendly Infiniband Endpoints

Infiniband normally requires each QP to have a dedicated Receive Queue (RQ) where Receive Requests (RR) are posted. This model may however be inefficient in terms of memory usage. First, in cases where a RQ runs out of RRs, it cannot use RRs from another RQ. Second, if too many RRs are allocated to a RQ that receives a few messages, some RRs will never be used and will waste memory. To reduce the memory required by RQs, the Infiniband specifications has been enhanced and introduce since the version 1.2 a new mechanism called Shared Receive Queues (SRQ). Instead of allocating one dedicated RQ per QP, a single SRQ can be shared between all QPs in the same process. As a result, a smaller number of RRs are needed and the memory required per QP is reduced [Sur+06b; Shi+06].

On the same topic, hardware vendors propose proprietary and non-standard solutions to reduce the amount of memory required by the network. ConnectX, the most recent generation of Infiniband HCAs released by Mellanox Technologies support two memory-friendly alternatives to the Infiniband Reliable Connection transport called eXtended Reliable Connection (XRC) and Dynamically Connected Transport (DCT [Tec13a]). When a XRC QP is connected to another process on a different node, it can reach all processes on that remote node via the same QP [Shi+08; KSP08]. As for DCT, it improves scalability by setting up and tearing down the connections by the adapter hardware on an 'as needed' basis. This transport protocol is available in a stable version since April 2014 and only Open MPI supports it for the moment.

2.3 Experimental Platforms

To run large-scale experiments, we dispose of the Curie supercomputer owned by GENCI and operated into the TGCC by CEA. It is the first French Tier0 system open to scientists through the French participation into the PRACE research infrastructure. All nodes are interconnected together on an Infiniband network following a 2-level Fat Tree network [Lei85]. The first level of the tree is composed of 36-port

switches, each connected to 18 324-port top switches. Compute nodes are connected to the Infiniband network through Mellanox MT26428 ConnectX III 1-port 4x QDR HCAs for a total of 32Gbit/s of unidirectional theoretical throughput. As of the time of writing, compute nodes are running GNU/Linux x86_64 2.6.32 customized by Bull and SLURM [JG04] is responsible for jobs allocation. Moreover, the version 1.5.4.1 of OFED is installed on the cluster, which includes the `verbs` library version 1.1.4.

Without including the hybrid nodes for GPU computing, Curie is now offering 2 different compute nodes. An additional partition composed of 32-core nodes was temporary available on Curie. In mid 2012, every 32-core nodes of this partition were converted into 90 super nodes of 128 cores.

2.3.1 Thin Cluster: 16-core nodes, 1 HCA

The *Thin Cluster* is composed of 5040 B510 Bullx nodes, each equipped with 2 eight-core Intel processors Sandy Bridge EP (E5-2680) clocked at 2.7GHz and assisted with 64 GB of main memory. According to the June 2013 Top500 list, Curie Thin nodes is the 15^{th} most powerful supercomputer with 1,359.0 TFlop/s on the Linpack benchmark.

2.3.2 Medium Cluster: 32-core nodes, 1 HCA

The *Medium Cluster* is no more available for production. It regrouped 360 compute nodes, each composed of 4 eight-core Intel Nehalem-EX X7560 processors clocked at 2.26GHz with 128 GB of main memory.

2.3.3 Large Cluster: 128-core nodes, 4 HCAs

The *Large Cluster* relies on a specific and proprietary *Bull Coherent Switch* (BCS) grouping 4 motherboards from the *Medium Cluster* together for a total of 128 cores and 512 GB of memory (see figure 2.3). This cluster targets hybrid parallel codes (e.g., MPI+OpenMP) that require large memory and a multi-threading capacity. Each level-2 NUMA nodes (groups of 32 processors) is topologically close to 1 HCA.

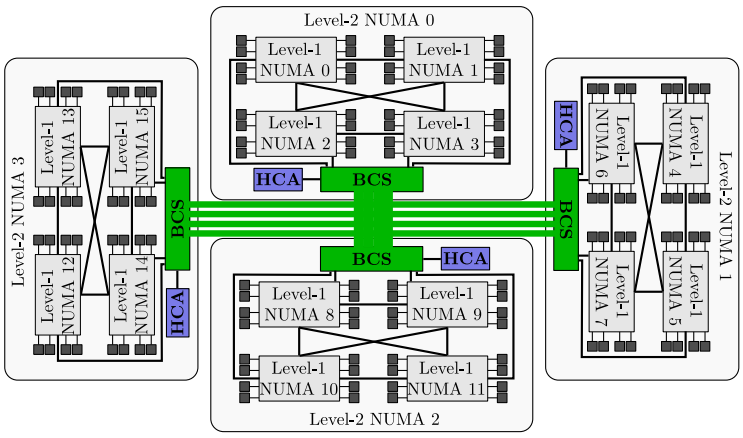


Figure 2.3 – Architecture of the compute nodes that compose the Curie's *Large Cluster*

Part II

Contributions

Chapter 3

Memory-Scalable MPI Runtime

"When comparing human memory and computer memory it is clear that the human version has two distinct disadvantages. Firstly, as indeed I have experienced myself, due to aging, human memory can exhibit very poor short term recall."

Kevin Warwick

3.1 Memory Footprint: a Limit to the Scalability of MPI Runtimes

Since the first revision of the standard by the MPI Forum, MPI is targeting extremely large-scale parallel systems. Until now with the large amount of memory available per execution unit, MPI developers did not have to worry too much about the memory consumption of the runtime for large-scale runs. As previously discussed in section 1.3.4, the trend however seems to be revering since the last exascale reports [Ama+09; Ash+10] and the recent hardware (e.g., Intel MIC architecture) suggest that the number of cores per node will increase by a factor of 4-5 while the amount of per-core memory will decrease by a factor of 4-6. Indeed, the memory per core envisaged is around 10-20 MB. To scale parallel applications on such future machines, efforts are needed to reduce the memory consumption from the MPI users down to the runtime [Tha+10].

Because MPI targets distributed memory architectures, the standard actually requires some user-level data to be duplicated across MPI tasks. To limit this duplication, MPI users may first share memory consuming data via a shared memory segment. With the aim at automatizing this cumbersome process, Tchiboukdjian *et al.* [TCP12] propose a technique based on `pragmas` to share at a node-level data that are often read and barely written. A second technique to reduce data duplication is to implement a hybrid version of regular full MPI codes using a shared-memory programming model like OpenMP, TBB or Cilk (see section 1.2.1). Both transformations presented however rely on source-code modifications and some cannot easily be implemented.

Changing the numerical schemes of MPI applications and implementing communication-avoiding algorithms may also be an interesting alternative for lowering the communication volume and so, the memory required to communicate. First, MPI developers may for example limit the usage of fully-connected collective communications such as All-To-All primitives in order to reduce the number of connected peers that could lead to scalability concerns. Alternatively, they may restrict the number of pending MPI operations to prevent bursts of memory allocations and a

large number of outstanding messages. Second, and to accelerate communications, MPI runtimes may occasionally buffer MPI messages in their internals [SGY10]. With the purpose of limiting this buffering, users may use MPI *ready* communications which guarantee that the message is transferred only when the receive buffer is posted. Finally, the MPI standard provides the concept of *derived datatypes* which avoid the packing/unpacking of non-contiguous data into temporary buffers. To be effective, this solution however requires the underlying runtime to efficiently support datatypes [WWP04; KHS12; Aum+07].

In the following section, we discuss the runtime resources that can limit the memory scalability of MPI applications. We first focus on network endpoints and we present mainstream mechanisms to reduce their impact on the memory consumption. We then detail state-of-the-art communication protocols and show that the memory consumption of network buffers is critical for enabling large-scale runs.

3.1.1 Scalability of Network Endpoints

One of the most challenging points for decreasing memory is certainly to address data structures whose size linearly increases with the number of MPI tasks. In fact, most MPI implementations store $\mathcal{O}(t)$ data per task where t is the total number of tasks in the communication. This includes for example, sequence numbers for message reordering (arrays that are stored locally in each task and which contain sequence numbers for each distant task) or information related to MPI communicators and groups [Bal+09]. MPI applications can also use a significant number of communicators since a common practice in HPC libraries requires to duplicate `MPI_COMM_WORLD` in order to separate their internal communications from the end-user application and other libraries in the same program [HS11b]. Finally, the runtime has to manage inter- and intra-node connections to the remote MPI tasks. These connections are by far the most memory consuming parts of the runtime and are composed of:

- network-independent structures that virtually connect two MPI tasks (for MPICH2 and MVAPICH2, these structures are stored in a *virtual connection* object [Goo+11]);
- several pools of network and shared-memory buffers for send and receive operations [BMG06];
- translation tables to topologically find where a remote MPI task is located on the cluster (i.e., on which node);
- sets of low-level communication endpoints related to the underlying interconnection network: e.g., Queue Pairs (QP) for Infiniband, Sockets for Ethernet;

As previously described in section 2.2.2, Infiniband supports Reliable Connections (RC) where the network library stores in memory the connection state between two QPs. Figure 3.1 estimates the memory usage for one process in a fully-connected graph with up to 32,000 processes and according to the number of WQEs in send and receive queues. It demonstrates that QP structures are a critical point for scalability: with 32,000 processes, more than 2 GB of memory are allocated per process and exclusively dedicated to QP management.

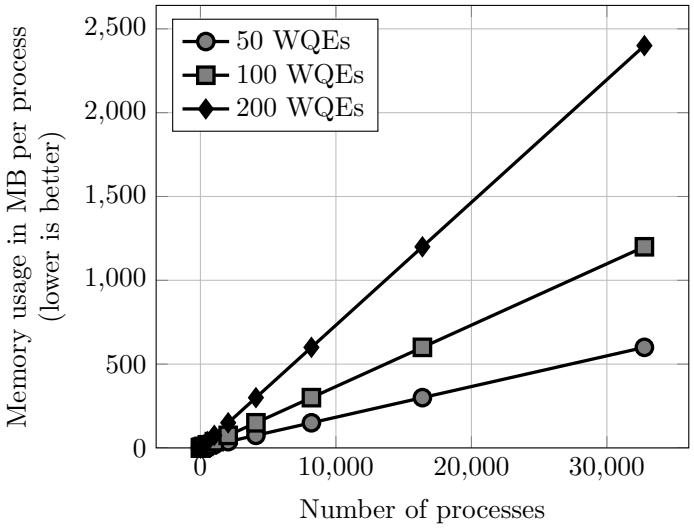


Figure 3.1 – Estimation of memory usage for Infiniband RC in the case of a fully-connected graph with 128 bytes of inline data and without SRQ nor XRC capabilities (see section 2.2.5). WQEs correspond to the entries in QPs which describe how messages will be sent (i.e., Send Requests in Send Queues) and how they will be received (i.e., Receive Requests in Receive Queues). They include for example the semantics, the target RDMA address or the data size). Inline data allows the reduction of latency of short messages by storing data directly into the WQE.

Aware of this scalability concern, MPI runtimes have already started to investigate alternative solutions to decrease the memory allocated per endpoints and/or reduce their number. The first generic proposal aims at optimizing the MPI task placement to maximize the volume of data exchanged within a compute node [MJ11; JM10]. The initial goal was to make an efficient use of high-speed intra-node communications but because two communicating tasks are spawned in priority in the same compute node, it may also reduce the number of network structures required to communicate. Another solution would be to create a virtual network where only a few processes are interconnected. This solution that is convenient for Grid computing [VU05] however induces a high latency penalty since communications require a software routing algorithm involving several hops. Finally, the size of inline data could be reduced, but it would consequently increase the latency of short messages.

As presented in section 2.2.5, the XRC protocol provides a better scalability on multi-core clusters since fewer QPs are required to communicate. This protocol has started to be integrated inside regular MPI runtimes and experimental results show a significant memory reduction [Shi+08; KSP08]. The Unreliable Datagram (UD) presented in section 2.2.2 is an attractive alternative to RC for memory-friendly MPI runtimes [Koo+07; Fri+07]. Experiments have however proven that a slight overhead is noticed while using UD. It is mainly because many RC capabilities are natively handled by the network hardware whereas it has to be implemented in the software stack for UD. As an example, RDMA operations as well as reliability and ordering are not supported with this transport mode [KSP07]. Additionally, UD requires messages to be manually split to the size of the Message Transfer Unit (up to 4 KB). This operation natively driven by the HCA in RC requires the involvement of the host CPU with UD. Thus, due to performance issues with UD, MPI runtimes often rely on hybrid RC-UD approaches where UD is used for the first few messages

before an RC connection is set up [YGP06; KJP08].

In practice, maintaining a fully-connected network is not required for regular parallel programs [VM03]. Because not all pairs of MPI tasks communicate with each other in most applications, MPI runtimes often implement an on-demand connection mechanism [Wu+02; YGP06]. Initially, the application starts with a minimum of connections. During the application lifespan, communication channels are dynamically created according to the affinity between the MPI tasks. When a communication requires a connection between two tasks, an on-demand connection protocol is triggered by the sender task. Once the protocol is completed, MPI tasks are connected and the data transmission can begin. As a result, only the pairs of endpoints that communicate are allocated in memory.

3.1.2 MPI Communication Protocols and Buffer Usage

For most of MPI runtimes, the whole inter-node communication infrastructure is built on top of regular point-to-point communications. Even communications that involve multiple tasks such as communicator management or collective operations commonly rely on point-to-point messages. With that observation in mind, we clearly understand why optimizing point-to-point communications is so crucial for achieving the best performance. For this purpose, mainstream MPI runtimes implement a large spectrum of different protocols that are most often dynamically selected according to the size of the message being transmitted [KJP08].

3.1.2.1 Low-Latency `Eager` Protocol

By opposition to the PGAS languages presented in section 1.2.2, the MPI standard is two-sided, meaning that a sender task does not know, *a priori*, the destination of the message in the receiver's virtual address space. To communicate data, every MPI runtime implements a basic algorithm called `eager`. As depicted in figure 3.2, the data is sent as soon as possible to the destination task, whether the receive buffer is posted or not.

When MPI sends data, it includes with the buffer additional information called the *envelope*. This information is, at least, composed of a message tag, a communicator, a source and a destination. Once the message has been received, the runtime extracts the envelope and reads it. According to its content, there are two possible cases: when no matching receive request is found, the *unexpected* message is stored inside the runtime internals until a matching receive request gets posted. In the other case, the message is *expected* and the envelope matches a previously posted receive request. Following the matching operation, the target address where the message has to be transferred is calculated. The data are then copied into the target buffer and the `eager` buffer used to move the data is finally released for a later reuse.

The major drawbacks of the `eager` protocol are the memory consumption and the lack of control flow. In fact, because it is a one-sided protocol, the receiver stores unexpected messages into an extra memory space until corresponding receive requests get posted (e.g., `MPI_Recv` or `MPI_Bcast`). Let us consider a sender task that sends a 100 MB message. If the message does not fit the available memory on the receiver, the application may crash. In such a case, the receiver should begin to retrieve the message only when the application has posted the receive buffer. Taking this concern into account, MPI runtimes implement an additional class of protocols

called `rendezvous` which is complementary to the `eager` protocol and represented in figure 3.3.

3.1.2.2 High-Bandwidth `Rendezvous` Protocols

`Rendezvous` are two-sided protocols that require the sender and the receiver to negotiate the buffer availability on both sides before the message transmission actually takes place. Contrary to the `eager` protocol, the first clear advantage of using `rendezvous` protocols is that no temporary copy of the message is performed and the sender waits for the receiver buffer to get ready. The sender initially sends only the message envelope to the destination task. Whenever the matching operation occurred and the final destination address calculated, the receiver requests the sender to send the data. Furthermore, `rendezvous` protocols enable the efficient RDMA *put* and *get* operations of modern interconnects to transmit the data.

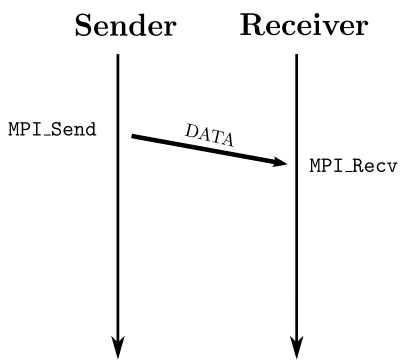


Figure 3.2 – One-sided `eager` Protocol

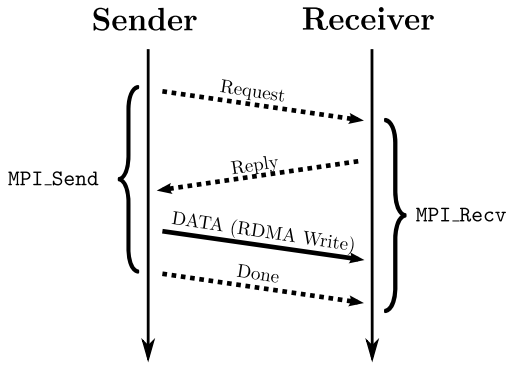


Figure 3.3 – Two-sided `rendezvous` Protocol (based on RDMA write operations)

Designing a `rendezvous` protocol over RDMA however requires the considerations of two severe implications for message latency and bandwidth.

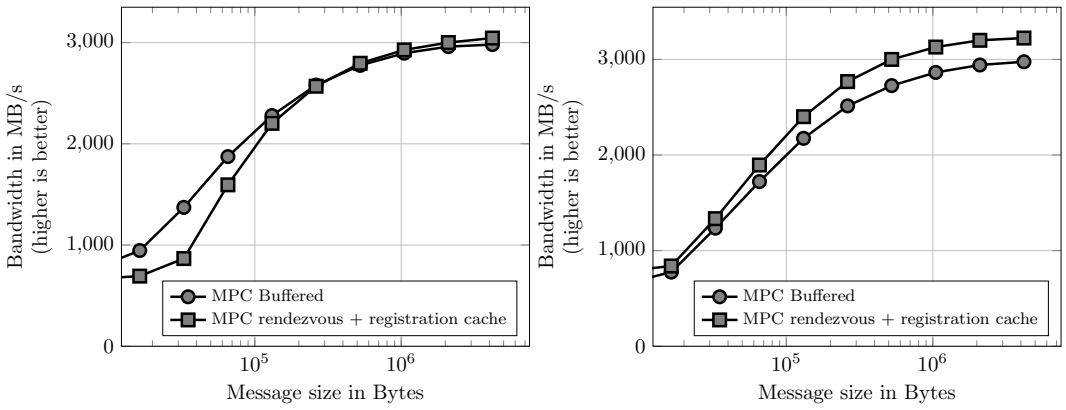
First and as depicted in figure 3.3, `rendezvous` protocols usually require to exchange three control messages which are sent eagerly. More specifically on Infiniband networks, the `DONE` message is usually not transferred by the communication layer and an immediate value can be automatically returned to the receiver by the hardware when the RDMA completes (see section 2.2.1.2). Thus, the `DONE` message can be transferred without involving the sender task. To evaluate the cost of communications using a `rendezvous` protocol, let the network latency be l. The cost in terms of latency to exchange short expected messages using a `rendezvous` protocol is $3 \times l$. In contrast to the `eager` protocol where messages are transmitted with a latency cost equals to l, the `rendezvous` methods are three times slower. In fact, many implementations usually restrict `rendezvous` protocols to large messages. For smaller messages, each MPI task reserves a limited amount of space for eagerly delivering messages. The message size at which a message switches from `eager` to `rendezvous` protocol usually differs from an MPI runtime to another and often depends on the performance of the underlying network. For mainstream MPI implementations over Infiniband, this threshold usually varies from 8 KB to 16 KB.

Second, and as mentioned in section 2.2.3, RDMA operations require a costly memory registration process on both local and distant memory regions prior to transfers. Previous researches have concluded that HPC applications exhibit locality of MPI communication patterns in the spatial domain [KL98] (i.e., a small

number of communication patterns) and in the temporal domain [Fre+04] (i.e, iterative patterns). To optimize the bandwidth of `rendezvous` protocols, MPI runtimes often implement techniques where the memory unregistering is delayed and cached in the runtime internals for a hypothetical later reuse [Hua+06; Gab+04; Liu+03]. When a memory registration is requested, the runtime first searches into the registration cache (or `Rcache`) if an already registered memory region overlaps the one being registered. If so, the registration process is skipped and no overhead is finally induced.

3.1.2.3 Pipeline Protocols

In addition to the regular `eager` and `rendezvous` protocols, several MPI runtimes have developed specialized communication protocols such as pipelines. In their most basic form, "flat" pipelines consist in splitting a message into several fixed-size `eager` buffers in order to pipeline their transmission on the network. Figure 3.4 compares the `rendezvous` protocol (based on RDMA write operations, registration-cache enabled) and the `buffered` protocol of the MPC runtime introduced in section 1.2.3.4. The IMB Ping-Pong benchmark [Intb] executes two tasks on different nodes without buffer reuse (a) and with buffer reuse (b). The MPC `buffered` protocol implements a flat pipeline where the transmission of the message is overlapped with memory copies from the end-user buffer into `eager` buffers. Without buffer reuse (see figure (a)), MPC `buffered` outperforms the standard `rendezvous` protocol up to 256 KB because no memory registration is required by the protocol. However in the case of buffer reuse in (b), the `rendezvous` protocol clearly improves up to 8% the bandwidth of large messages. This overhead is actually due to the multiple memory recopies involved in the `buffered` protocol and that prevent communications to reach the maximal bandwidth. Furthermore, the `rendezvous` protocol only registers once the communication buffers for each message size sample. The registration overhead is consequently substantially reduced.



(a) without buffer reuse (-off_cache 32M)

(b) with buffer reuse

Figure 3.4 – MPI evaluation of the `rendezvous` protocol (based on RDMA write operations, registration cache enabled) and the `buffered` protocol of MPC. The IMB Ping-Pong benchmark executes two tasks on different nodes without buffer reuse (a) and with buffer reuse (b). No buffer reuse means that communication buffers are different within all repetitions and the registration cache consequently fails to re-use previously registered memory regions.

To optimize the performance of flat pipelines, Denis *et al.* [Den11] propose the *superpipeline*. While this pipeline is running, chunk size is increased from chunk to chunk, which leads to a lower number of gaps due to splitting. Experiments on benchmarks with low buffer reuse show better performance than regular `rendezvous` algorithms because the *superpipeline* is capable of achieving a high overlap of RDMA write operations with memory registration operations. In the same line of thinking, Open MPI [Woo+06] implements an RDMA pipeline protocol that efficiently overlaps the cost of memory registration with RDMA operations.

3.1.2.4 Discussion on Network Buffers

To communicate data, MPI runtimes use many network buffers. They are involved at all levels, from the `eager` to the `rendezvous` protocols including pipelines. Table 3.1 reports the amount of memory reserved by send and receive Infiniband buffers inside Open MPI 1.7 running a seismic modeling application. According to the results, Infiniband buffers represent a large amount of memory with 35% of the total memory allocated by the runtime. This observation inside a mainstream MPI runtime such as Open MPI clearly highlights the impact of network buffers on memory consumption.

Table 3.1 – Seismic modelling application on 1,024 MPI tasks with Open MPI 1.7 and a domain size of $5,192^3$. The table reports the memory allocated for different groups: the application, Infiniband buffers and the remaining memory allocated but not profiled.

Group	Memory Footprint (GB) aggregated on all nodes	% memory (w/o application)
application	1,096.10	
Infiniband recv buffers	19.53	34.23
Infiniband send buffers	0.63	1.10
other[1]	36.89	64.66

[1] *Other* regroups the memory allocated but not profiled such as thread stacks, intra-node structures, Infiniband endpoints and others structures allocated inside the runtime.

As a first optimization to reduce the memory footprint of these buffers, mainstream MPI runtimes usually rely on dynamic buffer allocations. An initial pool of buffers is allocated during `MPI_Init` and this pool is extended with additional buffers when needed [Hua+06]. Second, and as presented in section 2.2.5, Shared Receive Queue (SRQ) helps to reduce the memory usage of Infiniband receive queues since receive requests are shared between multiple receive queues. Because SRQ pools use fixed-sized buffers and may waste memory, Open MPI allocates several SRQ pools per process, each with a different buffer size. This approach called "Bucket SRQ" aims at providing a better receive buffer utilization [Shi+07].

Third, Koop *et al.* propose a technique called *message coalescing* where messages with the same envelope may be fused into the same network buffer [KJP07]. Moreover, Aumage *et al.* suggest a similar optimization in `NewMadeleine` where messages are accumulated while the NICs are busy [Aum+07]. As a result, the two latter approaches may both reduce the number of network buffers for short messages and increase the network bandwidth. These optimizations may however often increase network latency since transmission of messages is delayed. As for the *message coalescing*, it was mainly developed for the micro-benchmarks of communications since it imposes many restrictions on the message description (e.g., same

tag, same communicator) and does not always reflect the communication pattern of scientific applications.

Finally and for a better utilization of buffers over RDMA, Veloblock [KSP09] provides a mechanism to allow variable-sized memory buffers while the majority of implementations proposes a fixed-size version. The size of RDMA buffers consequently fits the size of the message to send, leading to a better memory usage.

3.2 Scalable Multi-Purpose Virtual Topology for High-Speed Networks

In the following section, the thesis proposes a scalable and fully-connected virtual topology for connection-based high-speed networks. In this topology, messages are transmitted using a routing protocol that requires the allocation of a few network endpoints. The contribution has been implemented inside MPC and supports TCP and Infiniband networks. Furthermore, we demonstrate the pertinence and the low-memory footprint of this work in a protocol for interconnecting MPI peers on demand.

3.2.1 Scalability Concerns of Connection-Oriented Networks

By definition, connection-oriented communications require a connection to be established before any data can be transferred. The difficulty here is to provide an alternative communication path to reliably exchange endpoint identifiers required to establish this connection.

As previously discussed in section 2.2.2, only the OpenFabrics stack provides a unified interface for RDMA devices to manage the connections of connection-based protocols (e.g., RC). RDMA Connection Manager (RDMA CM) is conceptually equivalent to a socket for RDMA communications. It provides a mode that reliably supports handshakes for connecting and disconnecting peers but unlike stream-based protocols such as TCP, communications are message-based. RDMA CM is however only portable to OFA devices including Infiniband, RoCE and iWarp, and prevents non-supported networks such as TCP to be used. In addition, it is restricted to QP connection and disconnection and does not allow users to develop their own communication protocols. To conclude on RDMA CM, it is even uncertain if the next-generation of connection-oriented networks will be supported by the interface: supporting a new connection-oriented network might require to totally rewrite the connection manager.

A virtual network defines a network which is designed in software and which is independent of the underlying interconnection network and its topology. Let us now envisage an alternative user-level virtual network that would permanently provide a reliable virtual communication path between all processes. First, this secondary network would offer a portable solution for managing endpoint connection and disconnection originating from the data network (i.e., MPI network). In addition, it would exhibit a unique interface to application developers for implementing their own protocols over a variety of underlying networks. Second, this network would be particularly interesting for supporting fault-tolerance algorithms. As presented in section 1.2.3.4, most fault-tolerant runtimes rely on keep-alive techniques where processes periodically emit *alive messages* to a manager which monitors the communication. This kind of communication is generally implemented using all-to-one

communications which naturally does not scale. In such a case, the signalization network would provide a reliable communication path to carry alive-messages to the monitor. Additionally, when a failure on the data network prevents any message to be exchanged, the secondary network would provide a convenient way for exchanging notifications between the two-disconnected processes.

Optionally, communications may fallback to this secondary network when an MPI function which requests a fully-connected graph is called. This communication pattern is commonplace in HPC applications since some collective algorithms require a direct connection between the source and the destination tasks. It includes for example (1) the *pairwise exchange* and the *linear* algorithms that are usually used in the `MPI_Alltoall` functions and (2) irregular operations where send count and displacements values are only known at the root task (e.g., `MPI_Gatherv`, `MPI_Scatterv`) [PG07]. Additionally, for some applications such as HERA from CEA [Jou05], the input dataset is opened by a single task (generally task 0) which then distributes it to other MPI tasks using an `MPI_Scatterv` operation. The MPI task 0 consequently establishes a connection to every other tasks during the initialization step and only a few of these links are later reused. As a consequence, these unused extra links remain in memory and could limit the scalability of the application.

Finally, this solution is even more efficient on systems embedding several HCAs or network ports: the primary (e.g., MPI communications) and the secondary networks could ideally be bound to two different HCAs.

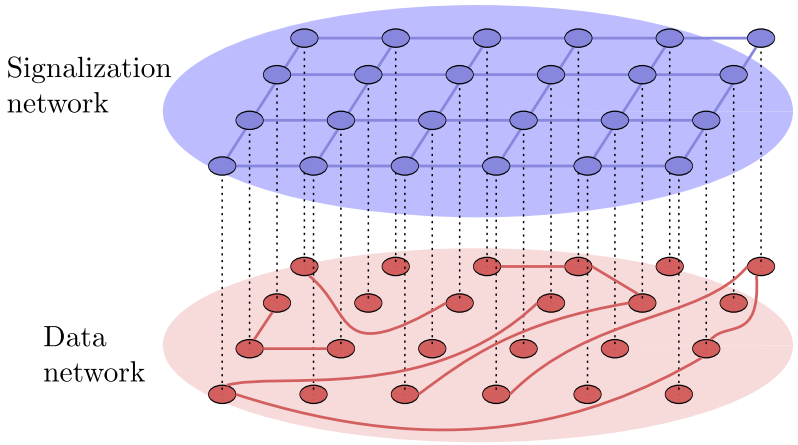


Figure 3.5 – Data and signalization networks. The data network is dedicated to MPI communications whereas the signalization network carries control messages (e.g., control messages used for on-demand endpoints interconnection)

In this contribution, we propose a flexible interface for exchanging control messages between not-connected MPI peers over connection-oriented networks. As depicted in figure 3.5, every process has a projection in both signalization and data networks. The signalization network establishes a fully-connected virtual topology where only a few connections are active, regardless of the underlying network. Whenever the data network needs to deliver a message to a not-connected process, the message is sent to the signalization network where it is routed to the final process. As a result, the signalization network allows any process to communicate with any other and only requires a low amount of memory to operate.

Topology	Degree	Average Distance
1D array	2	$N/3$
1D ring	2	$N/4$
2D mesh	4	$\frac{2}{3} \times N^{\frac{1}{2}}$
2D torus	4	$\frac{1}{2} \times N^{\frac{1}{2}}$
k-ary n-cube	$2n$	$\frac{n \times k}{4}$
Hypercube	$n = \log N$	n

Table 3.2 – Degree and average distance of several topologies. N is the total number of nodes in the graph.

3.2.2 Contribution: Scalable and Fully-Connected Signalization Topology for Connection-Oriented Networks

The signalization network allows to exchange point-to-point control messages across processes. Because latencies on this network are expected to be good, the virtual topology should support modern high-speed interconnects such as Infiniband. It should also be flexible enough for interfacing any network such as regular TCP/IP networks which does not rely on hardware-based features.

Furthermore, because the signalization network shares the same execution unit and the same memory than the data network, it should scale in terms of memory consumption regardless of the number of cores in the communication. As previously highlighted in section 3.1.1, the number of connections established is critical for scalability. Thus, the signalization network cannot rely on a fully-connected graph. Rather, it should keep active a low number of connections and provide an efficient routing algorithm. To finish, messages should be handled as soon as they are ready on the network card, even if the process is not in the communication library (i.e., executing a user function from the MPI application). Moreover, message reception should generate the minimum of extra-work to the CPU.

The torus topology is a n-dimensional grid network connected circularly where every node of the graph is also a network endpoint. According to the table 3.2, the average distance between two nodes of a torus graph (i.e., the number of hops) is $\frac{d \times k}{4}$ where d is the torus dimension and $k = n^{1/d}$ represents the number of elements along each dimension. An interesting feature of the torus topology is the ability to easily reduce the average distance by increasing the number of dimensions. A tradeoff must then be found between the number of dimensions and the degree of the graph (i.e., the number of connections per node). Additionally, the number of connections per node remains constant for a fixed dimension regardless of the number of nodes in the graph where each pair of processes is directly connected. Finally, the torus topology shows a good path diversity, i.e., the number of shortest paths between source/destination pairs. It allows a better load balancing in the network and provides several alternative routes to a same destination in case of a node failure.

Because the torus topology perfectly fits the requirements of the signalization network, we decided to choose this topology for the following contribution.

3.2.2.1 Multidimensional Torus Implementation

To bootstrap the signalization network, processes should be able to exchange endpoint identifiers to their neighborhood. The Process Manager Interface

(PMI) [Bal+10] provides a portable "key-value" database for parallel applications. A process can append information to the database and query information previously committed by other processes. In addition to "put" and "get" functions, it also provides a collective "fence".

Currently, both the Hydra process manager [Labc] and the SLURM resource allocator support PMI. Because of its proven reliability inside MPICH, a wide number of supported resource managers (such as SLURM, PBS, loadlevel, lfs, sge) and a portable implementation of the PMI-1 interface, we decided to rely on the Hydra process manager for developing our multi-purpose signalization network. Open-source PMI implementations are however subject to significant drawbacks which limit the scalability and performances. First, as far as we know, PMI implementations released as open-source all rely on UNIX TCP/IP sockets and lack to use native high-speed transport protocols like Infiniband. Second, the entire database is centralized in a unique process, leading to scalability issues.

Algorithm 1 Bootstrapping of the communication ring using the PMI interface

```
1: function CONNECT_RING
2:     prev = (rank + size − 1)%size
3:     next = (rank + 1)%size
4:     prev_ep = NEW_ENDPOINT()          ▷ e.g., QP for IB, Socket for Ethernet
5:     next_ep = NEW_ENDPOINT()
6:     PMI_KVS_PUT(key="rank : prev", input=prev_ep)
7:     PMI_BARRIER
8:     PMI_KVS_GET(key="next : rank", output=next_ep)
9:     PMI_BARRIER                       ▷ Communication ring established
10: end function
```

To address the two previously mentioned scalability concerns of the PMI interface, we propose a two-step virtual topology bootstrapping. As a first step, processes are spawned on compute nodes by the resource manager and they immediately join the same PMI Key-Value Space (KVS, i.e., name that identifies a database). A protocol described in algorithm 1 is then executed which aims at connecting each process on a ring. Endpoint identifiers (QP keys for Infiniband or the hostname/port for socket-based networks) are thus exchanged using PMI "put" and "get" operations. Because only the minimum number of links is created, the required number of entries in the database as well as the number of operations are significantly reduced. Once that minimalist network has been established between the p processes, each process requires a linear $O(p)$ time to communicate with any other process. As a second and final step, the higher-level torus topology is then bootstrapped using the previously established communication-ring and a routing algorithm based on the shortest distance.

3.2.2.2 MPI Connection Establishment over Infiniband

In the Infiniband specifications, each QP has a unique integer used for identification and called QP-ID. For reordering messages in connection-based protocols (e.g., RC), the two communicating QPs must be synchronized on the same Packet Sequence Numbers (PSN). Moreover, an additional unique Local Identifier (LID) is also registered in each HCA for network identification.

One way to establish a connection between two QPs is to exchange the QP keys (QP-ID, PSN and LID) and manually program the QP state transitions. As depicted in figure 3.6, implementing an on-demand QP connection over Infiniband usually requires a three-way connection protocol. Upon the first communication between the sender and the receiver, the sender sends its QP keys via a request for new connection to the receiver. At the end of the three-way exchange, both processes are finally interconnected and the communication can start.

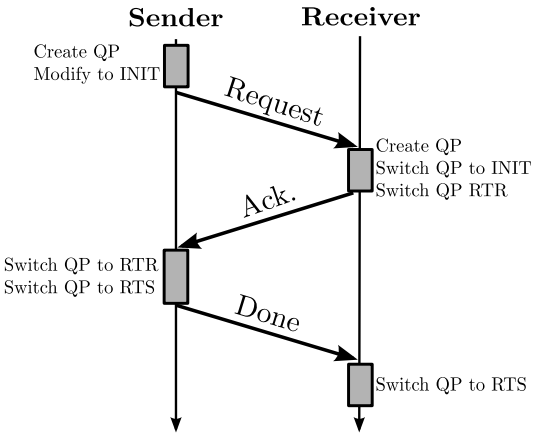


Figure 3.6 – On-demand QP connection algorithm over Infiniband. The RTR and RTS states respectively indicate that the QP is Ready To Receive and Ready To Send messages

To demonstrate either the efficiency and the scalability of the multi-purpose signalization network, we developed it inside MPC. MPC was a good candidate for this contribution since it implemented a not-scalable on-demand connection manager where control messages were exchanged over regular TCP/IP-based networks. We therefore modified the on-demand connection manager of MPC to use the signalization network for transmitting these control messages. As a consequence, on Infiniband networks, MPC now interconnects QPs using the same high-speed network than the MPI communications.

Latency and Memory Constraints over Infiniband Networks

To keep a small bounded latency, the signalization network relies on an event-driven message progression. As explained in section 2.2.4, this message progression allows the runtime to preempt any computation threads for retrieving messages from the HCA.

Because communications are less intensive on the signalization network, a low number of entries are posted to QPs, SRQs and CQs. Furthermore, the `rendezvous` protocol is disabled and the size of `eager` buffers is set to 1 KB because the signalization network mostly communicates short messages.

On-Demand Connections Evaluation

With the aim of evaluating the performance of peers interconnection over different MPI runtimes, we designed a benchmark where a root task sequentially sends a ping-pong message to each other nodes. Because MPI tasks are initially disconnected, the time for performing the ping-pong exchange includes the time required for connecting the two peers. From this value, the benchmark then subtracts the time required for performing a regular ping-pong test when both peers are already connected. Thus, we ensure that only the time required to establish the connection is measured. Additionally, one single MPI task is spawned per compute node since a few runtimes such as MPC create QPs at the granularity of the compute node.

Table 3.3 reports the average time to connect a peer using 128 compute nodes

from the *Thin Cluster*[1] among a variety of MPI implementations with on-demand connections enabled. We compare MPC implementing different dimensions of the torus topology (1D to 3D), Bullx MPI, MVAPICH2 and Intel MPI. Furthermore, the table prints the number of QPs allocated for the signalization network over the torus topology. Since Bullx MPI is derived from Open MPI, it also supports the *B-Buckets* optimization [Shi+07] and initially allocates multiple QPs per MPI task. With the purpose of fairly comparing runtimes together, we forced Bullx MPI to allocate a single QP per MPI task with the following runtime option: `-mca btl_openib_receive_queues S,65536,256,128,32`. Moreover, the UD capability of MVAPICH2 has been disabled to force the runtime to use of RC.

As expected, the time required to establish a connection decreases while the number of dimensions (and so, the number of links) increases. Furthermore, the 3D version of the torus topology only requires a negligible number of 6.36 additional QPs per compute node. One would notice that the number of QPs is slightly more than 6, which is actually the number of neighbors in a 3D torus. It is due to the additional endpoints that are created for connecting the communication-ring described in section 3.2.2.1. Indeed, some of these links connect some peers that do not belong to the torus neighborhood. Compared to the other runtimes, Infiniband peers connection over the 3D torus virtual topology gives competitive results and provides performance close to the regular on-demand protocols. These timings do however not exclusively reflect the time to exchange control messages required by peers interconnection. Indeed, lots of operations are performed during peers interconnection such as QP allocation/initialization and buffer creation. Furthermore, QP creation may significantly vary from a runtime to another according to the volume of buffers allocated and the number of WQEs into the QPs.

Runtimes	1D torus	2D torus	3D torus	Bullx MPI	MVAPICH2	Intel MPI
Time (μs) per connection	11,624	7,249	7,030	10,321	2,645	3,355
Speedup related to torus 3D	0.60	0.97	1.00	0.68	2.66	2.10
Number of QPs per node (signalization network)	2.00	4.19	6.36	No signalization network		

Table 3.3 – Average time to connect two peers over Infiniband using different MPI runtimes and 128-core compute nodes

3.2.3 Limit of the Design and Possible Enhancements

Designing a user-level virtual network often leads to performance issues because the software is involved in operations that the hardware normally handles. First of all, as the routing algorithm is implemented in software, each hop requires processes to actively participate to message forwarding. As a result, a task performing computation from the MPI application may be descheduled for retrieving messages from the signalization network. One way of investigation would be to offload the routing algorithm onto the hardware. On Infiniband clusters, the Mellanox CORE-*Direct* technology discussed in 2.1.4.2 provides a convenient solution for offloading collective communications onto the HCA [Gra+10; Kan+11] and could be used for

[1] The Thin Cluster is detailed in section 2.3.1

routing messages. Second, because the signalization network involves several hops for routing communications, it cannot leverage zero-copy RDMA operations from source to destination. Thus, it is not well-suited for sending large messages.

With the aim of extending our work, we propose to export the signalization network into a dedicated process, i.e., a process different than the one executing the MPI application. This extension would imply that a faulty MPI application would not prevent the signalization network to exchange messages. Second, a dedicated process would be convenient for plugging the signalization network under any MPI runtime or HPC application that would need a scalable communication path for exchanging messages. To finish, since the network topology affects communication performance [SVP13], a possible way for optimizing communications of the signalization network would be to map the virtual topology on the physical cluster topology [HS11a; Peñ+13].

3.3 Optimizing Network Endpoint Usage for Multi-Threaded Applications

As discussed in section 1.3.4, thread-based MPI runtimes conveniently allow to share some internal structures such as connection endpoints or network buffers for reducing the memory required by the runtime. This "everything-shared" model involving one unique set of network resources can however have an impact on performance compared to the "nothing-shared" approach: it increases the time spent in inter-thread synchronizations while accessing shared data structures and implies distant memory accesses in NUMA architectures (see section 1.3.6). In addition, some user-level network interfaces such as the `verbs` for Infiniband associate a single network endpoint with one unique HCA. As a result, one network endpoint per compute node prevents the use of multi-rail configurations where multiple HCAs are available per compute node. The runtime consequently has to allocate as many network endpoints as the number of HCAs and thus duplicate network resources over the compute node.

With the purpose of addressing the overheads due to multi-threading and overcoming the limitations imposed by one unique set of network endpoints, the thesis exposes a runtime technique for duplicating in software network-related resources across the compute node. This contribution proposes a hybrid design between the memory-friendly "everything-shared" strategy and the performances of the "nothing-shared" approach. We first explore several routing algorithms for transporting messages in multi-rail configurations. We then demonstrate the relevance of our contribution on compute nodes equipped with multiple HCAs. We finally show that our proposal can also benefit to single HCA configurations since it ensures the locality of network-related data in NUMA architectures.

3.3.1 Performance Implications of Multi-Threaded Endpoints

Because of the implicit shared-memory context between MPI tasks, thread-based MPI runtimes allow more flexibility for developing inter-node messaging protocols. Unlike process-based MPI runtimes where each MPI task has its own and private network endpoints (e.g., QP for Infiniband), thread-based MPI runtimes do not require a network endpoint to be set up for each remote task. In fact, a unique network endpoint can be shared among MPI tasks running in the same UNIX process. To clarify this point, let us consider two compute nodes referred to as A and B: tasks

0 and 1 are running in node A, tasks 2 and 3 in node B. In this configuration, the tasks 0 and 1 can communicate with the tasks 2 and 3 using the same network endpoint. By extension, a unique receive queue centralizes all incoming messages for all destination tasks in the compute node. As a result, when checking the completion of its messages, the task 0 is able to check and progress messages of the task 1 and *vice-versa*.

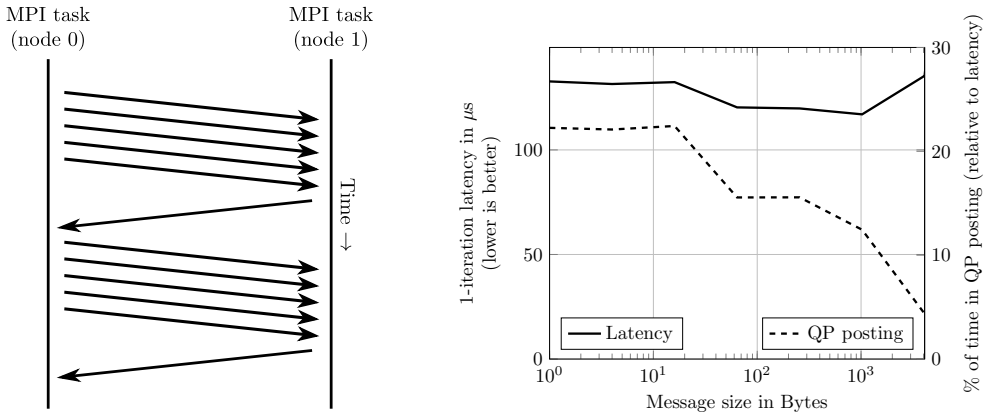
As highlighted in section 3.1.1, the number of connections is critical for scalability. In theory, thread-based runtimes can achieve better scalability on multi-core systems as they decrease the number of endpoints in proportion to the number of cores per compute node. First, and besides memory aspects, reducing the number of endpoints may improve overall performance. Indeed, because one endpoint centralizes all communications from one compute nodes, the progression function consequently requires to poll less network endpoints. Second, Infiniband HCAs provide an embedded memory for caching information of a limited number of endpoints (less than 128 QPs for a Mellanox MT25218 HCA [Sur+05], shared across processes on the node). When the HCA attempts to read the information related to an endpoint that is not cached, it results in a significant latency penalty because the HCA must access the host memory. To illustrate the latter point, Sur *et al.* show that transferring messages shorter than 1 KB to a QP that is not cached nearly doubles the latency [Sur+05]. Consequently, thread-based MPI runtimes push back the limits of this cache since the same number of QPs can actually address more MPI tasks than for process-based runtimes.

Naturally, sharing low-level network structures in a multi-threaded context requires locks to protect network structures from concurrent accesses to the same resource [Luo+11]. It means for example that several tasks cannot simultaneously post a message to the same endpoint or poll the same completion queue. To illustrate this point, figure 3.7 reports the results of an MPI bandwidth test on 2 nodes where all the 16 tasks in the first node send messages back and forth to the remote tasks located in the second node. In addition, one unique QP is allocated per compute node. As we can see in the figure, the contention on the QP for posting new buffers participates to the high latency of short messages since QP posting represents more than 20% of the total latency for 1-byte messages. As soon as the overhead in QP posting falls, the latency decreases from 133 to 117 μs. At 1 KB, the latency starts to increase since messages are getting larger.

Another drawback of having only one network endpoint per compute node concerns the maximum number of entries an endpoint can handle. For example with Infiniband, due to hardware restrictions, the HCAs limit the number of Send Requests that can be posted to the same QP. With ConnectX-2 HCAs, this limit is set to 15,000 entries. In other words, no further entry can be posted while the QP is full and the MPI tasks accessing the QP have to wait a free entry when such a case happens. As a consequence when dealing with large compute nodes, having only one endpoint per node may be insufficient to efficiently carry all messaging traffic.

To finish, with the increasing number of cores per compute node, one workaround commonly considered is to equip compute nodes with several HCAs since multiple HCAs scale better with the number of cores. Some network manufacturers such as Mellanox even propose solutions which regroup several ports on the same HCA. One example of this trend is the Curie's *Large Cluster*[2] which is composed of 4 physi-

[2]The Large Cluster is detailed in section 2.3.3



(a) Algorithm of the bandwidth test

(b) Latency of one iteration and QP posting overhead

Figure 3.7 – MPI Bandwidth test on 2 compute nodes and 16 MPI tasks per node using the `eager` protocol. MPI tasks perform pairwise communications with a task from the other node following the pattern depicted in figure (a). In figure (b), performance of short messages is low, partially because of a high contention on the QPs while posting new network buffers.

cally distinct HCAs. The aggregation of multiple HCAs has been demonstrated to be a relevant solution for breaking-up the network bandwidth limitation due to the increase in the number of cores inside compute nodes [CRS09; LVP04]. According to the Infiniband specifications, it is however impossible to configure an endpoint to share several HCAs and one distinct endpoint must consequently be created for each HCA. In this case, one unique endpoint would definitely waste the parallelism potential of multiple HCAs since one single HCA would be used. Moreover, previous research teams have also demonstrated that the Infiniband hardware is able to process multiple send requests in parallel [Fri+07] and using one unique QP per node would waste this HCA parallelism.

In the following section, we present a flexible and multi-threaded communication layer that replicates network structures in the NUMA nodes. The contribution accelerates MPI communications since it enhances data locality and easily leverages the parallelism potential of multi-HCA configurations. Moreover, it reduces the memory allocated since fewer network endpoints are required compared to a process-based approach.

3.3.2 Contribution: Multi-Threaded Virtual Rails

For the purpose of this contribution, the thesis refers to "virtual rail" (or *vrail*) as an abstract representation of the network resources used for communicating messages between processes. In concrete terms, a *vrail* is composed as follows:

- a **configuration**: describes the configuration of the *vrail* such as the number of buffers or their size. This configuration takes the form of an XML file and users may write their own configuration files.
- a **device**: defines which device (Infiniband HCA) and which port to open.
- **network structures**: QPs, SRQ and CQ for Infiniband.
- a pool of send and receive **network buffers**.

- a **routing protocol**: defines what are the conditions to use the *vrail*. This point is detailed in the next section.

The *vrail* model is designed to be as modular as possible. It means that depending on the hardware available in the compute nodes (i.e., number of cores, memory available), MPI users may easily configure the communication layer to stack several *vrails*. There are numerous different purposes for which stacking multiple *vrails* is interesting. MPI users could for example consider creating multiple *vrails*, each with a different size of buffers and select the *vrail* that best fits the message size being sent. This proposition similar to the Open MPI's "B-Buckets" described in section 3.1.2.4 would aim to better use buffers and save memory.

The following contribution focuses on *multi-rail* configurations where multiple HCAs or ports are available in the compute node to communicate. More precisely, an application supports multi-rail configurations only if (1) several independent communication paths are available between two compute nodes and (2) these communication paths are accessible in the same UNIX process. To clarify this point, we do not consider configurations where several processes of the same compute node open a distinct HCA. In this case, all HCAs are used at the compute node level but only one HCA is opened per process and the application does not support multi-rail configurations.

3.3.2.1 Multiple HCAs Configurations

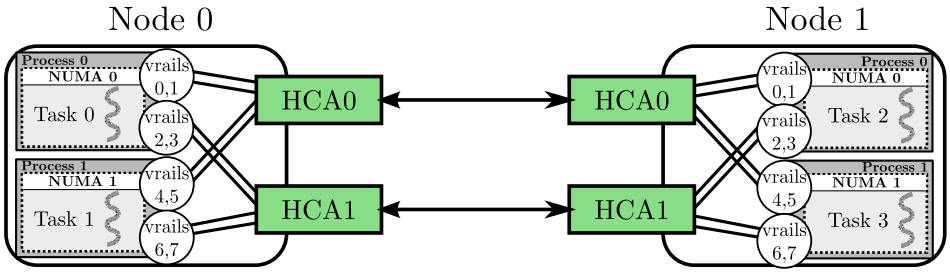
In regular HPC systems, Hyper Transport (AMD), QPI (Intel), BCS (Bull) are some technologies which allow multiple processor sockets to share the same memory (see section 1.1). These technologies introduce new effects on network communications such as Non-Uniform Input/Output Access (NUIOA) where the network performance varies according to the locality of memory banks that are being accessed [MG+07]. Certainly one of the most representative examples of this trend is the compute nodes that compose the Curie's *Large Cluster* and that count 128 cores in 4 groups of 4 NUMA nodes. As presented in section 2.3.3, the compute nodes consist of four HCAs in total and each level-2 NUMA node is topologically close to one physically distinct HCA. As a consequence, accessing a distant HCA induces communication penalties because of the high NUMA factor induced by the BCS.

Multi-rail configurations have been widely studied in the literature and many policies have already been investigated for parallelizing communications such as:

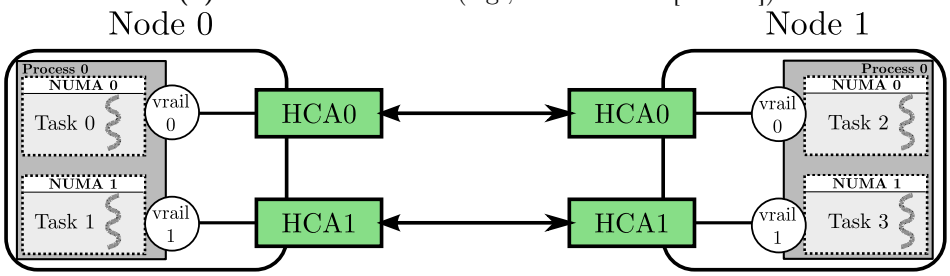
1. *round-robin*: the HCA to use is selected according to a circular order over all available HCAs;

2. weighted and not weighted *message stripping*: the message is stripped into multiple segments and each segment is sent through a different HCA. The stripping process may also adjust the size of each segment according to the NUIOA effects of the underlying hardware (weighted policy) or not [MGN10; LVP04];

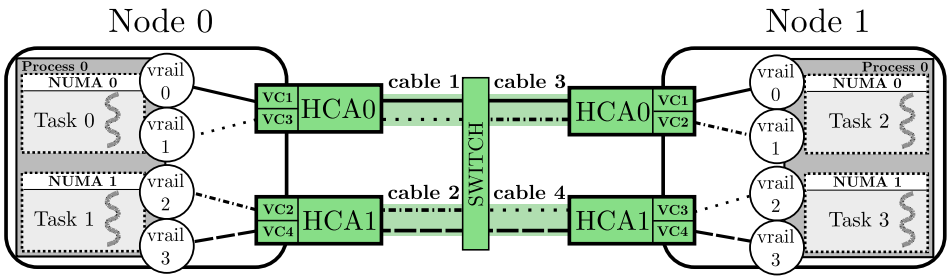
3. *idle-based* routing: the runtime selects an idle HCA for the transmission [Aum+07].



(a) Process-based MPI (e.g., MVAPICH2 [LVP04])



(b) Multi-threaded MPI runtime and few NUMA effects



(c) Multi-threaded MPI runtime with logical cross-links between HCAs and without NUMA effects

Figure 3.8 – Estimation of the number of *vrails* (or *virtual subchannels*) in a multi-rail configuration for a process-based runtime (a) and a multi-threaded MPI runtime (b and c). In (a), each MPI task locally opens four *vrails* (number of remote tasks multiplied by the number of HCAs) and the runtime prevents the *vrails* to be shared between the MPI tasks. The design depicted in (b) requires one unique *vrail* per HCA and per compute node to utilize the total network bandwidth available. In (c) the runtime connects HCA0 and HCA1 of the two compute nodes using two logical cross-links and referred to as *Virtual Channels* (VC). Additionally, this design requires per NUMA node as many *vrails* as the number of HCAs in the compute node.

In some aspects, our *vrail* design is close to the concept of *virtual subchannels* described in [LVP04] where the MPI tasks can open several HCAs (or ports) to communicate. This design depicted in figure 3.8(a) creates a set of *virtual subchannels* (referred to as *vrail* in the figure) per MPI task and there is no way for an MPI task to access a *virtual subchannel* from another task.

In multi-threaded MPI runtimes, all *vrails* can be accessed by any MPI tasks running in the same node. Figure 3.8(b) creates one unique *vrail* per HCA and per compute node and the design utilizes the total network bandwidth available. Thus, whenever the tasks 0 or 1 send a message, they can choose between both *vrail* 0 and *vrail* 1. As a consequence, our approach reduces the number of endpoints by

a factor equals to the square of the number of cores per node compared to the approach depicted in figure 3.8(a). The major drawback of this design occurs when the remote destination MPI task is located in a NUMA node ID different than the one where the source task is running in. Let us consider that every task depicted in figure 3.8 runs in a different NUMA node and that task 0 sends a message to task 3. Moreover, communication buffers are allocated in the same NUMA node than the one where the task is executing. In such a case, whatever the HCA used for the communication (HCA0 or HCA1), it will require a distant memory access to transmit the data. It has however to be noticed that this overhead only affects the communication performed by the HCA and does not involve the CPU.

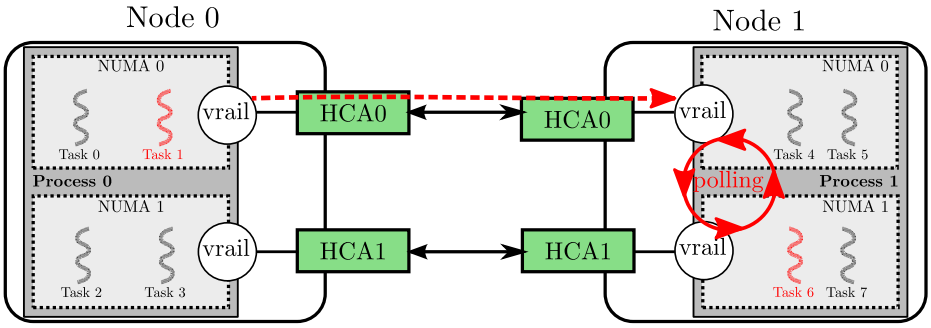
In a shared-memory context, an alternative design is presented in figure 3.8(c) where the runtime asymmetrically interconnects compute nodes using cross-links between the HCAs. This approach does actually not involve the NUMA effects highlighted in figure 3.8(b) but it requires per NUMA node as many *vrails* as the number of HCAs in the compute node. Additionally, since there is a unique physical Infiniband cable between an HCA and its network switch, symmetrically (e.g., HCA0–HCA0) and asymmetrical virtual channels (VC) are not independent. As a consequence, the bandwidth is shared between the two VCs and no speedup is expected from two communications concurrently operating in two VCs that share the same HCA.

The section 3.3.1 has previously established that one *vrail* per compute node gives the lowest memory consumption. However, it has also been demonstrated that this design may adversely affect the performance. Because a single *vrail* allocated per HCA appears to have a balanced trade-off between performance and memory consumption, the following thesis focuses on the design depicted in figure 3.8(b). Our motivations for such a configuration are threefold. First, the contribution increases the network bandwidth of MPI communications since all HCAs available in the compute nodes are used. Second, the runtime can improve locality in NUMA architectures since the data from a *vrail* are accessed locally. Finally, the contribution significantly reduces the number of endpoints (so the memory consumption) since one unique *vrail* is allocated per NUMA node where the majority of MPI runtimes open as many endpoints as MPI tasks. In the next section, we detail two routing strategies used for selecting which *vrail* is the most appropriate to use for an MPI message being sent.

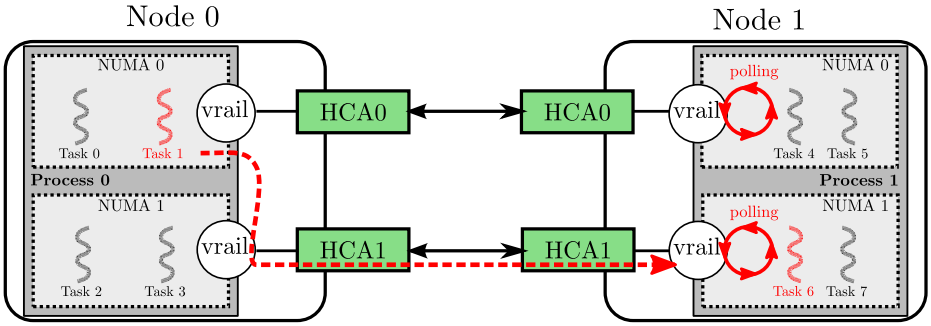
3.3.2.2 Sender-Driven and Receiver-Driven Routing Strategies

A routing strategy defines how a *vrail* is elected to transmit a message to a remote compute node. Two different routing strategies are possible for a *vrail* selection: on the one hand, the sender-driven routing policy is based on the location of the sender (figure 3.9(a)) and on the other hand, the receiver-driven policy focuses on the location of the receiver (figure 3.9(b)).

The sender-driven routing policy depicted in figure 3.9(a) is the easiest technique to implement. Whenever the MPI task 1 sends a message, the *vrail* from the closest NUMA node is chosen to transmit the message. At the receiver side, the MPI task 6 polls all the available *vrails* as it can potentially receive a message from any CQ from any *vrail*. The main advantage of a sender-driven routing policy is the opportunity to optimize the access to the same *vrail* when several threads attempt to send a message. Indeed, as the *vrail* is local to the NUMA node, posting a message to



(a) Sender-Driven Routing Policy



(b) Receiver-Driven Routing Policy

Figure 3.9 – Comparison of two routing policies for selecting a *vrail*: the sender-driven (a) and the receiver-driven (b). On both figures, the MPI task 1 sends a message to the MPI task 6.

a NUMA-aware *vrail* should provide better results than accessing a *vrail* from a remote NUMA node. While waiting for a message, the receiver task has however to poll all the *vrails* in order to guarantee the proper progression of all outstanding MPI messages. On large NUMA nodes with a large number of cores, this technique will definitely not scale since the time required to progress messages linearly increases with the number of *vrails*. In addition, message progression requires repetitive accesses to CQs and their corresponding locks. It would consequently generate a large traffic in the NUMA interconnect.

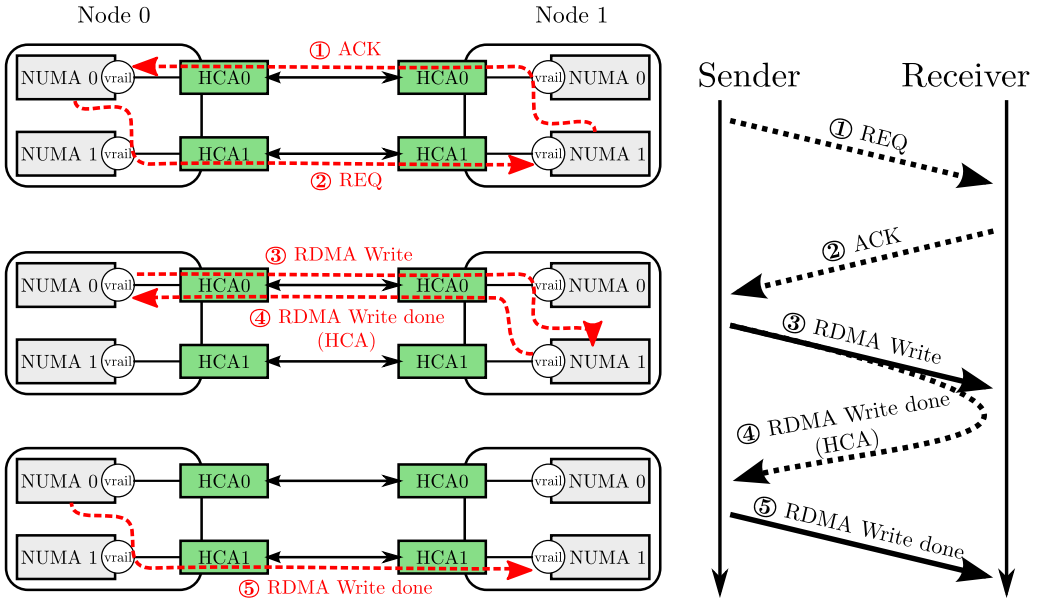
The receiver-driven routing policy is much more complicated to implement than the one previously described. For each message being sent, the sender task determines where the remote task is executing and consequently selects the proper *vrail* that delivers the message to the NUMA node where the distant task is running in. As an illustration of this strategy, MPI task 1 in figure 3.9(b) chooses to communicate through HCA1 because task 6 is running in NUMA node 1. Contrary to the sender-driven policy, the sender may potentially access a remote *vrail* located in a different NUMA node. At the receiver side, MPI task 6 is however guaranteed to receive all its messages through the local *vrail* connected to HCA1. Thus, an MPI task only requires to poll the closest *vrail* for progressing messages and there is no memory access to a remote *vrail* (i.e., to a remote NUMA node) while polling.

Since the sender-driven policy has been previously discarded because non-scalable, we only consider the receiver-driven routing policy. In the next section, we discuss the design of a `rendezvous` protocol with a receiver-driven routing policy when using multiple *vrails*.

3.3.2.3 `rendezvous` Protocol with the Receiver-Driven Routing Policy

Unlike the `eager` protocol, `rendezvous` protocols involve control messages to synchronize the two MPI peers. In a multi-*vrails* context, efforts must focus on ensuring that control messages and RDMA operations are delivered using the proper *vrails*. Indeed, if a message is sent through a wrong *vrail*, the remote task would never be notified that a network message is pending in this *vrail*.

Let us now consider the configuration depicted in figure 3.11(a) and which regroups two compute nodes. Each compute node is composed of two NUMA nodes attached with two HCAs and connected as follows: NUMA 0 and NUMA 1 are respectively connected to HCA0 and HCA1. For routing MPI message across the different *vrails*, we assume the receive-driven policy where the receiver always pools the *vrail* local to the NUMA node it belongs to. According to this routing policy, NUMA 0 only polls the *vrail* connected to HCA0 and does not access the *vrail* bound to HCA1.



(a) `rendezvous` *vrail* selection

(b) `rendezvous` message flow-graph

Figure 3.11 – `rendezvous` message transmission from a task running in node 0/NUMA 0 to a task running in node 1/NUMA 1 with a receive-driven routing policy. The selection of the *vrail* is determined according to the type of the message to transfer and the location of the destination task (receiver-driven routing policy).

In order to highlight what are the challenges when developing a `rendezvous` protocol in a multi-*vrail* context, we examine the following statement: the MPI task running in NUMA node 0 attempts to send a message to a task located in the NUMA node 1 using the 5-step `rendezvous` protocol described in figure 3.11(b). In such a configuration, the REQ and the ACK control messages are respectively delivered using ① HCA1 and ② HCA0. In ③, the RDMA write operation is initiated by HCA0 while one would expect HCA1 to be used. In fact, when a memory operation is completed, an acknowledgment referred to as step ④ is automatically returned by the remote HCA to the HCA which has initiated the RDMA operation (see section 2.2.1.2). Moreover, the Infiniband specifications do not allow this acknowl-

edgment to be returned to another HCA (e.g., HCA1 posts the RDMA operation and the acknowledgment is received via HCA0).

The Infiniband standard specifications defines a convenient capability where the HCA may automatically generate an immediate value to notify the remote HCA that an RDMA operation has completed. In a receive-driven configuration, this mechanism cannot however be used to notify the receiver that a `rendezvous` message is done. Indeed, in our example, the immediate value would be returned to the wrong CQ (so to the wrong *vrail*) and the receiver would never be notified that the `rendezvous` message is completed. Instead, we disabled the immediate value and in ⑤, the sender manually sends the *done* message to the proper *vrail*. As a consequence, each MPI task only requires to pool the *vrail* associated with its NUMA node for being notified of a completed message and thus, regardless of the protocol used (`eager`, `buffered` or `rendezvous`).

3.3.2.4 Implementation and Locality Concerns

As presented in section 1.3.6, locality while accessing data is primordial for achieving performance and it is the runtime's responsibility to ensure that the data related to a *vrail* (buffers, network structures) are allocated in the right NUMA node during initialization.

We developed our modular multi-threaded communication layer inside MPC because, as presented in section 1.2.3.4, it is a thread-based MPI runtime that efficiently implements a high-speed inter-process communication layer over Infiniband. During the initialization of MPC, one thread is responsible for allocating each *vrail* one by one. As it is, the issue is that every *vrail* is allocated in the same NUMA node because of the first-touch policy. To address this locality issue and before initializing a *vrail*, the runtime determines according to the configuration provided by the user in which NUMA node the *vrail* shall be initialized. The initialization thread is then bound to a CPU belonging to the target NUMA node using the `hwloc` library [Bro+10b]. Finally, the network structures are then initialized. In this way, the runtime ensures that all data related to a *vrail* are allocated in the right NUMA node.

3.3.3 Multi-Threaded Network Buffers Management

Network buffers are the smallest entities used for communicating data between a pair of MPI tasks. They are involved in every communication protocol, from the `eager` to the `rendezvous` protocols, including pipelines. Since they are frequently accessed by the runtime, it is easy to understand why it is important to optimize their usage for achieving efficient communications. In the following section, we describe the techniques that have been designed to optimize the utilization of network buffers in NUMA architectures. For ease of reading, this section denotes as `ibuf` a network buffer over Infiniband.

Send and receive network buffers (`ibufs`) are usually designed as a centralized FIFO list where *push* and *pop* manipulations are protected using locks. A centralized design does however not take into account the non-uniform memory accesses of NUMA architectures.

As a first optimization to provide an efficient `ibuf` management on these architectures, we designed one list of send `ibufs` per NUMA node. First and most importantly, this approach aims at reducing the contention on the list because (a)

it is replicated in all NUMA nodes and (2) the lock protecting the list during manipulations is only accessed by local threads. As a result of the second point, the lock latency is reduced since the NUMA interconnect (e.g., QPI or BCS) is not involved. Second, prior to message sending, the `eager` protocol requires to copy the user buffer to the `ibuf`. This operation is faster if the memory pages belonging to the `ibuf` are allocated in the same NUMA node than the one where the MPI task is running in.

The second optimization focuses on `ibufs` dedicated to message reception. Contrary to send `ibufs`, we designed the poll of receive `ibufs` as a single global list shared across the compute node. Indeed, a unique SRQ is allocated per *vrail* and using only one thread is sufficient for posting `ibufs` to the SRQ. Moreover, the runtime determines with `hwloc` which NUMA node is the closest to the HCA to open and subsequently allocates receive `ibufs` in this NUMA node. Once the number of remaining `ibufs` posted to the SRQ is running low, a task is in charge of re-filling the SRQ with `ibufs` from the list. This verification is actually carried out when a receive `ibuf` is released to the global list. To prevent a heavy contention on the global list, MPI tasks do not directly release their free receive `ibufs` to the global list. Instead, the receive `ibufs` to free are cached into a queue local to the NUMA node where the task is scheduled. At this point, it is important to point out that cached receive `ibufs` cannot receive network messages. When the queue reaches the maximum number of receive buffers accepted in cache, the cache is merged with the global receive buffer queue and these buffers can then be posted to the SRQ.

Lastly, it should be noted that one pool of send `ibufs` is allocated per NUMA node and per *vrail*. Furthermore with the aim of saving memory, send and receive pools are initially created with a minimum of entries and they dynamically enlarge during the execution.

3.3.3.1 NUMA-Aware `ibufs` Micro Evaluation

To handily demonstrate the performance of the NUMA-aware `ibufs` introduced in section 3.3.3, figure 3.12 presents some results extracted from the IMB Exchange benchmark on 2 compute nodes of the *Thin Cluster*[3]. In this benchmark, each task t sends and receives data to/from both left $(t-1)$ and right $(t+1)$ neighbors and the time reported per iteration includes the four messages. Additionally, an MPI task and its two neighbors are running in two different compute nodes and the benchmark exclusively communicates using Infiniband (IMB argument `-map`). The version of MPC without buffer locality is compared to the the version of MPC that implements the previously described NUMA optimizations for managing `ibufs` (denoted as "Opt. `ibufs` " in the figure). The figure reports the average latency of 1-iteration and the MPC runtime was modified to report the time spent inside locks protecting *push* and *pop* operations on `ibuf` lists as well as the time in locked regions (e.g., it includes the time spent while posting receive buffers to the SRQ).

The figure shows that NUMA-aware `ibufs` decrease the latency from 82 μs to 64 μs for `eager` messages up to 1 KB. Compared to the regular `ibuf` implementation, this performance increase with NUMA-aware `ibufs` is due to the large amount of time saved in locks protecting `ibuf` lists. Indeed, the time spent in locks and regions falls from 32 μs to 5 μs while the same work is performed inside locked regions between both versions. From 1 KB, we observe a reduction of the contention

[3]The Thin Cluster is detailed in section 2.3.1

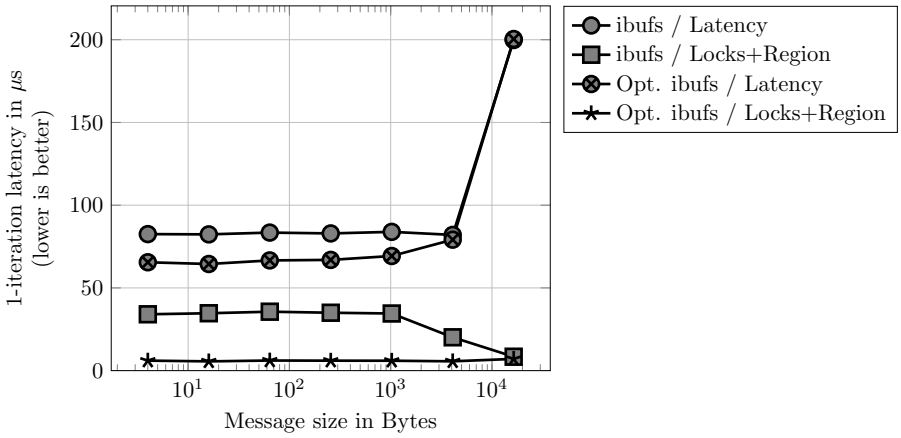


Figure 3.12 – IMB Exchange benchmark on 2 nodes, 16 MPI tasks per node with the `eager` protocol. The tasks communicate exclusively using the network (IMB argument `-map 16x2`). The latency is relative to 1 iteration.

on locks and both `ibuf` implementations perform the same way. As a conclusion to these results, a locality-aware management of network buffers is primordial for achieving performance for messages up to 1 KB, even with architectures exhibiting a few NUMA nodes (2 in this case).

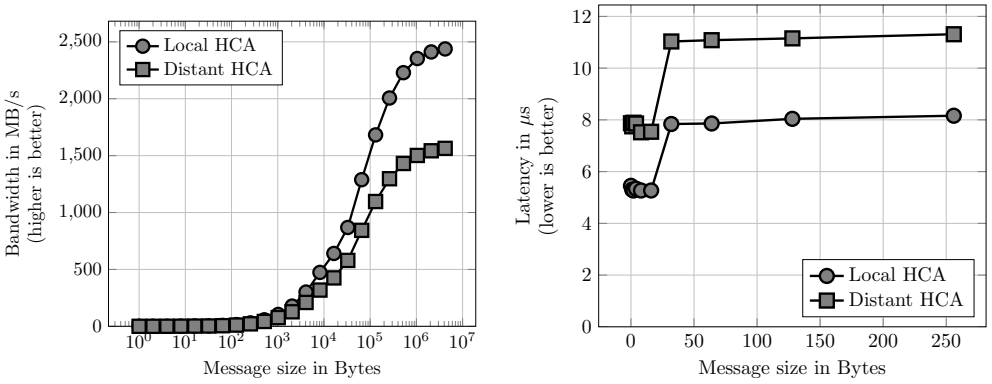
3.3.4 Evaluation of the Design

In the following section, we evaluate the performance and the impact on the memory consumption of our multi-*vrail* design described in section 4.4.2. First, we examine how much the BCS impacts the performance of network communications. Second, we present the relevance of our contribution on a micro-benchmark involving AllToAll communications and extracted from the Intel MPI Benchmarks suite. Finally, we perform scalability experiments on Athena, a real-world scientific application [Sto+08].

3.3.4.1 Bull Coherence Switch and Network Communications

As a first evaluation, we propose to analyze the impact of the BCS on network communications. Figure 3.13 presents the results from an IMB Ping-Pong benchmark on 2 nodes from the *Large Cluster* according to the location of the HCA being used. *Local HCA* version indicates that the MPI tasks communicate using the closest HCA whereas Infiniband transfers of the *Distant HCA* configurations traverse the BCS. Figure 3.13(a) reports the bandwidth achieved for messages up to 4 MB and figure 3.13(b) focuses on latency of messages shorter than 256 bytes. According to the results, it is clear that the BCS significantly degrades network performance. Indeed, due to the NUIOA effects introduced in section 3.3.2.1, the maximum bandwidth is divided by 1.6 and the BCS adds 3 μs to communication latencies when using the *Distant HCA* compared to the *Local HCA*. These measures show that locality is primordial for achieving performance on such a system.

As a second experiment, we evaluate the impact on communications of several threads accessing the BCS while a network transfer is concurrently operating in this bus. To do so, we developed a micro-benchmark that is schematically represented in figure 3.14. The IMB Ping-Pong test is executed on 2 nodes, 1 MPI task per



(a) Bandwidth **(b)** Latency

Figure 3.13 – IMB Ping-Pong benchmark on 2 nodes from the *Large Cluster*, 1 MPI task per node according to the HCA used (*local* or *distant*). In figure (a), the bandwidth is represented for messages up to 4 MB whereas the latency for messages shorter than 256 bytes is represented in figure (b).

node and the runtime is configured to use a distant HCA. While the Ping-Pong benchmark is executing, additional threads are created and each of them performs large memory copies (128 MB) across level-1 NUMA nodes. As a consequence, these threads generate memory traffic in the BCS. Figure 3.14 illustrates the design of the experiment with 4 threads: 4 memory copies are operated from the NUMA nodes 4 to 0, 5 to 1, 6 to 2 and 7 to 3, respectively by threads in the NUMA nodes 0, 1, 2 and 3. Moreover and concurrently to these copies, a thread in the NUMA node 4 is performing a communication with HCA0.

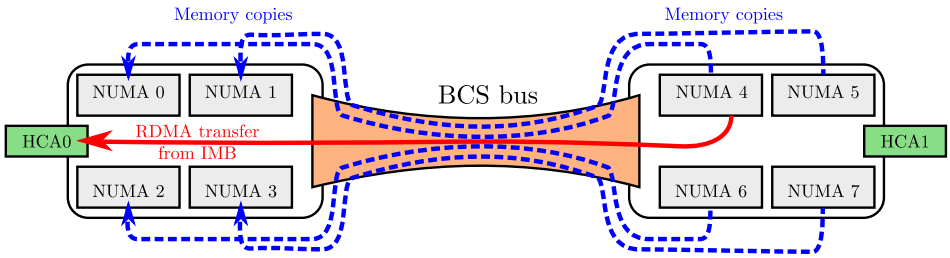
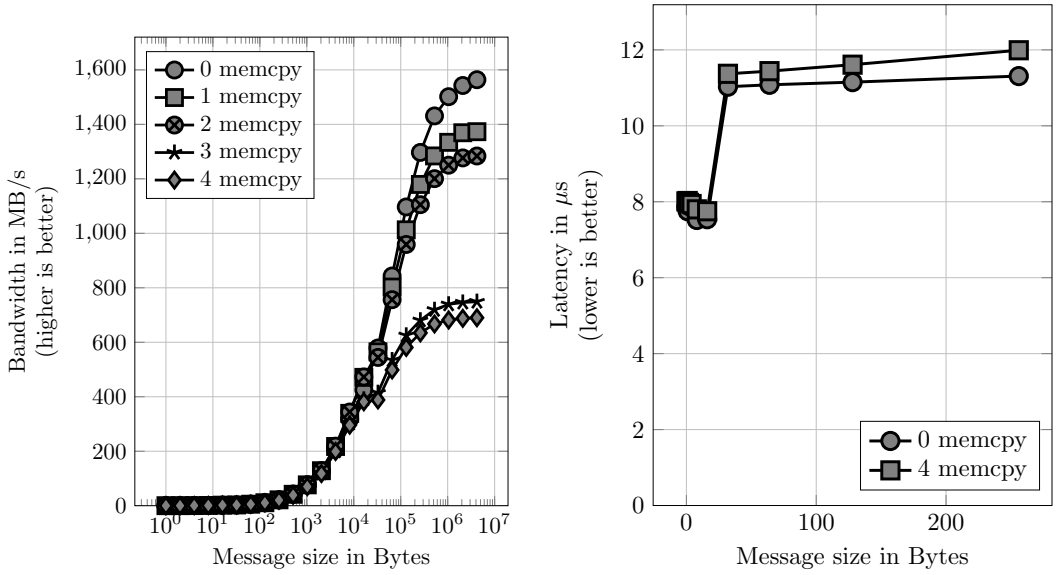


Figure 3.14 – Schematic representation of the benchmark that evaluates the impact of BCS on network communications

The results presented in figure 3.15 are decomposed into (a) bandwidth for messages up to 4 MB and (b) latencies for messages shorter than 256 bytes. Each curve defines a different number of memory copies that are performed in parallel. From 0 to 4 simultaneous memory copies, the network bandwidth falls from 1,564 MB to 690 MB. Furthermore, a slight overhead of 0.7 μs is noticeable for messages up to 256 bytes. To determine if the BCS is the unique component responsible for the performance degradation, we reproduced the same experiment but using the closest HCA so that Infiniband transfers are not traversing the BCS. In such a configuration, no overhead on network communication was observed. This experiment shows that it is primordial to limit the access to remote NUMA nodes to preserve the BCS bandwidth.



(a) Bandwidth **(b)** Latency

Figure 3.15 – IMB Ping-Pong benchmark on 2 nodes from the *Large Cluster*, 1 MPI task per node. A varying number of threads perform memory copies that are concurrently operating to network transfers. In figure (a), the bandwidth is represented for messages up to 4 MB whereas the latency for messages smaller than 256 bytes is represented in figure (b).

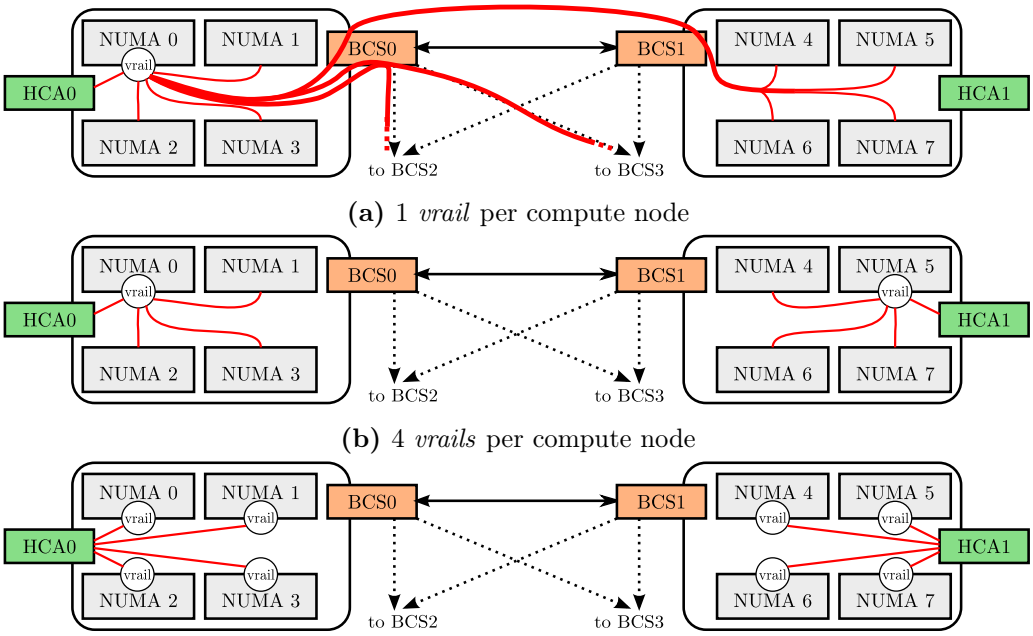
3.3.4.2 Multi-*vrails* Evaluation on Micro-Benchmark

To evaluate the performance as well as the impact on the memory consumption of multiple *vrail* configurations, we choose the AllToAll micro-benchmark from the Intel MPI Benchmarks Suite. This test-case performs `MPI_Alltoall` communications which require every task to communicate a unique set of data to the others, exhibiting a fully-connected graph. Experiments are conducted on the *Large Cluster* because it provides a large number of cores and compute nodes are equipped with 4 Infiniband HCAs. We set up three different *vrail* configurations as presented in figure 3.17 and described as follows. The first configuration in figure 3.17(a) allocates one unique *vrail* shared between 128 cores forming the compute node. In this case, the runtime uses one HCA in four available and the total network bandwidth is low. In the second configuration in figure 3.17(b), one *vrail* is created per level-2 NUMA node for a total of 4 *vrails* per compute node. In the last configuration in figure 3.17(c), one *vrail* is created per level-1 NUMA node for a total of 16 *vrails* per compute node.

As a first group of experiments, we set the configurations depicted in figures 3.17(b) and 3.17(c) to use the 4 available HCAs. Then, a second group of experiments examines how multiple *vrails* can improve the performance when only one HCA is used in the node.

Multiple HCAs evaluation

Figure 3.18(a) compares the execution times obtained on MPC with 1 *vrail* per compute node, 1 *vrail* per level-2 NUMA node (4 *vrails*) and 1 *vrail* per level-1 NUMA node (16 *vrails*) to Intel MPI in DAPL mode using Reliable Connection and Adaptive MPI (AMPI is a thread-based implementation of MPI on top of the



(a) 1 *vrail* per compute node

(b) 4 *vrails* per compute node

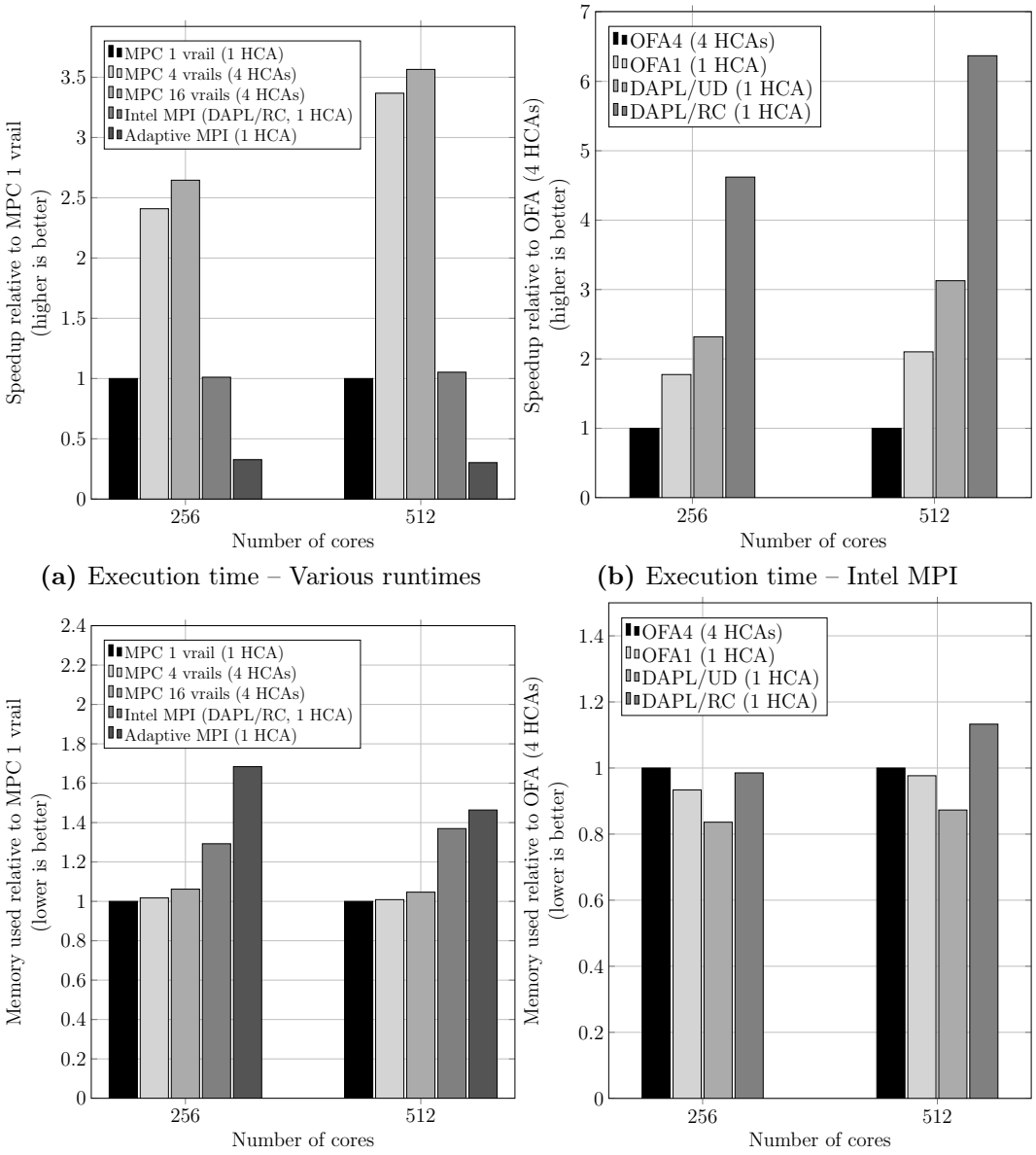
(c) 16 *vrails* per compute node

Figure 3.17 – Three possible configurations for the multi-threaded communication layer on 128-core nodes. One *vrail* is allocated and 1 HCA is used in figure (a). For polling and posting messages, every NUMA node accesses the same *vrail*, resulting in a large traffic on the BCS. In figure (b), one *vrail* is allocated per level-2 NUMA node and in figure (c), one *vrail* is allocated per level-1 NUMA node. For figures (b) and (c), the closest HCA is opened and no access through the BCS is required to poll the *vrails*

Charm++ runtime system [KK93]). Moreover, the evaluation was conducted on the IMB AllToAll micro-benchmark and for 1 MB messages. All the results are relative to MPC with 1 *vrail*. Bullx MPI does not appear in the figure because, in this case, the benchmark deadlocks with messages larger than 256 bytes with multi-node configurations. Furthermore, MVAPICH2 is not represented in these results as it fails to initialize on the machine. Concerning Adaptive MPI, the runtime was compiled with SMP and IBVERBS supports and the communication thread was disabled through the `+CmiNoProcForComThread` argument because we faced performance issues as long as it was activated.

From 256 to 512 cores, the speedup of multi-*vrail* configurations compared to 1 *vrail* significantly increases. This can be explained by the fact that they are three times more inter-node messages from 256 to 512 tasks, resulting in a heavy contention on the unique *vrail*. On 512 cores, MPC with 16 *vrails* exhibits a speedup of 3.5 compared to the version with 1 *vrail*. Additionally, MPC with 16 *vrails* gets a slight performance gain over the 4-*vrail* configuration. We believe this gain is due to the *vrail* structures that are accessed locally by MPI tasks running in the same level-1 NUMA node. Indeed, an in-depth study has demonstrated that the mean access time to CQs is dramatically reduced in a 16 *vrail* configuration compared to the versions with 1 or 4 *vrail* (s).

Compared to other MPI runtimes, the multi-*vrail* versions of MPC show performance improvement between 2x and 3.5x. The poor performance of Adaptive MPI can be explained by two factors. First, Adaptive MPI achieves low performance on



(a) Execution time – Various runtimes

(b) Execution time – Intel MPI

(c) Physical memory footprint per core – Various runtimes

(d) Physical memory footprint per core – Intel MPI

Figure 3.18 – IMB AllToAll micro-benchmark evaluation up to 512 cores (4 nodes) with 4 HCAs and 1 MB messages. Comparison of execution time and memory used between MPC with a various number of *vrails*, Intel MPI using different configurations and Adaptive MPI (AMPI). MPC with 1 *vrail* per compute node uses 1 HCA whereas MPC with 4 *vrails* (1 *vrail* per level-2 NUMA node) and MPC with 16 *vrails* (1 *vrail* per level-1 NUMA node) opens 4 HCAs. Intel MPI OFA4 and OFA1 respectively refers to as Intel MPI using the OpenFrabrics fabric with 4 HCAs and 1 HCA (`I_MPI_FABRICS=shm:ofa` and `I_MPI_OFA_NUM_ADAPTERS=4 and 1`). DAPL/UD and DAPL/RC open 1 HCA and respectively refers to as Intel MPI using the DAPL fabrics with Unreliable Datagram (UD enabled with `I_MPI_DAPL_UD=enable`) and with Reliable Connection (RC)

one compute node (128 cores) and runs two times slower than MPC. These results are unexpected since it supports the same one-copy intra-node communication than MPC and thus should perform similarly to MPC with 1 compute node. Second and

as far as we know, Adaptive MPI does not support multi-rail configurations and each MPI task communicates through the same HCA. Concerning Intel MPI, the DAPL fabric does not support multi-rail configurations. The only way to use several HCAs with Intel MPI is to enable the OFA interface and to set the environment variable `I_MPI_OFA_NUM_ADAPTERS` according to the number of HCAs in the node. Figure 3.18(b) shows the execution times with various configurations of Intel MPI. As we can see, the OFA interface using 4 HCAs surprisingly performs poorly compared to the DAPL fabric. We suppose that the OFA fabric is not as much optimized as the one based on DAPL and lacks in providing competitive results using multiple HCAs. This assumption is confirmed since the OFA interface using 1 HCA performs slower by a factor of 2 on 256 cores and 3 on 512 cores compared to the DAPL/RC interface.

Figure 3.18(c) focuses on the physical memory allocated per core (i.e., per MPI task) including the runtime and the application. When comparing the different versions of MPC together, MPC with 1 *vrail* has the lower memory consumption. For MPC with 4 and 16 *vrails*, the physical memory allocated slightly increases compared to the 1-*vrail* version because both configurations respectively allocates 4 and 16 times more Infiniband endpoints. If we correlate results from figure 3.18(c) with figure 3.18(a), we notice that MPC with 16 *vrails* is the best trade-off on this machine because the number of *vrail* is multiplied by a factor of 4 and the speedup achieved is nearly 3.5.

When comparing MPC with other MPI runtimes, we notice that MPC allocates significantly less memory, especially when the number of cores grows. There are two main factors that are responsible for this memory overhead. First, as discussed in section 1.3.4, process-based MPI runtimes like Intel MPI usually allocate a large shared-memory segment for intra-node MPI messages whereas thread-based MPI runtimes like MPC or Adaptive MPI perform direct memory copy between end-user buffers. Results obtained with one 128-core node confirm this assumption as, in this configuration, the inter-node communication layer is not initialized and MPC uses less memory. Second, the `MPI_Alltoall` communication pattern requires a fully-connected graph to be established among MPI tasks and MPC allocates less network endpoints than the other runtimes.

To understand why MPC allocates less network endpoints, table 3.4 reports the number of QPs required by several transport protocols including our design referred to as RC-*vrails*. In RC mode, the AllToAll benchmark running on 512 cores requires the creation of 196,608 QPs per compute node to fully support multirail configurations (i.e., every MPI tasks can communicate using the 4 HCAs). As regards the DAPL fabric, it does however not support multirail configurations and only one HCA is used for a total of 49,152 QPs per compute node.

With 512 tasks, figure 3.18(d) shows that the UD mode from Intel (DAPL/UD) is able to reduce by 23% the physical memory used per core compared to RC (DAPL/RC). As presented in table 3.4, UD only requires a single endpoint to connect all compute nodes and it consequently decreases down to 128 the total number of QPs per compute node and per HCA. As shown in figure 3.18(a), UD however affects the performance and communications are slower by a factor of 2. First and as presented in section 3.1.1, UD natively performs sub-optimally because some capabilities normally handled by the HCA are handled by the CPU in this mode. Second and as RC, the DAPL fabric using UD does not support multirail configurations and one single HCA is used.

The OFA fabric of Intel MPI supports the XRC capability previously presented in 2.2.5. We reproduced the same experiments with this mode but surprisingly, no memory gain nor performance degradation was observed compared to the OFA interface with 1 HCA.

Finally, MPC with 4 *vrails* and 4 HCAs only allocates 12 Infiniband QPs per compute node. Furthermore, these 12 QPs are reliably connected using RC and do not express the regular overhead of UD.

Transport Protocols	**# of QPs**
RC	$(N-1) \times C^2 \times H$
XRC	$(N-1) \times C \times H$
RC 4 *vrails*	$(N-1) \times H$
RC 16 *vrails*	$(N-1) \times 4 \times H$
UD	$C \times H$

Table 3.4 – Comparison in the number of network endpoints per compute nodes for several transport protocols with multirail support. Fully-connected cluster with N nodes, C cores per node and H Infiniband HCAs per node. $H = 1$ if no multirail support.

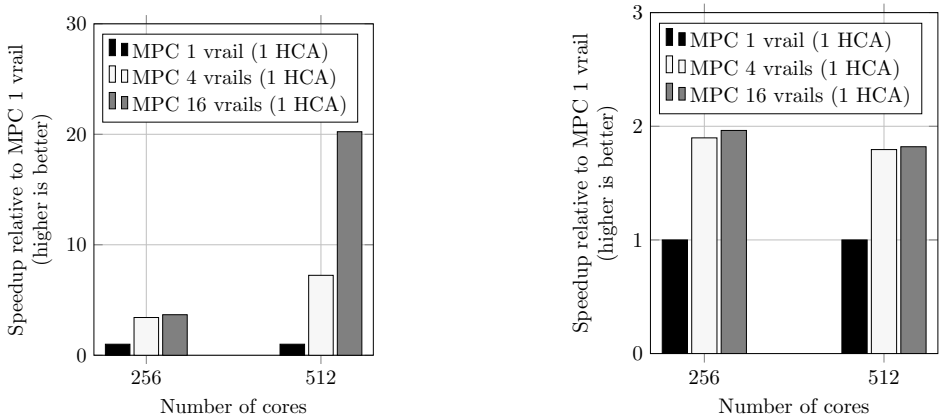
Single HCAs evaluation

We now evaluate the benefits of using multiple *vrails* while only one HCA is opened in the compute node. Figure 3.19 reports the results of the IMB AllToAll micro-benchmark up to 512 cores but, in contrast to the figure 3.18, the same HCA is opened for all *vrails*. We split the results in two figures: in figure 3.19(a), short 1-byte MPI messages are exchanged while large 1-Mbyte messages are used in figure 3.19(b). As we can see in the figure dedicated to short messages, multi-*vrail* configurations exhibit a speedup of 3.5 on 256 cores compared to the mono-*vrail* configuration. With 512 cores, the benefits of multi-*vrail* configurations are even more significant with speedups of 7 and 20 respectively with 4 and 16 *vrails*. This performance gain is actually due to a lower contention on network endpoints. Indeed, 4 *vrails* and 16 *vrails* improve the time spent while posting Send Requests to the Send Queues respectively by a speedup of 5 and 36 compared to 1 *vrail*.

In figure 3.19(b), both versions of MPC with 4 and 16 *vrails* perform similarly. These results are expected because large messages are involved and saturate the network bandwidth. Furthermore, the multi-*vrail* configuration outperforms the regular mono-*vrail* version with a speedup of nearly 2. With 1 *vrail*, tasks waiting for MPI messages aggressively poll the unique *vrail* and potentially access the BCS to read data that reside in a distant NUMA node. As investigated in section 3.3.4.1, repeated accesses to the BCS can lead to the degradation of network communications that also traverse the same bus. As a result, we suspect the MPI tasks to generate a large traffic on the BCS while polling, leading to a penalty of network communications. However, since we do not have sufficient permissions on the machine, we cannot confirm this assumption with an access to hardware counters to measure NUMA traffic.

3.3.4.3 Weak-scaling Evaluation on Athena

Athena is a grid-based code for astrophysical magnetohydrodynamics (MHD) [Sto+08] written in C and parallelized using the MPI interface. We



(a) Short MPI messages (1 byte) **(b)** Large MPI messages (1 MB)

Figure 3.19 – IMB AllToAll micro-benchmark evaluation up to 512 cores (8 nodes) with 1 HCA. Comparison of execution times on MPC with 1 *vrail* per compute node, 1 *vrail* per level-2 NUMA node (4 *vrails*) and 1 *vrail* per level-1 NUMA node (16 *vrails*). Benchmark conducted on the AllToAll micro-benchmark for short MPI messages (a) and large MPI messages (b)

decided to use this application to evaluate the performance and the memory consumption of our multi-threaded communication layer for several reasons. First, Athena fits the requirement of the majority of MPI runtimes as it only requires MPI 1.3. Second, the scalability of the application has been proven up to 25,000 cores[4].

We now evaluate the scalability of MPI runtimes in terms of execution time and memory consumption on Athena running the 3D Rayleigh-Taylor instability problem on the *Large Cluster*, from 256 to 6,144 cores. In order to fit the conditions of a real run, we saturate the compute nodes with the maximum amount of physical memory. To do so, we determined on 2 compute nodes the maximum grid resolution for which all the MPI runtimes successfully run. In these conditions, Adaptive MPI is our reference point for this weak-scaling experiment as it consumes the largest amount of memory. Moreover, we fix to 154^3 the grid resolution per core because it is the highest resolution achievable with Adaptive MPI on 2 compute nodes.

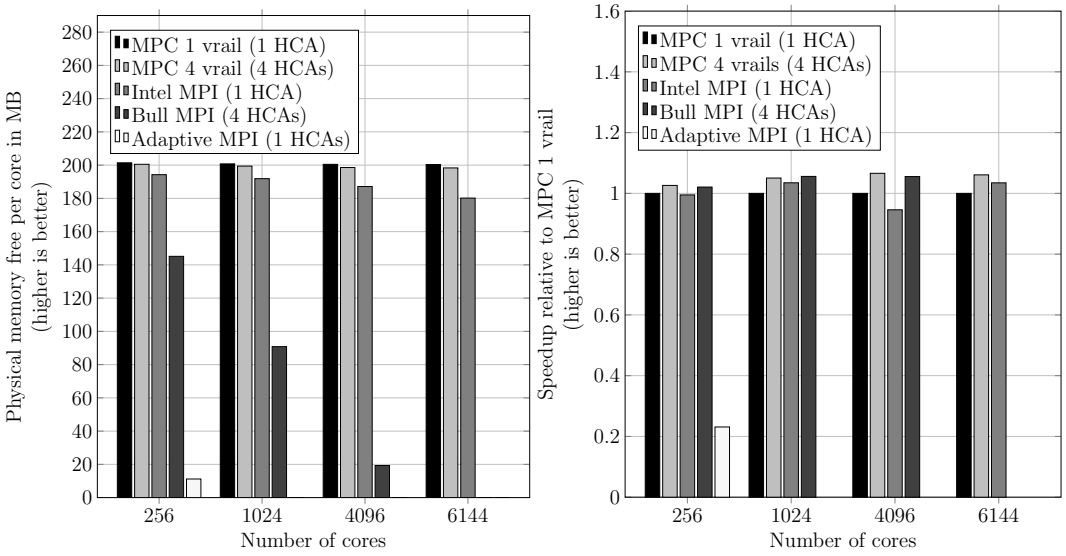
For this case study, we use the `-mem-per-core` option from the SLURM job manager , which bounds the maximum amount of physical memory a task is allowed to use. We set this option to 3 GB, which is the maximal value. One would notice that the maximum amount of per-core memory in this machine is actually 4 GB. In fact, only 3/4 of memory is usable in this cluster because compute nodes should be able to react to the job manager without swapping in case of memory shortage (no swap is actually configured on the system). If a job allocates too much memory, all running MPI tasks are killed and the job manager automatically aborts the job and releases all allocated nodes.

Figure 3.20(a) compares the amount of free memory per core on MPC with 1 *vrail* per compute node, 1 *vrail* per level-2 NUMA node (4 *vrails*/4 HCAs) Intel MPI 4.1.0.024 in DAPL mode (1 HCA), Bullx MPI 1.1.16.5 (4 HCAs) and Adaptive MPI from the Charm++ 6.5.0 package up to 6,144 cores. We slightly modified Adaptive MPI to support the `MPI_STATUS_IGNORE` status flag and manually privatized Athena's global variables to TLS variables using the `__thread` key-

[4]From Athena's website: https://trac.princeton.edu/Athena/

word. This manual privatization allows us to compile all MPI runtimes (especially thread-based runtimes) using the same Intel Compiler 13.0.0 with the same compilation flags. Furthermore, we disabled any output file that Athena could generate. Because Bullx MPI is derived from Open MPI, it inherits the multirail support from Open MPI where all available network links are used to transfer long messages [Woo+06]. Moreover, we set to 4 the maximum number of HCAs to use with the `btl_openib_max_btls` argument. As regards MPC, we only provide the 4-*vrail* version (1 *vrail* per level-2 NUMA node) because we observed minor improvements using 16 *vrails* (1 *vrail* per level-1 NUMA node) with Athena.

On 1,024 cores, Adaptive MPI is the first runtime to be killed by SLURM due to memory shortage. From 256 to 4,096 cores, the free physical memory per core quickly decreases for Bullx MPI and the job is finally killed at 6,144 cores. Intel MPI perfectly passes the test and the runtime exhibits a good scalability up to 6,144 cores and the memory used per core slowly decreases. Concerning MPC with 1 and 4 *vrails*, both versions perfectly scale and the free memory per core is nearly constant whatever the number of cores allocated for the job.



(a) Amount of physical free memory per core **(b)** Execution time

Figure 3.20 – MPC, MVAPICH2, Intel MPI and Open MPI weak-scalability evaluation on Athena using the Rayleigh-Taylor instability problem with a constant resolution of 154^3 per core. Experiences conducted on *Large Cluster* from 32 to 6,144 cores.

Figure 3.20(b) reports the execution time of Athena. Adaptive MPI gets a large overhead due to communications. We believe that Adaptive MPI relies on a unique thread for communicating MPI messages. This model is however not appropriate for large-scale NUMA nodes because one unique thread cannot saturate the network bandwidth and does not leverage the multi-HCA configuration of the cluster. From 256 to 6,144 cores, execution times obtained through the different runtimes, not including Adaptive MPI, are really competitive. More precisely, Bullx MPI and MPC with 16 *vrails* perform slightly better (speedup of 1.06 on 4,096 cores compared to MPC with 1 *vrail*) since both runtimes support multi-rail and open one unique HCA. However, although Bullx MPI efficiently leverages multi-rail configurations, it fails to scale in terms of memory and crashed at 6,144 cores. On the other hand,

MPC with 4 *vrail* provides both best performance and good scalability in memory, even in multi-rail configurations. Finally, Intel MPI shows competitive results while it only uses one HCA.

3.3.5 Future Work: Contention-Based Message Stripping Policy

Thread-based MPI runtimes allow a general overview of the status of all MPI tasks and *vrails* (HCAs) running in the compute node. As an example, when an MPI task sends a message, it can query the status of any *vrails* in the compute node and determine if they are already accessed by another MPI task. If a *vrail* is busy, the message could ideally be sent using another *vrail*.

As a future work, we plan to examine the integration of our contribution with a idle-based (like the one developed for `NewMadeleine` [Aum+07]) or more generally a contention-based routing policy where the HCA that has the lowest workload is used for the communication. Indeed, thread-based MPI runtimes would provide better performance of communications since the routing decision is evaluated at the compute node level and not at the granularity of an MPI task. Additionally, we intend to combine this contention-based routing policy with a stripping algorithm in order to integrate the link contention as a parameter to determine the size of each message fragment.

A message stripping policy with one *vrail* per NUMA node would however face the same scalability issue than the one highlighted in section 3.3.2.2. Indeed, the receiver task would have to poll all *vrails* in order to gather every segment of a previously stripped message. Thus, we plan to design an approach similar to the one implemented by MVAPICH2 and previously depicted in figure 3.8(a). Although this design increases the number of *vrails*, implementing it inside a thread-based MPI runtime would reduce the number of network endpoints by a factor equal to the number of cores per NUMA node.

Finally, some network controllers support new hardware optimization like the Single Root I/O Virtualization specification. Originally developed for hypervizor virtualization like KVM [Kiv+07] or Xen [Bar+03], SR-IOV enables the creation of virtual devices, all bound to the same physical devices. We plan to evaluate the performance of SR-IOV in the context of *vrails* and investigate how it could reduce the overhead due to the multi-threaded access to Infiniband endpoints.

3.3.6 Related Work

Previous research papers have demonstrated the benefits of aggregating multiple network interfaces [LVP04; Col+03]. With Ethernet networks, Penoff *et al.* evaluate the advantage of Concurrent Multipath Transfer (CMT) on MPI [Pen+10]. A few network libraries such as `Elan` on $QsNet^{II}$ and `MX` on Myrinet are capable of providing automatic stripping of RDMA messages and a few collective communications [QA08]. With Infiniband networks, Vishnu *et al.* focused on MPI-2 one sided communication and achieved good performance with `MPI_Put` and `MPI_Get` operations for large messages [Vis+06]. To optimize the message stripping policy, Moreaud *et al.* presented a new scheduling policy that strips MPI messages according to the underlying NUMA effects [MGN10]. In addition, `NewMadeleine` distributes messages over available network links according to the state (idle or busy) of each NIC [Aum+07]. However as far as we know, no previous work allows to share network endpoints among several MPI tasks.

3.4 Automatic Readjustment of Network Buffers

A widespread MPI optimization for two-sided communications is to leverage RDMA operations of the memory semantics for the `eager` protocol and the control messages of the `rendezvous` protocol [KSP09]. This optimization often referred to as `eager` RDMA (or RDMA *Fast-Path*) was first implemented by the MVAPICH2's team [LWP04] and later integrated in other runtimes such as Open MPI.

As presented in section 2.2.1.1 the Send/Receive (SR) semantics requires to negotiate the availability of a receive buffer. By opposition to the SR semantics, the `eager` RDMA directly copies the message to the final address and consequently provides better latency since it induces a lower overhead at the receiver side. In addition, the memory semantics does not involve the Completion Queues described in section 2.2.4 for detecting incoming messages. Indeed, the message detection for the `eager` RDMA protocol is actually performed by polling a specific address in memory.

Despite performance aspects of `eager` RDMA, the usage of this communication protocol is often limited to a few connections. The root cause is a high memory footprint for each connection established since the network buffers cannot be shared between connections. In the following section, we expose a novel runtime technique for dynamically adjusting the memory consumption of RDMA connections according to the MPI application being executed. We prove the efficiency of this approach on a real scientific application where it successfully reduces the communication latencies. We finally explore different future directions to reduce the memory reserved for RDMA connections or, as a last resort to selectively disconnect RDMA channels when the compute node is running out of memory.

3.4.1 `Eager` Network Buffers over RDMA Protocol

`Eager` RDMA channels use two memory regions for communicating unidirectionally with a remote process. It means that for communicating in both directions, the protocol requires the allocation of four RDMA regions.

As depicted in figure 3.21, the `eager` RDMA channels are composed of a set of fixed-size buffers at both sender and receiver sides. All buffers of the same channels are grouped together using linked-lists and each buffer at the sender side has its corresponding receive buffer at the receiver side. To manage the lists, a set of two flags are involved: the *Head* flags corresponds to the next buffer that is free and the *Tail* flag refers to as buffers that have not been yet acknowledged by the receiver. Finally, buffers are consumed in a FIFO way from the *Head* until the *Tail* is reached, meaning no more buffers are free.

To detect the arrival of network messages, the receiver polls a specific memory address and no event is generated to the CQ. Because the sender does not know when reusing buffers, the receiver acknowledges the sender for each completed message using an RDMA write operation. For more details about the `eager` RDMA protocol, the reader may refer to the corresponding paper [LWP04].

Unlike the SR semantics, the `eager` RDMA protocol uses persistent memory regions. It means that a set of buffers is associated with a unique remote process and cannot be shared with other processes. Because the memory allocated for RDMA regions may be large, standard MPI runtimes often implement ondemand `eager` RDMA connections. Initially, MPI tasks are connected using the SR protocol.

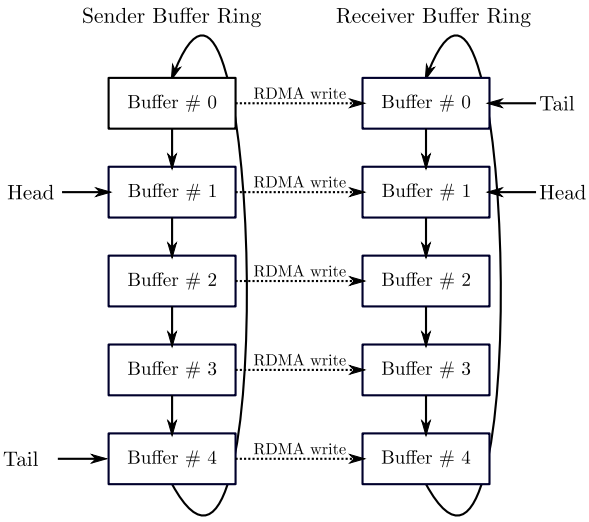


Figure 3.21 – Schematic decomposition of the `eager` RDMA protocol

An `eager` RDMA connection is only established between MPI tasks communicating a large amount of data. When the number of messages exchanged between two MPI tasks excesses a threshold value fixed by the runtime, an `eager` RDMA connection is requested to the remote task.

3.4.2 Contribution: Auto-Reshaping of `Eager` RDMA Buffers

In the following section, we motivate our contribution with an application extracted from the NAS Parallel Benchmarks Suite [Bai+95]. Then, we introduce our contribution that resizes (reshapes) RDMA buffers during the execution of an application. This contribution is twofold. First it saves memory since network buffers are ideally sized to fit the communication volume of each pair of MPI tasks. Second, if an RDMA region has been under-evaluated during a previous allocation, the runtime can enlarge this region (i.e., allocate more memory) to increase the performance.

3.4.2.1 Motivations

		RDMA		
Semantics	Execution Time	# Slots	% Miss	Memory (MB)
RDMA	36.11	1,024	0.00	1,009
RDMA	36.10	512	6.05	505
RDMA	36.61	256	17.97	252
RDMA	36.67	128	31.83	126
RDMA	36.96	64	45.90	63
RDMA	37.10	32	58.81	32
RDMA	37.27	16	71.30	16
RDMA	37.61	8	82.66	8
SR	37.63	0	100.00	0

Table 3.5 – NAS Fourier Transform (FT) Class D on 512 MPI tasks, 32 nodes from the *Thin Cluster*. Size of SR and RDMA slots are set to 16 KB. Results are reported per process.

Table 3.5 reports the NAS Fourier Transform application (FT) Class D on 32

nodes for a total of 512 MPI tasks from the *Thin Cluster*[5]. For a number of RDMA buffers varying from 0 (full SR without RDMA buffers) to 1,024, the table shows the memory allocated per process as well as the percentage of RDMA misses over the total number of messages exchanged (RDMA + SR). The table refers to RDMA miss as a buffer which has been sent using the SR protocol while the RDMA connection between the pair of tasks was being established. A high number of RDMA misses denotes that the amount of memory allocated for RDMA communications is not sufficient for the volume of data exchanged by the application.

At first glance, the table shows that the more RDMA buffers are allocated, the better execution time is: from full SR buffers without RDMA buffers to 1,024 RDMA buffers, the application gets a speedup of 1.04. Furthermore, the best configuration that allocates 1,024 `eager` RDMA buffers exhibits a large memory consumption up to 1 GB per process. Indeed, since FT performs plenty of `MPI_Alltoall` operations, it requires a fully-connected graph where 253,952 $((512-16)\times(512))$ network messages are transmitted during each iteration. As a conclusion to this experiment, it is important to fit the requirements of the application in terms of buffers. Indeed, undersizing RDMA buffers may slow communications since the runtime continuously fallbacks to SR buffers.

In fact, RDMA misses may originate from several levels. Mainstream MPI runtimes usually check the completion of outstanding messages using a polling-based strategy. As a result, long periods of time without calling any MPI function may prevent the polling function to check incoming messages, leading to a large number of buffers waiting for being completed in RDMA channels. Moreover, a high number of unexpected MPI messages may quickly fill RDMA buffers. In such a case, network buffers are received but cannot be released because no matching receive requests have yet been posted. Finally, the communication pattern as well as the MPI neighborhood may change during execution. At some point, the volume of RDMA buffers determined during a previous allocation may thereafter become insufficient. On the contrary, oversizing RDMA buffers may waste useful memory in the compute node. In the worst case, it could even require the node to swap or the job to abort.

As far as we know, current mainstream MPI implementations lack to provide an adaptive `eager` RDMA protocol that dynamically readjusts the amount of memory to fit the application's requirements. At best, the amount of memory allocated to `eager` RDMA depends on the number of MPI tasks in the communication. It means that the more MPI tasks are in the communication, the less memory allocated to `eager` RDMA. As an example with MVAPICH2, from less than 8 compute nodes to more than 128, the maximum amount of memory allocated for `eager` RDMA buffers per connection is decreased from 384 KB (32×12 KB) to 8 KB (4×2 KB) with Curie's Infiniband HCAs.

In the next section, we describe the protocol used for reshaping RDMA buffers.

3.4.2.2 Auto-Reshaping Protocol

Dynamically reshaping `eager` RDMA buffers is a two-sided operation where the sender and the receiver actively participate to the reshaping of RDMA buffers. To synchronize the sender and the receiver, we designed a three-way handshake protocol

[5]The Thin Cluster is detailed in section 2.3.1

close to the one used for on-demand MPI peers connection. The figure 3.22 highlights the protocol used when the sender is the initiator of the reshaping request.

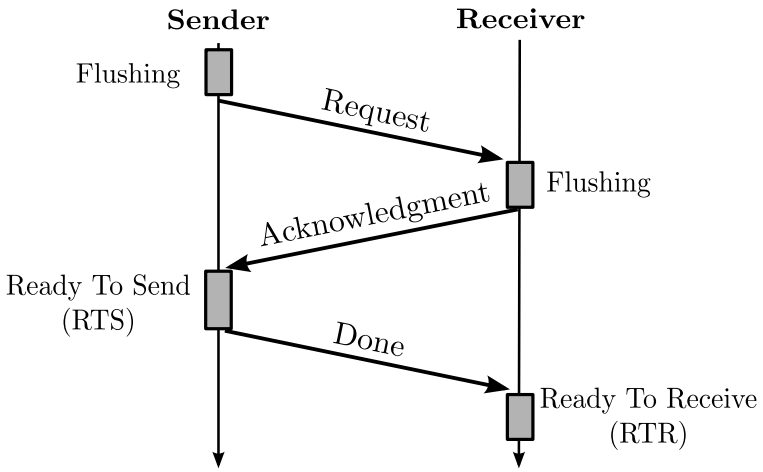


Figure 3.22 – RDMA buffer reshaping workflow. The sender initiates the request.

When a reshaping operation is initiated, a request that defines the configuration of `eager` RDMA buffers to use for the next reshaping is transmitted to the receiver. Once the request is delivered to the destination, the receiver may accept or decline the new configuration. There are several reasons why a receiver may cancel a request: the receiver may for example be out of memory and cannot allocate additional buffers or a disconnection request may already be in progress. Moreover, before re-initializing RDMA buffers, every buffer must be flushed at both send and receiver sides to ensure that outstanding buffers have been properly sent and received.

Once an `eager` RDMA connection is established between two MPI tasks, the previous SR connection remains active. During the reshaping of an `eager` RDMA channel, communications cannot use the RDMA protocol on this channel and automatically fallback to the SR semantics. In other words, while the `eager` RDMA reshaping protocol is progressing, message passing between involved MPI tasks is never interrupted. At worst, communications are penalized with higher latencies induced by the SR semantics. Additionally, no further Infiniband structures – apart from communications buffers – are allocated to communicate using RDMA since SR and RDMA protocols use the same QP, SRQ and CQ structures.

To motivate the need for auto-reshaping `eager` RDMA buffers, we present two typical HPC use cases. First, reshaping `eager` RDMA channels allows to dynamically increase the size of buffers and their number until it covers the requirements of the MPI application. The runtime consequently limits the number of messages sent using the SR semantics and maximizes the usage of RDMA buffers to accelerate communications. Second, we propose to undersize RDMA memory regions for preserving memory. In this context, we present a protocol to dynamically undersize RDMA regions and, as a last resort, selectively release RDMA channels if the compute node is running out of memory.

3.4.2.3 Resizing To Accelerate Communications

As discussed in section 3.4.2.1, `eager` RDMA is a fast communication protocol on the condition that the runtime allocates enough communication buffers to cover the requirement of the application.

During execution, the MPI runtime should collect several informations to estimate the configuration of the network buffers required by the application. The first naïve implementation would be to profile the MPI application to get an estimation of the average size and the number of MPI messages. This solution would however not be efficient as it is too coarse-grain and does not take into account the variations in communication patterns (e.g., bursts of MPI messages during short periods of time). To dynamically adapt RDMA buffers, we rather chose a sampling-based approach where the runtime records the details of the last messages exchanged.

Let us consider $size_{buffer}$ and $number_{buffer}$ the size and the number of RDMA buffers to allocate for a specific RDMA channel. For each neighbor process, a process stores two different data: (1) $messages\_size$ cumulates the size of the last MPI messages sent and (2) $messages\_number$ registers the number of theses messages. Moreover, the runtime records the messages that are transmitted using the `eager` and `buffered` protocols, including the control messages of the `rendezvous` protocol. The requesting process calculates the value of $size_{buffer}$ as it:

$$size_{buffer} = \frac{messages\_size}{messages\_number} \tag{3.1}$$

To approximate the number of buffers $number_{buffer}$, the runtime should determine the maximum amount of data pending in the network since the application startup. This value is private to each RDMA channel and it is referred to as $max\_pending\_data$ in the current paragraph. In addition to the $max\_pending\_data$ variable, the $current\_pending\_data$ variable indicates the current amount of data pending in the network for a specific RDMA channel. Before sending a buffer, $current\_pending\_data$ is incremented with the payload's size of the buffer that is being to be sent. Once the send buffer is free to be reused, an entry that describes the message sent is generated to the corresponding Completion Queue. When this entry is polled, $current\_pending\_data$ is decremented with the value of the payload's size of the buffer sent. At this precise moment, if $current\_pending\_data$ is larger than $max\_pending\_data$, the runtime updates $max\_pending\_data$ with the value of $current\_pending\_data$. The requesting process then calculates the value of $number_{buffer}$ as it:

$$number_{buffer} = \frac{max\_pending\_data}{size_{buffer}} \tag{3.2}$$

When an RDMA connection occurs, both $size_{buffer}$ and $number_{buffer}$ are calculated by the sender and are finally transfered to the receiver through a request for `eager` RDMA connection (see figure 3.22).

3.4.2.4 Resizing to Reduce Memory Consumption

As established before in section 3.4.2.1, the `eager` RDMA protocol may consume a large amount of memory. The memory aspect however becomes critical as soon as the amount of free memory becomes short.

Some applications may express variations in the memory consumption during their execution. This is for example the case of the Adaptive Mesh Refinement (AMR) applications where the grid evolves during the execution time. To illustrate this point, figure 3.23 reports the evolution of the physical memory on two compute nodes executing HERA, an AMR application from CEA [Jou05]. Two observations

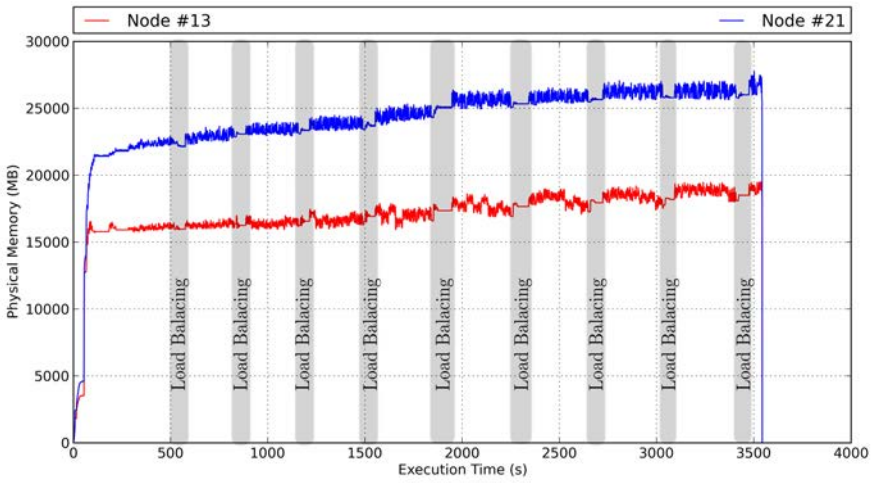


Figure 3.23 – HERA on 64 nodes, 1,024 MPI tasks running on top of MPC. Grid of size 256^3 on 300 timesteps. The figure reports the physical memory allocated on compute nodes 13 and 21.

can be drawn from the figure. First, with the input dataset used for generating the figure, the physical memory reported slightly increases over time. During the first iterations, the runtime could establish RDMA connections because enough memory is available on the node, but over the long term, these RDMA connections may require the job to abort due to the shortage of memory. Second, several phases in the communication can be observed, which correspond to load-balancing and computation phases. During computation phases, it could for example be relevant to disconnect some RDMA channels to limit the memory consumption that slightly increases. At the opposite, the runtime could reconnect RDMA connections to leverage the performance of RDMA `eager` buffers during load-balancing operations that are communication-intensive and consume less memory.

To dynamically reduce the memory allocated for `eager` RDMA, we propose a protocol which, at first, reduces the memory requirements of `eager` RDMA before disconnecting RDMA channels as a last resort. To do so, we suggest three different algorithms to select RDMA connections that shall be reshaped.

- **Emergency**: Disconnect the channels that consume the most memory. This protocol is the most aggressive and should be used when the memory reaches a critical threshold;

- **Normalization**: Select all RDMA regions that consume more than x bytes and reduce their size to x (x can be tuned by the user);

- **Least Recently Used (LRU)**: Disconnect channels according to an LRU algorithm until enough memory has been released.

Finally, RDMA channels are one-sided, meaning that communications are unidirectional inside a single RDMA region. In certain circumstances, it may happen that the receiver is running out of memory while the sender is not. In such a case, the receiver should be able to initiate the auto-reshaping protocol for reducing memory dedicated to RDMA channels.

3.4.3 Multi-Threaded Implementation

We decided to develop our auto-reshaping protocol inside the MPC runtime presented in section 1.2.3.4 and thus for several reasons. First, because it is a thread-based MPI runtime, RDMA connections are at a compute node level and a unique connection may address any task on the node. In addition, MPC supports the Collaborative-Polling (technique later described in chapter 4) and allows a task to poll a message for any other task on the node. This optimization has been extended to RDMA channels.

We developed the `eager` RDMA protocol described in 3.4.1 with several modifications. First, we implemented the `eager` RDMA protocol as a one-way protocol. If two tasks require to communicate using `eager` RDMA, they need to establish two connections, one per direction. Second, we protected structures using a fine-grain locking strategy to enable several threads to access the same RDMA connection.

To communicate control messages for reshaping RDMA buffers, MPC utilizes the signalization network previously described in section 3.2. When a reshaping-request occurs, the runtime reads the `/proc/` pseudo-filesystem for evaluating the physical memory allocated by the process (one unique process is spawned per compute node). To this value is added the memory required for the RDMA channel that is being to be allocated and the result is then compared to a limit set by the user at runtime. If the memory requested overflows the limit, the RDMA connection is rejected and the sender is acknowledged. Furthermore, 10% more buffers are allocated for each connection to limit the aggressiveness of the reshaping.

3.4.4 Experiments

To evaluate the impact of RDMA buffers reshaping on MPI communications, we execute the HERA application on 32 compute nodes of the *Thin Cluster* for a total of 512 tasks. Because HERA uses plenty of packed messages that prevent the runtime to leverage zero-copy communications, we set to 256 KB the switching point from `eager` to `buffered` protocols. Indeed, according to the section 3.1.2.3, `buffered` protocol without cache reuse performs better than `rendezvous` under this threshold. In addition, the last 2,000 messages are profiled to approximate the communication requirements of the application.

Table 3.6 reports the execution times obtained for the full SR mode and three several configurations for the RDMA mode. The RDMA "best config" is the best configuration we have been able to achieve after manually setting up 128 RDMA buffers and tuned their size to 32 KB. The RDMA mode with 1 reshaping initially allocates RDMA buffers with 0 entries and limit to 1 the number of reshaping that can occur during execution. The RDMA mode with an infinite number of reshaping allows RDMA connections to be established while enabling an unlimited number of reshaping. For each configuration, the execution time decomposes into initialization and working times. Furthermore, we instrumented MPC to report the average per process of the memory dedicated to RDMA connections, the number of connections, the number of reshaping and the ratio of RDMA misses. RDMA "best config" shows no RDMA miss because RDMA regions are large enough for supporting the volume of messages that are transmitted. Finally, the miss ratio of the version with an unlimited number of reshaping includes the RDMA misses that are generated before a reshaping protocol is initiated.

Focusing on the initialization time, RDMA with the best configuration performs

	Times (s)		RDMA (average per process)			
Mode	Init	Work	Mem. (MB)	# conn.	# reshaping	miss ratio
SR	432.22	554.97	0	0	0	0
RDMA (best config)	420.39	541.63	130.38	32.19	0	0.00
RDMA (1 reshaping)	428.38	545.31	2.38	32.19	32.19	0.33
RDMA (∞ reshaping)	429.22	538.40	76.70	32.19	42.13	0.06

Table 3.6 – HERA on 32 nodes, 512 MPI tasks. Grid of size 256^3 and 40 timesteps. Comparison between the SR protocol, `eager` RDMA with the best configuration manually achieved, `eager` RDMA limited to 1 reshaping and `eager` RDMA unlimited in the number of reshaping

by far better than versions with one and an infinite number of reshaping. They are different possible explanations. First, RDMA buffers with one reshaping are once allocated at the beginning of the application and remains for the whole application lifespan. As a consequence, the slot configuration does not fit the application requirements. This assumption is confirmed by the high value of the miss ratio. Second, RDMA with an infinite number of reshaping requires some time to converge to a stable value. Indeed, 10 reshaping requests occur, leading to a slight increase in initialization time.

Concerning the working time, the RDMA version with an infinite number of reshaping outperforms other versions. Because the size of the buffers may exceed 32 KB, data are split in less segments and larger buffers are sent than the best RDMA configuration. As a consequence, fewer accesses to network structures are required to send and receive a single message and a 3-second gain can be observed. In addition, because buffers are optimally dimensioned for fitting the size and the number of MPI messages of each connection, physical memory allocated for `eager` RDMA channels is decreased by a factor of 1.7 compared to the best RDMA configuration.

3.4.5 Discussion and Future Work

As an extension to this work, we propose to undersize and disconnect RDMA channels following the protocol described in section 3.4.2.4. However, disconnecting RDMA channels may not be sufficient when facing memory starvation. Indeed, since MPI communications fallback to SR, a heavy network traffic would require SR buffers to be extended and thus, the memory to be increased. More broadly, we plan to investigate disconnection and reshaping of SR buffers, QP disconnection and automatically fallback to the signalization network or over a not-connected network protocol (e.g., UD). A similar approach to QP disconnection has been investigated by for PGAS languages [VKB11].

3.5 Partial Conclusion

In this chapter, we proposed three techniques to reduce the memory consumption of MPI runtimes in order to achieve large-scale executions of parallel applications. First, we have presented a scalable and fully-connected virtual topology for routing messages over connection-based high-speed networks. This contribution implemented inside MPC allows fast MPI peers interconnection over Infiniband and carry control messages during `eager` RDMA reshaping operations. Then we proposed a technique for replicating network structures (or *vrails*) at the NUMA-node level

for improving data locality while accessing network endpoints in a multi-threaded context. We have evaluated two routing policies for selecting a *vrail* and we proposed a protocol for sending `rendezvous` messages across the *vrails* available on the node. The experiments have demonstrated the relevance of the contribution on both, mono- and multi-rails architectures where our design allocates significantly less Infiniband endpoints than the related work for similar performance. Finally, we designed a protocol for dynamically reshaping `eager` RDMA regions. This protocol was implemented for increasing the volume of RDMA buffers and we proposed an extension to release buffer regions when the free memory becomes low.

Chapter 4

Improving MPI Communication Overlap With Collaborative Polling

"Our prime purpose in this life is to help others. And if you can't help them, at least don't hurt them"

Dalai Lama

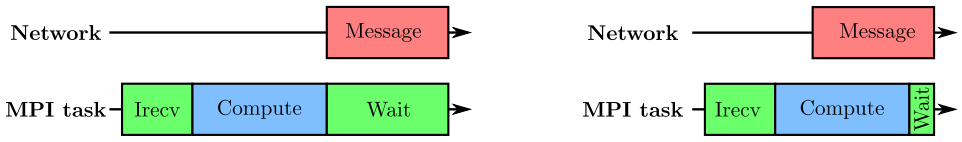
As detailed in section 1.2.3, the regular two-sided communications of MPI require a matching operation to resolve where the message will be copied to. However, section 2.1.3 has previously shown that a wide variety of interconnects such as Infiniband do not fully support the MPI standard in hardware. As a consequence, the host CPU is required to ensure the asynchronous progression of MPI messages. This observation notably poses a problem for the `rendezvous` protocols introduced in section 3.1.2.2. Indeed, this protocol implies the transmission of several control messages that cannot easily be offloaded to the network controller.

This chapter presents a message progression based on Collaborative-Polling, which allows an efficient auto-adaptive overlapping of communication phases by performing computing. This contribution is new as it increases the application overlap potential without introducing the overheads of a threaded message progression. In addition, the proposition improves the independent progression of the control messages involved in `rendezvous` protocols.

4.1 Introduction

The scalability of a parallel application is mainly driven by the amount of time in the communication library. One solution to decrease the communication cost is to hide communication latencies by performing computation during communications. From the application developer's point of view, parallel programming models offer the ability to express this mechanism through non-blocking communication primitives. The MPI standard defines non-blocking *send* and *receive* primitives (i.e., `MPI_Isend` and `MPI_Irecv`) that allow the application to overlap communication with computation. As an example to illustrate those communication patterns, figure 4.1(a) exposes one MPI task performing a non-blocking communication without overlapping capabilities. In such a situation, the message is actually received from the network during the `MPI_Wait` call. As regards the figure 4.1(b), the same example with overlapping shows a significant improvement reducing the overall time consumed.

Achieving overlap usually requires a lot of code restructuring and transformations. Users are often disappointed after spending a lot of time to enforce over-



(a) No Overlapping **(b)** Overlapping

Figure 4.1 – Influence of Communication/Computation Overlapping in MPI

lap because the runtime does not provide an efficient support for asynchronous progress [IB99; BRU05]. Since MPI is a standard, it does not define how asynchronous communications should be implemented inside the runtime. In fact, most of the current MPI libraries do not support true asynchronous progression and performs message progression within MPI calls (i.e., inside `MPI_Wait` or `MPI_Test` functions). The main difficulty with these implementations occurs when an MPI task performs a time consuming function with no call to MPI routines for progressing messages (i.e., calls to BLAS kernels).

In this chapter, we propose a Collaborative-Polling approach for improving the communication overlap without disturbing compute phases. This runtime optimization has been implemented inside the MPC runtime previously presented in section 1.2.3.4. Collaborative polling allows message progression when a task is blocked waiting for a message, enabling overlapping with any other task within the same compute node. This method expresses a significant message-waiting reduction on scientific codes. For this contribution, we focus on the MPI standard and Infiniband networks but the Collaborative-Polling could be adapted to any network interconnect and could be extended to other distributed-memory programming models.

4.2 Related Work

4.2.1 Message Progression Strategies

Previous work has shown significant speedups using overlap of communication on large-scale scientific applications [Bel+06; Sub+11]. For common MPI runtimes, message progression is accomplished when the main thread calls a function from the MPI library. To achieve overlap at user level, MPI applications may be instrumented with repeated calls to the `MPI_Test` function to test all outstanding requests for completion. This solution is not convenient for the developer and irrelevant for not MPI-aware functions (e.g., optimized algebra libraies). For implementations supporting the `MPI_THREAD_MULTIPLE` level of thread safety, (1) Thakur *et al.* [TG07] present an alternative overlapping technique where an additional user-thread is created and blocked inside a `MPI_Recv` function. (2) Hager *et al.* [HJR09] investigate a hybrid MPI/OpenMP implementation with explicit overlap optimizations. (3) Nguyen *et al.* [Ngu+12] propose Bamboo, a source-to-source translator for enabling automatic overlap of communications in MPI programs. However, these three techniques rely on source-code modifications and some involve multiple programming models.

As discussed in section 2.1.2.3, some recent Host Channel Adapters (HCAs) provide hardware support for total or partial independent progress. To enable software overlapping without user source code modifications, FG-MPI [KW12] extends the MPICH2 runtime and allows over-subscribed and non-preemptive MPI threads to

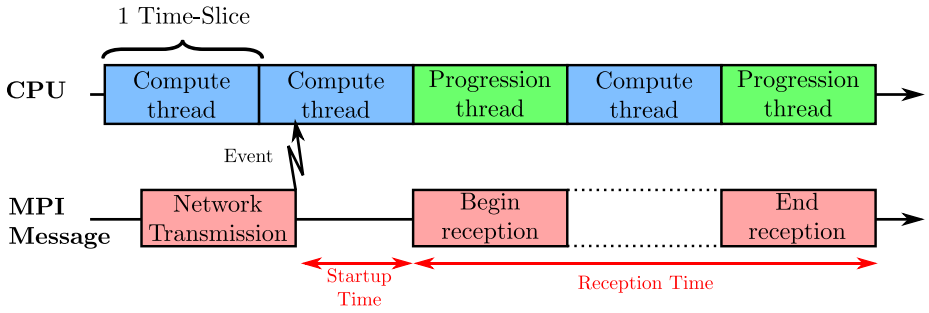


Figure 4.2 – Overheads in a threaded message progression

share the same MPICH2 process. The proposed solution however limits the message progression strategy to a physical core whereas Collaborative-Polling enables it at the compute node level. MPI libraries also investigate a threaded message progression. Additional threads (also known as progression threads) are created to retrieve and complete outstanding messages even if large computation loops prevent the main thread to call the runtime library. For accessing the network hardware, progression threads may be set to use the two strategies previously presented in section 2.2.4: the *polling* and the *interrupted-driven* strategies.

In a full MPI context, the polling approach increases performance on a spare-core thread subscription where the progression thread is bound on a dedicated core. This strategy is for example adopted by IBM in the Bluegene systems (see section 2.1.3.1). Because the spare-core mode wastes computational resources, it is however used in a few limited cases. As an example, the users may under-populate CPU cores when the application faces a memory scalability issue. In fact, MPI is often used in a fully subscribed mode where the same core is shared between the progression thread and the user thread. The decision when and how often the polling function should be called is however non-trivial. Too many calls may cause an overhead and not enough calls may waste the overlap potential.

The interrupted-driven message detection is different from the polling approach since it allows the sender or the receiver to have an immediate notification of completed messages [AA04]. If no work has to be done, the progression thread enters into the wait queue and goes to sleep. When a specific event is generated from the network card (i.e., an incoming message), an interruption is emitted and the progression thread goes back to the run queue. Because generating an interruption for each message may be costly, MPI runtimes often implement a selective interrupt-based solution [Sur+06a; Kum+08]. Only messages that are critical for overlapping performance may generate an interruption.

For the fairness of the CPU resource sharing, each process has a maximum time to run on a CPU: the time-slice. For example, with a Linux kernel it varies from 1 to 10 milliseconds. Once the time-slice is elapsed, the scheduler interrupts the current running thread, places it at the end of the run queue for its static priority and schedules a new runnable thread.

When an interruption occurs, the progression thread has to be immediately scheduled, raising two main concerns. First, it is unclear how much time is required to switch from the active thread to the progression thread: the scheduler may wait for the running thread to finish its time-slice and it is uncertain that the progression thread is the next to be scheduled. Second, one time-slice may be insufficient to poll, match and, if needed, recopy the network message to the end-user buffer. These

overheads are respectively referred to as "Startup Time" and "Reception Time" on Figure 4.2. One solution to increase the reactivity would be to use real-time threads. However, this could increase the context switching overheads since the progression thread is scheduled every time an interrupt occurs [HL08].

The approach most closely related to our proposition is described in the I/O Manager PIOMan [TD09] where the preemptive scheduler is able to run tasks in order to make the communication library progress, leading to an efficient overlap messages in a multi-threaded context. We applied the idea behind PIOMan to MPI runtimes and we propose an optimization where an MPI task may progress messages from another task.

4.2.2 Thread-Based MPI

As introduced in section 1.2.3, thread-based MPI runtimes allow the MPI tasks to share the same memory address space within the same UNIX process.

Because of the implicit shared-memory context among tasks, thread-based runtimes are well suited for implementing global policies, such as message progression, within a compute node. We implemented our contribution in the MPC framework presented in section 1.2.3.4. According to our needs, MPC brings the three following features:

- A thread-based MPI runtime;
- A customizable two-level thread scheduler. It helps to tune the message progression strategies;
- A support for a high-speed and scalable network. It provides an access to Infiniband networks using the OF `verbs` library with an OS-bypass technology;
- An automatic privatization of user's global variables to thread-private variables using a patched version of GCC [CPJ11].

4.3 Our Contribution: Collaborative Polling

During the execution of a parallel MPI application, the time spent while waiting for messages or collective communications is wasted. This idle time is often responsible for the poor scalability of the application on a large number of cores. Even on a well-balanced application at user level, some imbalance between tasks may appear from several factors such as:

- Distance between communicating MPI peers: inter/intra-node communications, the number of network hops.
- Number of neighbors.
- Micro-imbalance of communication (network links contention, topology).
- Micro-imbalance of computation (non-deterministic events such as preemption) [Sub+11].

The main idea of the Collaborative-Polling is to take advantage of idle cycles due to imbalance for progressing messages at the compute node level. During its unused waiting cycles, an MPI task is able to collaborate on the message progression of

any other MPI task located in the same compute node. Figure 4.3 compares the processing of messages arriving from a Network Interface Controller (NIC) with a regular message progression and with our Collaborative-Polling method.

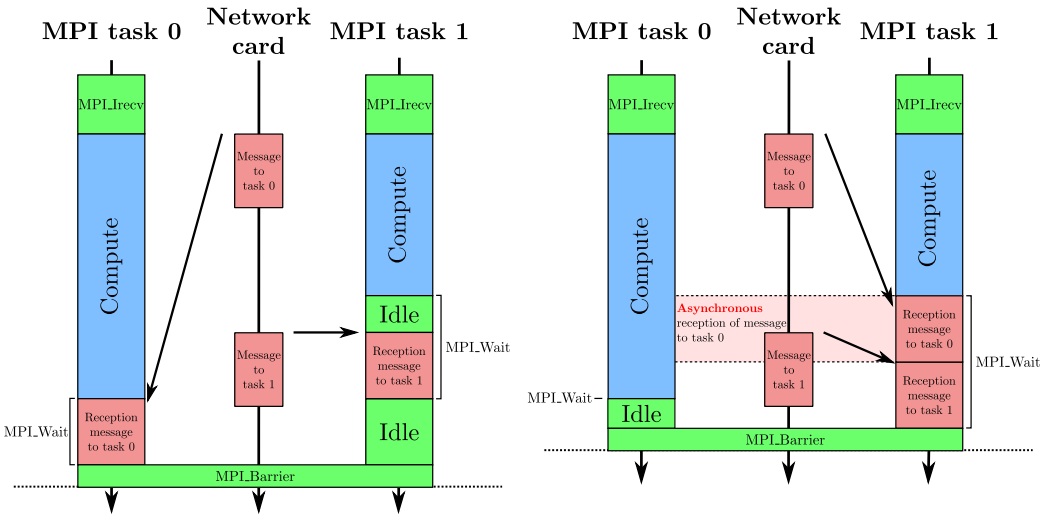


Figure 4.3 – MPI runtime without Collaborative-Polling (left) and MPI with Collaborative-Polling (right)

Figure 4.3 depicts an MPI application performing the following algorithm: each MPI task executes an MPI-unaware function (Compute) with an unbalanced workload between tasks before waiting for a message and calling a synchronization barrier. On the left part, a regular message progression is presented. On the right part, the Collaborative-Polling method is used. Collaborative-Polling allows task 1 to benefit from the unused cycles while waiting for its message: it can poll, receive and match messages for task 0 which is blocked into a non-interruptible computation loop. Once the computation loop is done on task 0, the expected message has already been retrieved by task 1 and the `MPI_Wait` primitive immediately returns.

As described in section 4.2.1 most message progression methods require to suspend the computing phase (with an interruption, an explicit call to MPI or a context switch to the progression thread) to perform progression. Collaborative-Polling does not require these interruptions as it only uses idle time to perform progression. Thus, the impact of Collaborative-Polling on compute time is reduced compared to other methods. Collaborative-Polling also provides an auto-adaptive polling frequency. Indeed, the frequency of calls to the polling function is correlated with the amount of tasks waiting for a communication. For example, when the number of tasks waiting on a barrier increases, the frequency of calls to the message progression method increases as well.

4.4 Implementation

4.4.1 Discussion on Message Sequence Numbers

We designed and implemented our Collaborative-Polling approach into MPC. Since the Infiniband implementation of MPC uses the Reliable Connection (RC) service, the message order is guaranteed and messages are reliably delivered to the receiver. Three message transfer protocols are available: `eager`, `buffered` (see section 3.1.2.3) and `rendezvous` based on RDMA write (see section 3.1.2.2). To guarantee the order

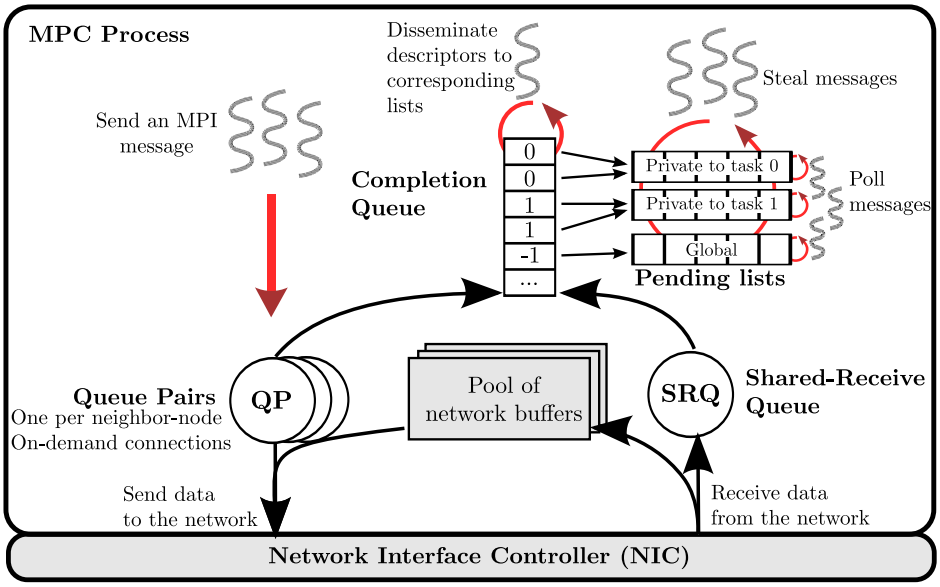


Figure 4.4 – Collaborative-Polling Implementation inside MPC Infiniband Module

across these three protocols, MPC relies on a reordering interface in charge of sorting incoming messages. MPI runtimes usually rely on a Packet Sequence Number (PSN) variable for each pair of MPI tasks. Every message sent carries the current PSN for the corresponding pair of MPI tasks and also increment it. Each receiver maintains an Expected Sequence Number (ESN). When an out-of-order message arrives, it is put into a dedicated queue and its processing is deferred until the missing messages have been handled. Moreover, the reordering interface of MPC guarantees that a stolen message uses the PSN/ESN couple of the stolen MPI task.

4.4.2 Polling Concerns

Recent interconnects such as Infiniband usually exploit Event Queues. When a message is completed by the NIC, a new completion descriptor is posted to the corresponding completion queue (CQ). Then, the CQ is polled to read incoming descriptors and process messages. MPC implements two CQs per *vrail* (see): one for send, another for receive. Both of them are shared among tasks meaning that all notifications are received and multiplexed into the same CQ.

As depicted in figure 4.4, each MPI task implements one private pending list for point-to-point messages. An additional global pending list is dedicated to collective operations and may be concurrently accessed by several tasks. To ensure message progression, the MPC scheduler calls the polling function every time a context switch occurs. The polling function is divided into three successive operations. *First* the task tries to access the CQ and returns if another task is already polling the same CQ. we limit to one the number of tasks authorized to simultaneously poll the NIC because we observed a performance-loss with a concurrent access to the same CQ. Then, each Completion Queue Entry found in the CQ is disseminated and enqueued to the corresponding pending list. At this time, the message is not processed. *Secondly*, the global and the private pending lists are both polled. If some messages reside in the lists, they are processed until an expected MPI message is found. *Thirdly*, with Collaborative-Polling, if a task does not find any message to match, it tries to steal a CQE from a task located in the same NUMA node before lastly

trying another NUMA node.

4.4.2.1 Progression of the `rendezvous` Protocol

As described in section 3.1.2.2, the performance of the `rendezvous` protocol is driven by the capability of the MPI runtime and the network controller to progress synchronization messages. MPC implements such a `rendezvous` protocol based on RDMA write operations. In addition and to reduce the impact of memory registration [Tez+98], this protocol also combines a lazy deregistration and a registration cache to re-use existing registered addresses. Finally, no intermediate copy is allocated, meaning that the receiver waits the receive buffer to be posted before sending the `ACK` message and proceeding to an RDMA write operation.

Figure 4.5, left part, depicts the reception of a `rendezvous` message without Collaborative-Polling. While computing, the receiver cannot handle the `REQ` message. As a result, the matching and the reply only occur inside the `wait` function. With Collaborative-Polling (right part), an idle MPI task may steal the `REQ` message, register the memory for the RDMA operation and finally reply the `ACK` to the sender. As a result, the message transfer can even begin whereas the receiver is still computing.

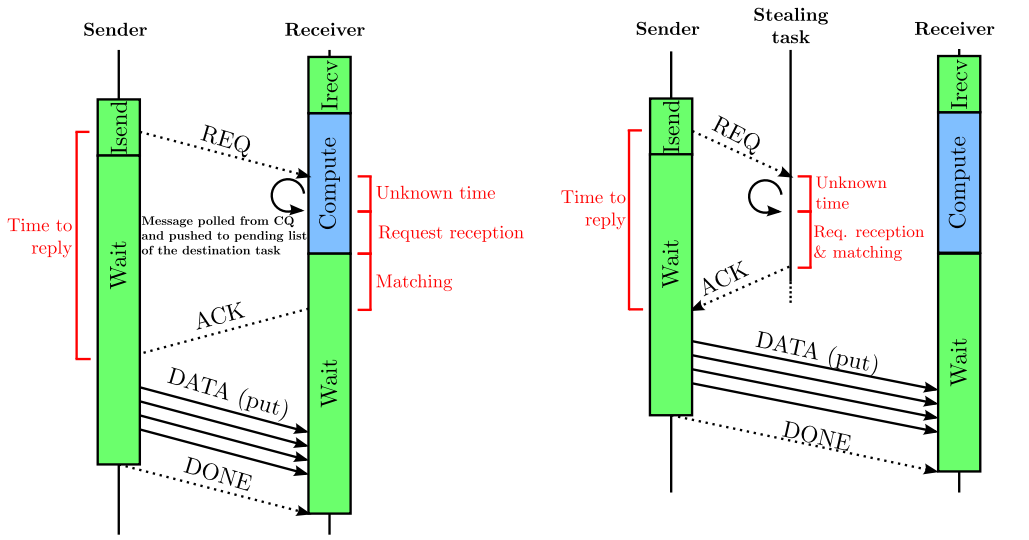


Figure 4.5 – The `rendezvous` protocol with Collaborative-Polling (left) and without (right). With Collaborative-Polling, an idle MPI task may steal a rendezvous control message, match and send the `ACK` to the sender.

4.4.3 Extension to Process-Based MPI

Collaborative-Polling requires the underlying MPI runtime to share some internal structures among tasks located in the same compute node. Within a regular process-based MPI runtime, Collaborative-Polling could be implemented by mapping the same shared-memory segment in each process. The first cumbersome job here is to extract the polling-related structures from the existing runtime and place them in the shared memory.

The second difficulty is to bypass the OS security which prevents several processes to share the same network endpoint. For Infiniband, the Protection Domain (PD) provides an increased level of protection against inadvertent and unauthorized

accesses: a process cannot affect a QP in a different Protection Domain. As far as we know, two processes cannot share the same PD and the compliance C10-7 from the Infiniband Architecture Specification[Inf] requires that each QP in an HCA shall be associated with a unique Protection Domain. To address this issue, we propose an implementation guideline where the runtime spawns and pins for each process as many POSIX threads as physical cores on the compute node. When an MPI task is idle, it can wake and schedule a thread from another process running the same core than it. The newly scheduled thread then may call the progression function and handle incoming messages. Since this approach however requires $O(p)$ threads to be scheduled on each core (p is the number of processors on the compute node), it should be evaluated to quantify the overhead due to context-switching.

An alternative approach would be to use the Linux XPMEM Kernel module that enables a process to expose its virtual address space to other MPI processes [BP11]. This solution was for example adopted by Open MPI in the *vader* Byte Transfer Layer (BTL). However, since installing an external kernel module on an HPC center is discouraged for security reasons and because the project seems no more maintained, we did not focus on this solution.

4.4.4 Extension to Other High-Speed Interconnects

For the following contribution, we designed the Collaborative-Polling for Infiniband networks. However, this approach would be ported to any interconnect, in condition that the HCA does not support a fully independent message progression. In the case of MPI over Infiniband, computation parts such as message matching cannot be offloaded to the HCA and require the involvement of the host CPU to complete the reception. In addition, Collaborative-Polling does not require the underlying network to support communication offload but should be more efficient on such networks.

4.5 Experiments

This section presents the impact of Collaborative-Polling on three MPI applications: EulerMHD [Wol+12], the NAS Parallel Benchmark suite [Bai+95], and Gadget-2 [Spr05] from the PRACE benchmarks. These codes run on the Curie's *Medium Cluster*[1] and we compare our approach (MPC CP) against the regular version of MPC (MPC), MVAPICH2 1.7 (MV2), Open MPI 1.6.1 (OMPI) and Intel MPI 4.0.3.088 (IMPI) which is based on the MPICH runtime. Both, the application and the runtimes have been compiled using GNU GCC 4.4.0 and same compilation flags, except for the MPI runtime from Intel. The results are an average of three runs and the same nodes have been used for comparing the different runtimes.

4.5.1 NAS Parallel Benchmarks

The NAS Parallel Benchmarks (NPBs) are a collection of MPI applications that are distilled from real computational fluid dynamics applications. We omitted the EP benchmark from our study as it is exchanging a negligible number of MPI messages.

Figure 4.6 illustrates the results obtained running the NAS SP, MG, BT, FT, CG and IS with class D on 1,024 cores on several MPI implementations. It decom-

[1]The Medium Cluster is detailed in section 2.3.2

poses the time spent inside the MPI runtime from the computational time. For SP, MG and BT, Collaborative-Polling significantly reduces the time in MPI communications. Apart from Intel MPI on MG where `MPI_Wait` and `MPI_Barrier` functions slow down the execution time, Collaborative-Polling provides performance close to the related work. It respectively gives a speedup of 1.34, 1.25 and 1.69 on the communication time for SP, MG and BT. Figure 4.7 depicts how much time is spent for an MPI task to retrieve its own messages as well as to steal and process messages from another task. For these three benchmarks, we observe a large amount of time stolen by MPI tasks. It causes a significant reduction of the time spent for a task to receive its own messages. We also notice a slight overhead in the message processing. Since we do not have sufficient permissions on the cluster to access the hardware counters, we can only assume that this effect is due to NUMA effects. Indeed, the copy of network buffers to end-user buffers is more costly when it is processed by an MPI task located on a different NUMA node than the node where the end-user buffer is posted. However, this overhead does not negatively affect the total execution time as the message processing occurs during idle time.

On NAS CG, MPC with Collaborative-Polling behaves like the regular version of MPC. Some messages are stolen but the stolen time does not accelerate the execution of the application, probably because the workload is well balanced across the tasks. The same benchmark shows an overhead for Open MPI and MVAPICH2 due to a slowdown in the `MPI_Send` function.

4.5.1.1 Collaborative-Polling and Collective Operations

On the other hand, NAS FT and IS exhibit an overhead using Collaborative-Polling. These benchmarks mostly communicate using collective operations like `MPI_Alltoall`, `MPI_Alltoallv` and `MPI_Allreduce`. The MPC implementation of collective operations uses point-to-point messages and tree-like communications. Collective communication patterns consist of multiple communication stages (a.k.a *rounds*): let assume a communication round k, each MPI task waits a message from the tasks involved in round $k-1$ after sending a message to the tasks of round $k+1$. When a task steals a message from a collective operation, it does not emit the messages corresponding to the next round of the stolen task. In this case, Collaborative-Polling cannot benefit from idle time to recover the time lost while stealing messages.

MPI-3 and non-blocking collective communications could address this issue. Once the collective buffer has been posted, the task should compute and record the communication graph corresponding to the collective for each round. When a steal occurs, the stealing task may access the previously computed communication graph of the stolen task and easily determine which tasks are involved in the next round.

A look at Open MPI and MVAPICH2 shows that, in this configuration, they are both penalized on CG because of a high amount of time spent in `MPI_Send`. Furthermore, Open MPI gets a high overhead in `MPI_Alltoallv` on IS.

4.5.1.2 Block Tridiagonal Solver (NAS-BT)

In this section, we focus on the Block Tridiagonal Solver (BT). This benchmark solves three sets of uncoupled systems of equations. It uses a balanced three-dimensional domain partition in MPI and performs coarse-grained communications.

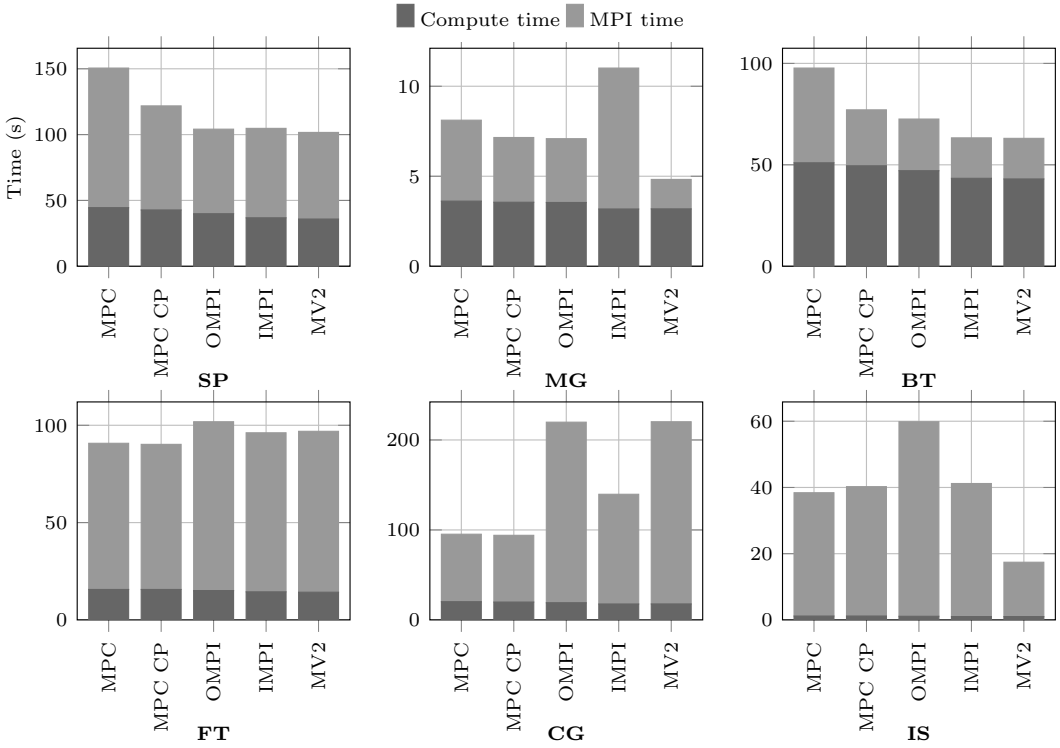


Figure 4.6 – NPB MPI Evaluation. Class D on 1,024 cores

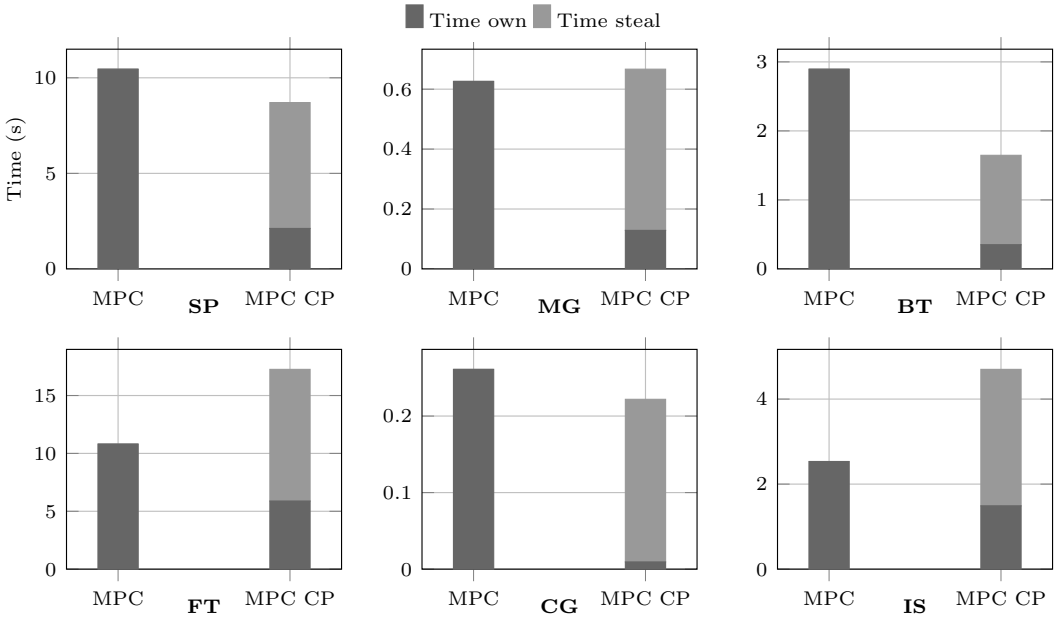


Figure 4.7 – NPB Steal statistics. Class D on 1,024 cores

Figure 4.1 exposes the details of the time spent in the MPI runtime. The gain in MPI time comes from the time spent inside the *wait* functions (`MPI_Wait` and `MPI_Waitall`) because the messages have already been processed by another task when reaching such function. Indeed, Fig. 4.8 shows the amount of messages stolen per task (locally on the same NUMA node or remotely on another NUMA node located on the same computational node). It clearly shows that the number of stolen messages is high, leading to the acceleration of the wait functions.

Function	MPC	MPC CP	Speedup
Execution time	97.69	77.09	1.27
MPI time	46.70	27.58	1.69
Compute time	50.99	49.51	1.03
MPI_Wait	30.58	12.73	2.40
MPI_Waitall	12.59	12.47	1.01
MPI_Isend	1.22	1.33	0.92
MPI_Irecv	1.83	0.67	2.75

Table 4.1 – BT MPI Time Showdown (class D)

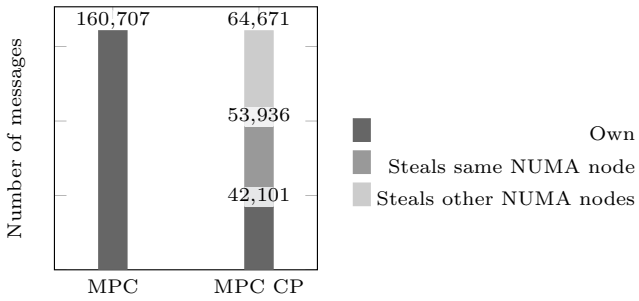


Figure 4.8 – BT steal statistics

4.5.2 EulerMHD

EulerMHD is an MPI application solving both the Euler and the ideal magneto-hydrodynamics (MHD) equations at high order on a two dimensional Cartesian mesh. At each iteration, the ghost cells are manually packed into contiguous buffers and sent to neighbors through non-blocking calls with no-overlap capabilities. Furthermore, each timestep, a set of global reductions on one float number each is performed.

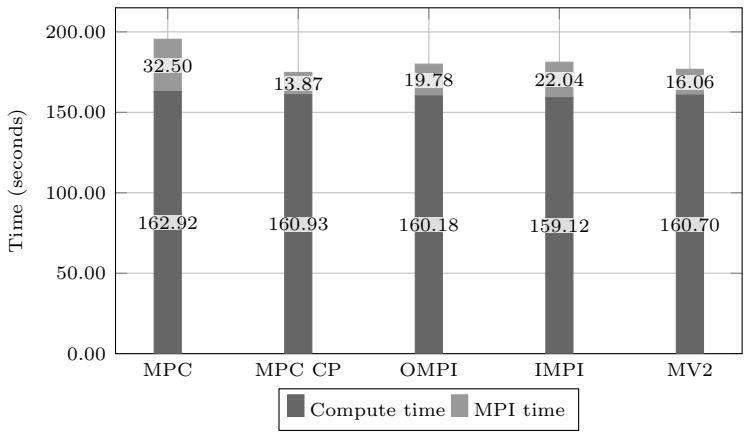


Figure 4.9 – EulerMHD Evaluation

In these experiments, we use a mesh of size $4,096 \times 4,096$ for a total of 1,024 MPI tasks and 193 timesteps. As depicted in Fig. 4.9, the Collaborative-Polling decreases the time spent in MPI functions by a factor of 2. Details of time decomposition is illustrated in Table 4.2. The first time-consuming MPI call, the `MPI_Wait` function, shows a significant speedup by more than 2.5. Surprisingly, the `MPI_Allreduce` function highlights a speedup of 1.58 in this application. It can be easily explained:

Function	MPC	MPC CP	Speedup
Execution time	195.43	174.80	1.12
MPI time	32.50	13.87	2.34
Compute time	162.92	160.93	1.01
MPI_Wait	26.27	10.36	2.53
MPI_Allreduce	4.17	2.63	1.58
MPI_Irecv	1.24	0.18	6.84
MPI_Isend	0.83	0.69	1.19

Table 4.2 – EulerMHD MPI Time Showdown

with Collaborative-Polling, faster MPI tasks already inside `MPI_Allreduce` may help the progression of tasks that did not yet reach this function. Thus, Collaborative-Polling aims to diminish the imbalance across MPI tasks and so the time in global synchronization points such as `MPI_Allreduce`.

The computation loop is also impacted and exhibits a minor improvement. With Collaborative-Polling enabled, the polling function is less aggressive while waiting for messages. This aims to reduce the overall memory traffic.

4.5.2.1 The `rendezvous` Protocol

We run EulerMHD with the same dataset as previous but we disable the Buffered protocol and force MPC to switch to `rendezvous` protocol. In this configuration, 97% of MPI messages are exchanged using `rendezvous`. Figure 4.10 decomposes the time spent inside the MPI runtime from the computational time and it clearly shows that Collaborative-Polling reduces the time to communicate. For a depth investigation, the `rendezvous` interface of MPC has been instrumented with three timers:

1. **Time to reply**: time between the `REQ` and the `ACK` messages on the sender side.
2. **Request reception**: time to handle the message while it as already been polled from the CQ at the receiver side.
3. **Matching**: time to match the message at the receiver side.

Table 4.3 previously presented in section 4.4.2.1 reports the value of these timers on EulerMHD, with and without Collaborative-Polling. On the sender side, the time to reply expresses a speedup of 2,77 using Collaborative-Polling. On the receiver side, because an idle task may handle the `REQ` message from a computing task immediately after it has been polled from the CQ, the request reception time is significantly faster with Collaborative-Polling. Since Collaborative-Polling allows multiple tasks to handle messages for the same remote task, several `rendezvous` messages can be matched in parallel, reducing the time required for matching messages.

4.5.3 Gadget-2

Gadget-2 is an MPI application for cosmological N-body smoothed particle hydrodynamic simulations. At each timestep, the domain is decomposed and the workload is balanced across MPI tasks using a combination of Allgather, Allgatherv and Ssend/Recv functions. During the force computation, each task exchanges

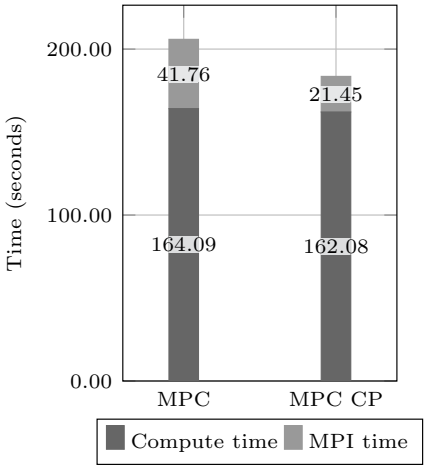


Figure 4.10 – EulerMHD Evaluation

Function	MPC	MPC CP
Time to reply	27.68	9.98
Matching	13.56	5.50
Request reception	6.27	0.08

Table 4.3 – EulerMHD `rendezvous` timers

the number of outgoing particles with a call to `MPI_Allgather` before sending a point-to-point message to each neighbor containing the new positions of the moving particles. From a task to another, the construction of the local tree differs causing an imbalanced workload and a variation in the number of neighbors. The configuration simulates $1e^7$ particles for 16 timesteps on 256 cores.

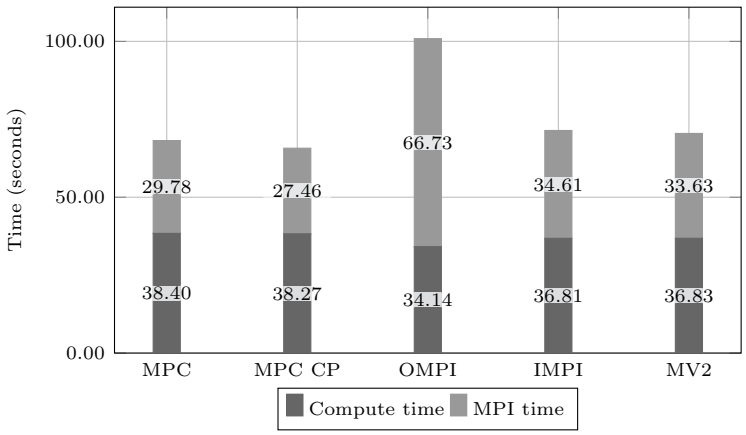


Figure 4.11 – Gadget Evaluation

Function	MPC	MPC CP	Speedup
Execution time	68.18	65.73	1.04
MPI time	29.78	27.46	1.08
Compute time	38.40	38.27	1.00
MPI_Allgatherv	9.51	8.86	1.07
MPI_Allgather	9.34	8.41	1.11
MPI_Sendrecv	3.75	3.47	1.08
MPI_Barrier	3.06	3.04	1.01
MPI_Allreduce	2.03	2.12	0.95
MPI_Recv	0.91	0.69	1.31
MPI_Reduce	0.76	0.52	1.47
MPI_Bcast	0.19	0.15	1.29
MPI_Ssend	0.14	0.14	0.99

Table 4.4 – Gadget MPI Time Showdown

Collaborative-Polling exhibits an improvement in message-waiting time (see Fig. 4.11). Open MPI gets an abnormal slow-down of approximately 10 on the `MPI_Allreduce` function compared to the other runtimes. Table 4.4 details the time acceleration of MPI functions: Collaborative-Polling allows speedup on `MPI_Recv` and `MPI_Sendrecv` functions leading to a 8% improvement for the MPI time compared to regular MPC run.

4.6 Conclusion and Future Work

In this contribution, we proposed a transparent runtime optimization called Collaborative-Polling. This solution does not require to modify the source code of the application nor the programming model. With Collaborative-Polling, the experiments on scientific codes show a significant improvement of the communication time up to a factor of 2. Regular blocking/non-blocking point-to-point communications can benefit from this optimization. Collaborative-Polling may also reduce the imbalance across MPI tasks, diminishing the idle time spent inside global collective operations like barrier, alltoall and allreduce. Additionally to this contribution, Collaborative-Polling was designed for MPI and Infiniband but may be extended to any programming model and any interconnect which does not implement a full independent message progression.

In the worst case of a perfectly well-balanced application, Collaborative-Polling fails to progress message asynchronously. We plan to investigate a mixed-solution with an interrupt-based polling in a future work.

Moreover, although recent network controllers support more and more operations in hardware, the main CPU is still required for completing messages that involve complex operation. It is for example the case with the collective offload capability of the ConnectX-2 interface that only support calculation operations of scalar values [Tec11]. As a consequence, the runtime cannot offload reductions on vector data for now [Kan+12]. Thus, we intend to evaluate the Collaborative-Polling on non-blocking collective communications of MPI 3.0. More specifically, we plan to integrate the NBC library to MPC [HLR07].

Finally, we also plan to focus on hybrid MPI/OpenMP codes where idle OpenMP tasks (i.e., tasks blocked in a barrier) would participate to Collaborative-Polling and progress messages of any MPI task located on the same compute node.

Chapter 5

Evaluation of MPI Runtimes in Hybrid Context

"When someone says: "I want a programming language in which I need only say what I wish done", give him a lollipop."

Alan J. Perlis 1922 – 1990

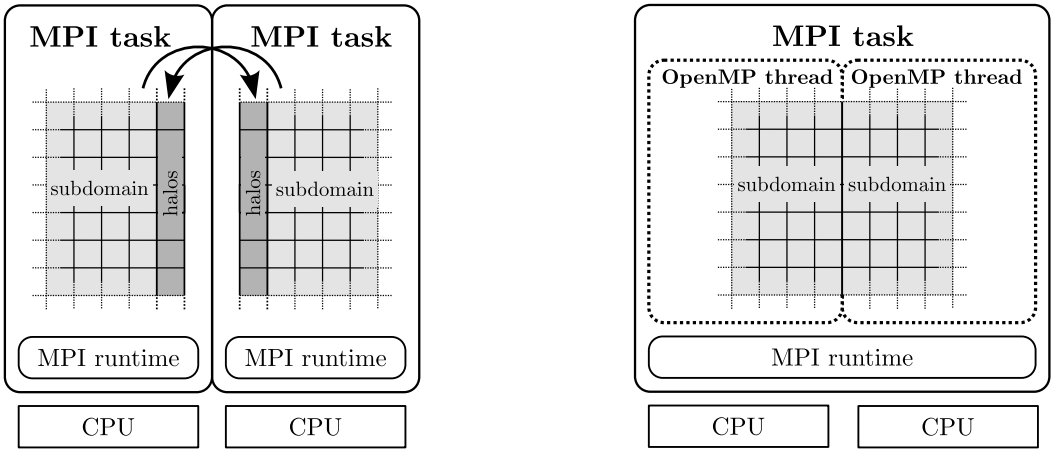
5.1 Introduction to Hybrid Programming

As discussed in section 1.3.4, the memory per core is likely to decrease and maintaining one MPI task per core is already an issue for the scalability of the MPI applications. Moreover, this issue will become more significant over the years.

To reduce the memory of an MPI application, one workaround commonly used on clusters of shared-memory systems is to under-populate CPU cores. As an example, HELIUM is an application that solves the Schrödinger equation to simulate the behavior of helium atoms [SPT98]. Because HELIUM uses a lot of memory, it requires under-populated CPUs for large problem sizes on the BlueGene/P system [Jow+09]. Although this solution improves the scalability of the applications, it clearly under-utilizes the cluster and wastes computational resources.

In a shared-memory context, previous works have already proposed efficient mechanisms for reducing the memory footprint of MPI runtimes by sharing resources [PCJ09]. Inside a compute node, the MPI standard does however not allow the application developer to take full advantage of the shared-memory context between MPI tasks and some user-data are unnecessarily duplicated in the memory. It is for example the case of communication buffers that replicate some cells from the neighborhood in domain decomposition codes whereas a direct memory access would be enough. To limit data replication, the OS and the MPI runtime may propose some mechanisms to share some data between MPI tasks [TCP12]. These solutions are nevertheless barely used because their implementation is either cumbersome or not portable from an MPI runtime to another. Moreover, many HPC applications make use of collective communications like `MPI_Alltoall` where the number of MPI messages and so the number of communication buffers grows with the square of the number of MPI tasks. This issue is much more severe with process-based MPI runtimes where the number of network endpoints increases as much.

Mixing a parallel message passing paradigm for inter-node communications with a shared-memory programming model for intra-node communications intuitively appears to better match these clusters of multiprocessors. The approach typically used and which exhibits the lower memory consumption is to create one unique MPI task per compute node and to spawn as many OpenMP threads as cores. In



(a) Full MPI **(b)** Hybrid MPI+OpenMP

Figure 5.1 – Memory representation of an application parallelized using the domain decomposition method in a shared-memory context

this mode, the single MPI task handles all the network traffic of all OpenMP tasks executing in the compute node.

Although the hybrid approach does not completely solve the scalability that MPI applications may face, it can at least increase the scalability by a factor equal to the number of cores per node. To illustrate this point, figure 5.1 represents the virtual address space of an application parallelized using a domain decomposition method. In the full MPI version (see figure 5.1(a)), the more MPI sub-domains a domain is divided into, the larger amount of memory required for halos (e.g., memory for mirroring neighbor side-domain cells). Additionally to halos, the MPI runtime internally stores in memory structures such as communication buffers, network endpoints or lookup tables. In hybrid mode (see figure 5.1(b)), the memory required by the MPI runtime to work and the number of halo cells are reduced. The combination of the two models has been widely studied and previous works have already successfully improved scalability of full MPI codes by combining them with OpenMP [Bal+09]. As an example, Quantum Espresso (QE) is an integrated suite of computer codes for electronic-structure calculations and materials modelling that consumes a large amount of memory. With the introduction of OpenMP to QE, the hybrid version has made possible the code to scale large datasets up to 65,000 cores on a BlueGene/P machine whereas it was not possible with the full MPI version [Jow+09]. In addition to the memory reduction, hybrid codes propose some other advantages over full MPI codes.

First, shared-memory programming models most often exhibit a convenient interface for automatically balancing workload across the available threads. Since the OpenMP 2.5 specification and earlier versions, the standard defines the `for` construct and the "dynamic" and "guided" scheduling [Ope13]. With the introduction of task parallelism in OpenMP 3.0, the language has become more dynamic. Dynamically sharing work across shared-memory tasks is particularly interesting for codes which express a high imbalance such as Adaptive Mesh Refinement codes [Key+00]. On the contrary, implementing a user-level load-balancing mechanism over MPI usually leads to significant communication overhead, particularly on fine grain parallelism problems.

Second, to communicate in shared memory, OpenMP threads only require con-

ventional memory read and write operations. As discussed in section 1.2.3.4, mainstream process-based runtimes usually involve two memory recopies for intra-node communications: the message is first temporary buffered into a shared memory segment before being copied back to the receive buffer [BMG06]. With the aim of reducing these memory copies, some process-based MPI runtimes rely on kernel modules (KNEM [GM12], LiMIC2 [Jin+07]) for providing one-copy mechanisms. These optimizations are however restricted to large messages as they involve an explicit synchronization between the sender and the receiver and increase the latency. More recently, Friedley *et al.* proposed Hybrid MPI (HMPI) [Fri+13], a runtime that performs zero-copy messaging between shared-memory processes. When the application requests memory, the runtime returns a memory space from a shared memory segment. Thus, each buffer passed to MPI calls is visible to every process at a node level. Although these optimizations improve intra-node communications, the MPI standard imposes at least one memory copy and the OpenMP model is consequently more appropriate for shared memory communications. Regarding inter-node communications, hybrid MPI/OpenMP applications typically exchange the same amount of data over the network but the number of inter-node messages is reduced and their size is increased as well.

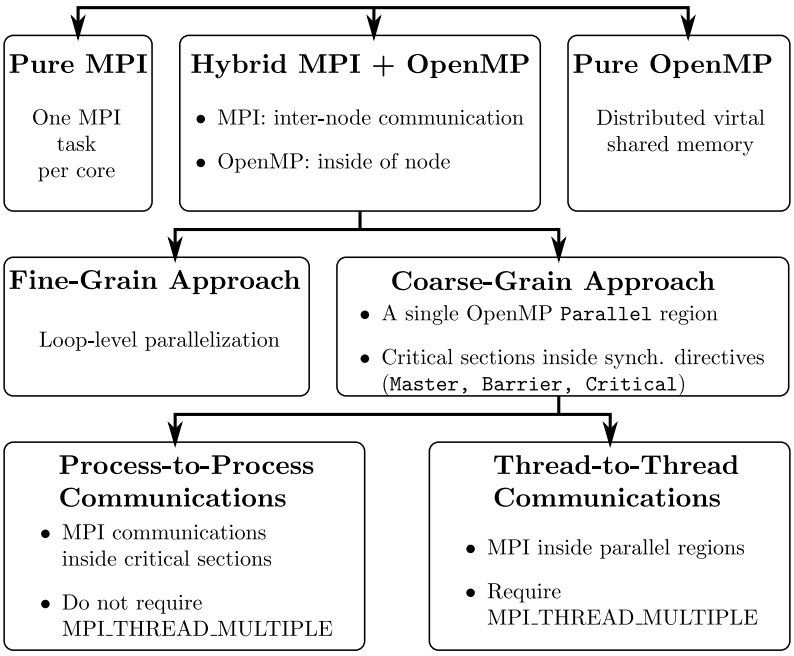


Figure 5.2 – Taxonomy of parallel programming models for hybrid MPI+OpenMP applications [RHJ09]

Developing a hybrid MPI/OpenMP application usually means integrating OpenMP into an existing MPI application and, for this purpose, two different approaches are typically used. As depicted in figure 5.2, the incremental approach consists in parallelizing loops one by one whereas the SPMD-like programming style defines an approach close to the regular MPI programming where the work is statically decomposed and processed according to the thread IDs.

5.1.1 Fine-Grain Parallelization

The incremental approach (or fine grain parallelization) consists in gradually hybridizing loop nests of an existing MPI application. In this context, programmers usually resort to profiling tools to select the loop nests that contribute the most to the global execution time.

Achieving efficient parallelization with OpenMP is actually not just about surrounding *for*-loops with `omp parallel for` constructs. Indeed, OpenMP like any other shared-memory programming model requires attention when accessing data in NUMA architectures and a high number of synchronization points may slow the performance of the application [Ric98]. To improve the parallel efficiency of the hybrid code, additional optimizations should then be performed such as loop permutation, loop exchange or use of temporary variables.

In practice, the incremental approach generally performs sub-optimally [CPP01]. To understand the results, let us decompose the execution time t_{tot} of a hybrid application into:

$$t_{tot} = t_{seq} + t_{comm} + \frac{t_{compute}}{p} \tag{5.1}$$

t_{seq}, t_{comm} and $t_{compute}$ are respectively the sequential time (i.e., synchronizations, not-parallelized code sections, thread management including opening and closing parallel regions), the MPI communication time (that is performed by only one OpenMP thread) and the computation time fully parallelized with OpenMP. Furthermore, p corresponds to the number of OpenMP threads per MPI task. The incremental approach has actually two limitations which may increase the time in t_{seq}. First, a high number of parallel constructs implies a high cost due to thread management and second, there is no guarantee of affinity between threads and processors for consecutive parallel regions. Moreover, because t_{seq} and t_{comm} cannot be accelerated, the Amdahl's law applies and the speedup is limited by $1 + \frac{t_{compute}}{t_{seq}+t_{comm}}$. Thus, if t_{seq} and t_{comm} represent for example 10% of the execution time, the maximum speedup that can be achieved with an infinite number of OpenMP threads and using the incremental approach is 10×.

5.1.2 Coarse-Grain Parallelization

To overcome the weaknesses of the fine-grain model, the coarse-grain method opens an OpenMP `parallel` region at the beginning of the program (just after the spawn of the MPI tasks) and the region remains active until the end of the execution. The programmer must then use synchronization directives such as `single`, `barrier` or `critical` in order to ensure the memory coherency of shared variables and non parallel sections. Although this model solves the performance issue due to the consecutive parallel regions, it still includes the cost to access critical sections due to data-sharing.

To reduce this cost, the SPMD programming style tries to limit the data exchanges between OpenMP threads [KC03; Ber+05]. In practice, each OpenMP thread acts similarly to an MPI task. The OpenMP `for` directive is no longer used for distributing loop iterations and the programmer rather calculates the OpenMP work distribution according to the OpenMP thread ID (obtained via a call to `omp_get_thread_num()`). Moreover, this approach goes against the principle of shared-memory programming models since shared-memory variables are recopied and stored into thread-private memory regions. As a result of the SPMD approach,

the sequential time (referred to as t_{seq} in the formula 5.1) is notably reduced but this solution however exhibits three important disadvantages that must be considered when mixing an application with OpenMP: (1) the programming complexity is significantly increased, (2) the SPMD approach sacrifices the availability of the OpenMP dynamic load-balancing since the work is statically distributed and, (3) more memory is used since the approach requires as much user buffers (e.g., communication buffers) as the full MPI version.

As a conclusion to the coarse-gain hybrid parallelization, one limitation is actually the MPI communication time referred to as t_{comm} in the formula 5.1. Indeed, with process-to-process communications depicted in figure 5.2, MPI calls are serialized into OpenMP `master` directives of equivalent. To reduce the value of t_{comm}, the thread-to-thread communication model proposes a concurrent access of the OpenMP threads to the MPI runtime. Although this model efficiently parallelize MPI communications, it however requires the MPI runtime to be thread-safe.

5.2 Performance Evaluation of MPI Runtimes in Multi-Threaded Context

5.2.1 Related Work

The research community has already spotted the need for an MPI implementation that efficiently supports multi-threading [PS98; TG07]. Since the version 2.0 of the standard, MPI introduces four levels of thread safety, which define how the runtime should behave in a multi-threaded environment. Thus, it is the user's responsibility to declare with the function `MPI_Init_thread` which level to use for the application. These levels are as follows:

1. `MPI_THREAD_SINGLE`: Each process has a single thread of execution.

2. `MPI_THREAD_FUNNELED`: A process can be multi-threaded, but only the thread that initialized MPI can perform MPI calls. A thread can determine if it is the master thread with the `MPI_Is_thread_main` call. In fact, this determines whether it is the same thread that called `MPI_Init`.

3. `MPI_THREAD_SERIALIZED`: A process can be multi-threaded, but only one thread at a time can call MPI.

4. `MPI_THREAD_MULTIPLE`: A process can be multi-threaded, and multiple threads can simultaneously call MPI functions.

Most share memory programming models have already been evaluated in combination with the MPI standard such as SMPSs [Mar+10], Habanero-C [Cha+13], OpenMP [Hag+], StarPU [Aug+12] or the distributed version of CnC on top of MPI [SBK13]. Results have demonstrated that threads enable better progression of asynchronous non-blocking MPI communications and that hybrid programming can outperform the original pure MPI version of the code. The previously cited implementations however rely on one unique thread that handles and progress MPI communications of the whole node. It is certain that this solution will not scale with large compute node and in the long term, one thread will be insufficient to saturate the network bandwidth [RW03].

Hager *et al.* in [RHJ09] present common overheads while developing hybrid MPI/OpenMP applications, most notably performance issues due to a bad thread placement policy over the cluster. Several researches have previously concluded that the benefit a hybrid application may be obtained is not trivial and requires consideration of issues such as the level of shared memory parallelization achievable, the communication cost, the performance balance of machine's main components, the domain decomposition and the memory access patterns [CE00; SB01; DK04; RW03]. Some researchers have demonstrated success programming real-world OpenMP application in a SPMD way where the thread id is using for explicitly managing data [KC03; Ber+05].

Focusing OpenMP, it is often cumbersome to get good performance with this programming interface due to the memory model that is not aware of the non-uniform memory access characteristics of the underlying compute node's architecture. To better match the hierarchical memory structure of modern architectures, `ForestGomp` (extension to the GNU OpenMP implementation [Bro+10a]) and the authors in [Jin+11] present an extension for augmenting locality in OpenMP.

Finally, another aspect to take into consideration is the multi-threaded ability of MPI runtime. Benchmark Suites have been developed for evaluating MPI performance using the `MPI_THREAD_MULTIPLE` level on a variety of platforms/MPI libraries and results have pointed out that MPI implementations behave very differently in a multi-threaded context [TG07; BEA09]. Developing a thread-safe MPI runtime actually requires to consider many aspects for achieving performance and correctness [GT06]. Furthermore, several extensions to the MPI standard have been proposed to provide a better combination of MPI and shared-memory programming models [Din+13; Hoe+10].

5.2.2 Motivations

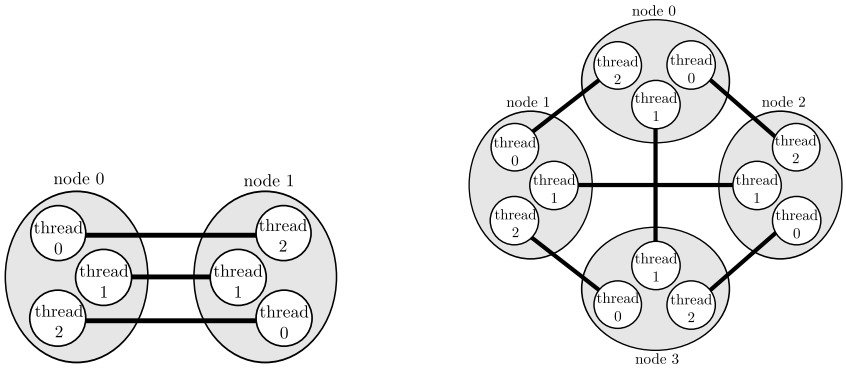
In practice, hybrid applications are often designed using process-to-process communications. The basic justification for this choice is that some MPI runtimes have only to support the `MPI_THREAD_FUNNELED` level since communications are sequentially executed by the master thread. Using this model, hybrid applications may however perform sub-optimally for several reasons. First of all, because only one thread communicates at any moment, it increases the sequential portion of the code. Second, it is uncertain if one thread communicating is sufficient for reaching the maximum network bandwidth [RW03]. This statement is even more true when they are multiple HCAs available on the compute node.

As shown in section 5.1.1, the coarse-grain parallelization approach with thread-to-thread communications seems more suitable for developing hybrid applications since it reduces the sequential portion of the code and parallelize MPI communications. However and as detailed in section 5.2.1, developing a thread-safe MPI runtime actually requires to consider many aspects for achieving performance and correctness. Most MPI runtimes efficiently implement thread levels up to `MPI_THREAD_SERIALIZED` but do not perform as well with the `MPI_THREAD_MULTIPLE` level. For example on Infiniband with `MPI_THREAD_MULTIPLE`, Open MPI disables its `verbs` communication interface and fall-backs to another communication protocol (e.g., TCP/IP). As a result, implementing the thread-to-thread communication model usually increases the sequential portion of the code and consequently provides poor performance [SB01; DK04].

In this chapter, we evaluate the performance of MPI runtimes in a multi-threaded context. We first focus on different hybrid MPI+Threads latency benchmarks on small and large compute nodes up to 96 cores. We then detail the parallelization of a hybrid seismic modeling applications using MPI+OpenMP. Three different hybrid versions of the code are evaluated on 2,048 cores and compared to the original full MPI version. We finally introduce the limitations of hybrid modes and discuss how these limitations could be addressed with the *MPI Endpoints*, i.e., an extension to the MPI standard.

5.3 Micro-Evaluation: MPI_THREAD_MULTIPLE Test Suite

To evaluate how MPI runtimes behave in a multi-threaded environment, we submitted MPC and Intel MPI to multiple different hybrid MPI+Threads benchmarks that require the `MPI_THREAD_MULTIPLE` level of thread safety.



(a) *single-rank* **(b)** *multi-rank*

Figure 5.3 – Hybrid latency benchmarks with 3 threads per compute node

The first benchmark named ***single-rank*** latency test (see figure 5.3(a)) has been originally developed by Thakur *et al.* [TG07] for the `MPI_THREAD_MULTIPLE` test-suite[1]. The benchmark runs on two nodes with one unique MPI task on each node. A set of threads are spawned using the POSIX API and each of them communicates with another thread from the second node. The goal of this benchmark is to highlight the fine-grain locks that protect runtime internals from concurrent accesses to the structures of the same MPI task. We slightly modified the original version of the code from the Thakur's paper in order to provide a fair comparison between the different runtimes. First, and to prevent thread migration, the benchmark manually pins each pthread in a compact mode using the `sched_setaffinity` Linux-specific system call. Second, start addresses of MPI buffers are manually aligned to a system page for achieving best performance on DMA-based networks such as Infiniband. This optimization is sometimes automatically performed by runtimes (e.g., Intel MPI, Open MPI). Indeed, during execution, the runtime captures dynamic memory allocations and aligns the block of memory to the size of a system page. Third, the benchmark now performs a warm-up iteration to bypass the on-demand connection mechanism of MPI runtimes and fill the cache memory.

The second benchmark named ***multi-rank*** latency test (see figure 5.3(b)) is a benchmark from our contribution. It is similar to the *single-rank* benchmark

[1]The test-suite is available for download at http://www.mcs.anl.gov/~thakur/thread-tests

previously described but all POSIX threads running on a same node communicate with different MPI tasks, so different compute nodes. This benchmark eliminates fine-grain locks and only focuses on coarse-grain locks that are taken whatever the remote MPI task to reach.

Our reference point for evaluating the overhead due to multi-threading is the ***full-MPI*** latency benchmark provided by Thakur. This code runs on two nodes and each task on the first node communicate with a task located on the second node.

5.3.1 Thread-Safe MPI Runtimes

For this multi-threading evaluation, we need to select MPI runtimes that combine both (1) a support for the `MPI_THREAD_MULTIPLE` level of thread-safety and, (2) an access to Infiniband networks.

Based on our experiments, Bullx MPI does not provide a level of thread safety higher than `MPI_THREAD_SERIALIZED`. Moreover, we attempted to compile OpenMPI 1.6.1 with a support of multi-threading (compilation flags `-enable-mpi-thread-multiple` and `-enable-opal-multi-thread` associated to `-with-openib`) but the runtime refuses to execute with another MCA (Modular Component Architecture) than TCP. As regards MVAPICH2 1.9, we compiled the library with the highest level of thread-safety (compilation flag `-enable-threads=multiple` combined with `-with-ibverbs`) but the application fails to correctly initialize with the `MPI_THREAD_MULTIPLE` level. Finally, we installed MVAPICH2-X 1.9, a runtime from Ohio State University and optimized for hybrid programming models but the execution fails in the `MPI_Init_thread` function.

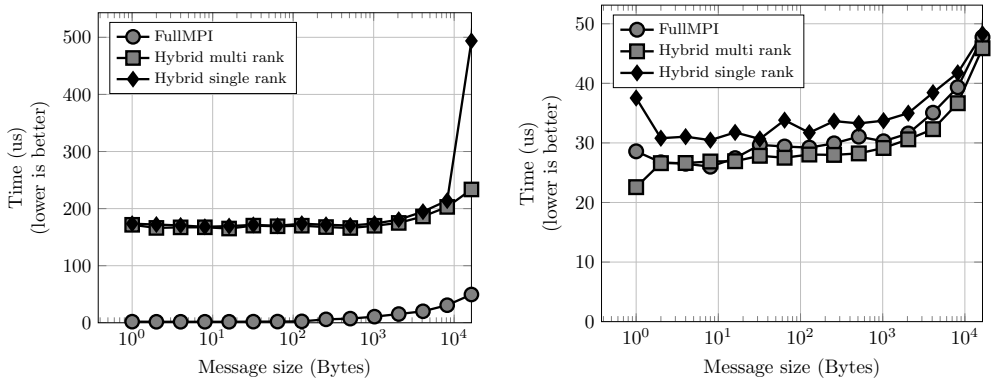
MPC presented in section 1.2.3.4 and Intel MPI are two MPI runtimes that fulfill our requirements for this evaluation. Indeed, they both support the `MPI_THREAD_MULTIPLE` level of thread safety detailed in section 5.2 and provide high-speed communications over Infiniband networks.

5.3.2 Thread Overhead on Small Compute Nodes (16 cores)

For this first set of results, we run the three latency benchmarks (*single-rank*, *multi-rank* and *full-MPI*) previously described in section 5.3 on the *Thin Cluster*[2]. Moreover, the benchmarks are executed using MPC and Intel MPI, two MPI runtimes that support the `MPI_THREAD_MULTIPLE` level of thread safety as discussed in section 5.3.1.

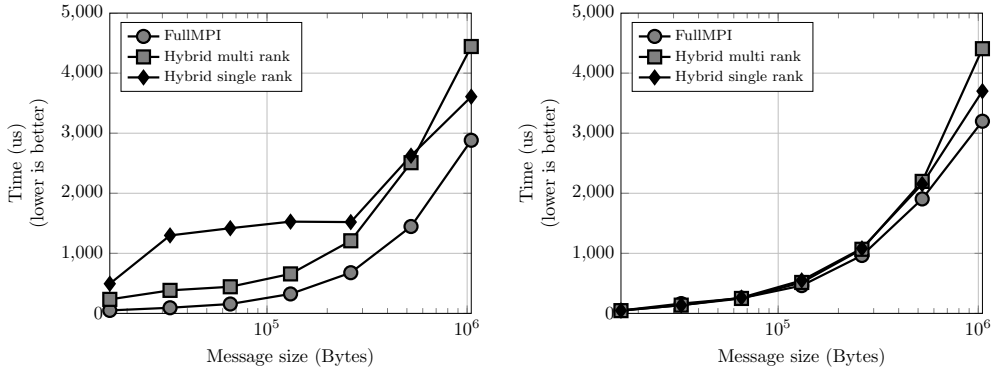
Figures 5.4 and 5.5 respectively report the latency of the three versions for `eager` messages up to 16 KB and `rendezvous` messages up to 1 MB. As we can see in figure 5.4(a), Intel MPI expresses a high and constant overhead of 170 μs for both the *single-rank* and *multi-rank* versions due to multi-threading. In contrast, MPC in figure 5.4(b) is slightly penalized by multi-threading and the *multi-rank* benchmark even performs better than the regular *full-MPI* version. To understand this speedup, let us focus on the implementation of the MPI matching semantics. MPI runtimes (including MPC) regularly implement two main queues for message matching: (1) a *posted receive queue*, which stores pending receives posted by an MPI task and, (2) an *unexpected message queue* for incoming messages, which have not been yet matched. In hybrid mode, those two MPI matching queues are actually shared

[2]The Thin Cluster is detailed in section 2.3.1



(a) Intel MPI **(b)** MPC

Figure 5.4 – Hybrid and *full-MPI* latency benchmarks on 16 cores node. `Eager` messages from 0 KB to 16 KB.



(a) Intel MPI **(b)** MPC

Figure 5.5 – Hybrid and *full-MPI* latency benchmarks on 16 cores node. `Rendezvous` messages from 16 KB to 1 MB.

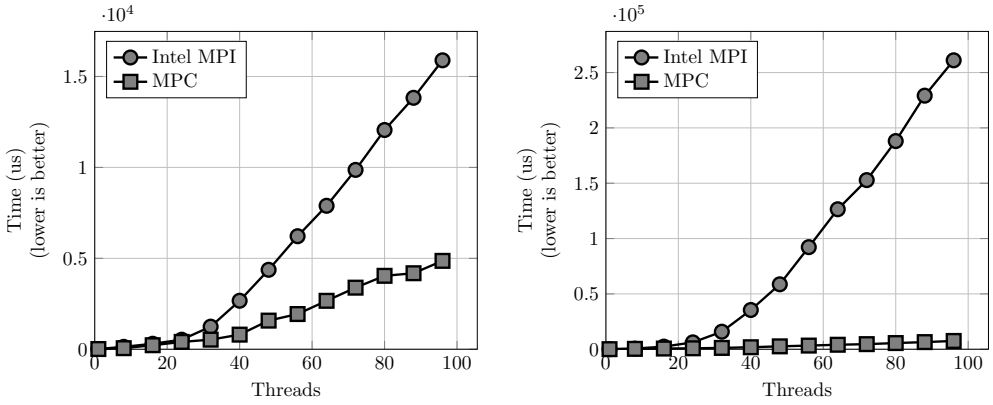
across all POSIX threads spawned by the same MPI task and the runtime does not distinguish two messages from two different threads. As a result, when a thread is waiting for a message, it can progress and match an incoming message with a request previously posted by another thread. This is actually what happens with the *multi-rank* benchmark and one would notice that this mechanism is similar to the one proposed by the Collaborative-Polling in chapter 4. Moreover, the *multi-rank* test is not subjected to overhead due to multi-threading because each thread in a compute node communicates with a different remote MPI task. As regards the *single-rank* benchmark, it slightly suffers from an overhead due to multi-threading. Indeed, compared to the *multi-rank* benchmark, the *single-rank* benchmark generates a higher contention on both MPI matching queues and network endpoints since 16 threads access the structures of the same remote MPI task [GT06].

Figure 5.5 reports the results obtained with Intel MPI and MPC for `rendezvous` messages up to 1 MB. In this case, Intel MPI gets a significant overhead for messages from 16 KB to 256 KB because of multi-threading. This overhead is even amplified with the *single-rank* benchmark where Intel MPI clearly shows a coarse-grain locking strategy for protecting network endpoints from concurrent accesses. From 256 KB, this overhead is attenuated since the contention on network structures decreases. As regards MPC, the three benchmarks provide really close results. From 512 KB,

a minor overhead due to multi-threading appears and MPC finally performs in the same order of magnitude than Intel MPI for 4 MB messages.

5.3.3 Multi-Threading MPI Scalability on Large Compute Nodes (128 cores)

In the following section, we propose to analyze the multi-threading scalability of MPI runtimes on the *single-rank* benchmark with 2 compute nodes from the *Large Cluster*[3]. We have actually limited the number of cores to 96 as the runtime from Intel fails to complete the test and crashes with a higher number of cores.



(a) `Eager` protocol: 1 KB messages

(b) `Rendezvous` protocol: 100 KB messages

Figure 5.6 – Latency *single-rank* test up to 96 cores

Figure 5.6 reports the latencies obtained with Intel MPI and MPC for 1 KB `eager` messages and 100 KB `rendezvous` messages. For both short and large messages, MPC slightly outperforms Intel MPI with 32 or less threads (one level-2 NUMA node). At this point, the latency of Intel MPI rapidly increases with 96 cores, the runtime is finally slower than MPC by a factor 3.3 and 34.8 respectively with `eager` and `rendezvous` protocols. As a conclusion to these experiments, an MPI runtime that efficiently supports multi-threading is primordial for achieving performance with hybrid applications. The results up to 96 cores highlight the limitation of Intel MPI and prove the viability of MPC to support hybrid programming on large NUMA nodes.

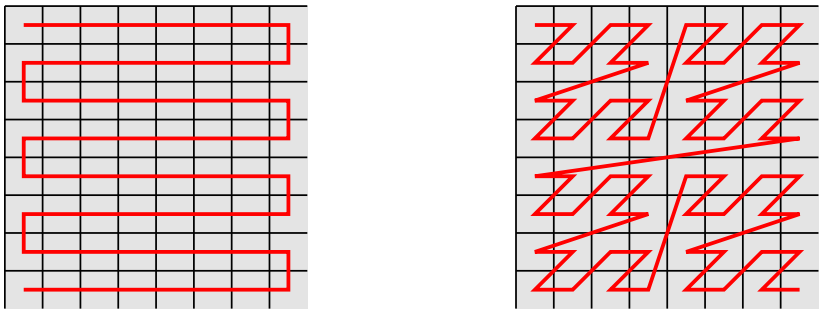
5.4 Reverse Time Migration Proto-Application

With the purpose of evaluating the multi-threading ability of MPI runtimes on a scientific application, we have developed our own parallel seismic modeling proto-application that mimics a real Reverse Time Migration (RTM) application. RTM [Ort+08] is an imaging algorithm used in seismic exploration for geologically complex subsurface areas. For simplification reasons, our proto-application (called RTM-*proto* in the remainder of the thesis) does not solve the standard two-way wave equation and the model is limited to the forward step, which simulates a wave propagation into an isotropic media. The one-way wave equation is solved using a

[3]The Large Cluster is detailed in section 2.3.3

order-8 in 3D space stencil where updating a stencil point requires the values of the 24 neighbor points.

We designed the full-MPI version of RTM-*proto* using a regular domain-decomposition method. As a first optimization to decrease the memory access costs when updating the stencil, our RTM-*proto* application applies a *blocking* technique. This technique aims at filling the cache that is closer to the CPU for preserving the locality of data being accessed. In addition, the blocks along the x-axis have been enlarged as much as possible in order to maximize linear accesses to the memory and to allow the CPU to pre-fetch useful data. To enforce data locality, the computational units (MPI tasks or OpenMP threads) process the stencil blocks following a *z-curve* pattern which increases the locality of blocks for a later buffer reuse (see figure 5.7). Although this optimization exhibits a minor advantage with the full MPI code, it shows significant speedups when applied on hybrid versions. Moreover, nowadays operating systems often rely on `first-touch` policies where virtual memory pages are physically allocated only when the application accesses memory pages for the first time. Thus, in order to prevent memory pages to be allocated during the first iteration, each computational unit (MPI task or OpenMP thread) first-touches its stencil blocks before entering the computation loop. Moreover, this strategy guarantees that stencil blocks are allocated in the local NUMA node in hybrid mode.



(a) Linear function **(b)** Z-order curve

Figure 5.7 – Different functions for accessing data in a two-dimension mesh. The linear function in figure (a) does not preserve locality of data whilst the Z-order curve in figure (b) preserves locality for improving later cache reuse.

The domain of RTM-*proto* is decomposed using the set of cartesian functions provided by the MPI interface. However, since most implementations of the MPI cartesian constructors barely consider the underlying topology of the machine (e.g., MPC and Intel MPI linearly distribute MPI tasks in the mesh), we designed our own cartesian interface that maps the MPI tasks in a 3D cartesian mesh using a *z-curve* function [Tra03]. Moreover, this user-level mapping ensures that the task mapping is constant from a runtime to another.

To deal with non-contiguous messages due to halo-exchange, sender and receiver tasks respectively pack and unpack halos into contiguous user-level buffers. Although the MPI standard provides an interface for sending non-contiguous data, we do not use this interface. We will detail the reasons of this choice later in section 5.4.2.

As depicted in listing 5.1, RTM-*proto* is based on persistent point-to-point communications that are started with `MPI_Start(all)` routines. Moreover, no collective

operations are involved in the computational loop. As regards communication operations, receive buffers are posted at the beginning of each iteration to prevent unexpected messages (line 16) and halos are sent after the stencil computation (line 27). Because communication overlapping usually shows a minimal speedup and complicates the writing of code (see chapter 4), the application only uses the blocking semantics of MPI. Moreover, the MPI tasks wait the completion of all outstanding send and receive requests using a call to `MPI_Waitany` (line 33). Unlike `MPI_Waitall`, `MPI_Waitany` enables a slight opportunity for the application to overlap communications with useful computation. Indeed, as soon a message is received, the communication buffer is unpacked to the stencil (line 36) and other communications not yet received may progress asynchronously to the unpacking operation. To conclude with the full MPI version of RTM-*proto*, it obviously only requires the `MPI_THREAD_SINGLE` level of thread-safety.

Listing 5.1 – Full MPI version of RTM-*proto* (simplified source code)

```
1  int main (int argc, char **argv) {
2    MPI_Request req[12];
3    int t, i;
4    initialize_domain();
5    initialized_mpi_buffers();
6    MPI_Init(&argc, &argv); /* equivalent to MPI_THREAD_SINGLE */
7
8    /* Initialize MPI buffers */
9    for(i=0; i<6; ++i) {
10     MPI_Send_init(..., neighbor[i], ..., &req[i]);
11     MPI_Recv_init(..., neighbor[i], ..., &req[i+6]);
12   }
13
14   for (t=0; t<t_max; ++t) {
15     /* Post receive buffers */
16     MPI_Startall(6, &req[6]);                          start
17
18     /* Compute stencil blocks */
19     for(i=0; i<blocks_max; i++) {
20       compute_stencil_block(i);                        stencil
21     }
22
23     for(i=0; i<6; ++i) {
24       /* pack halos */
25       pack_halos(i);                                   halos
26       /* Send halos */
27       MPI_Start(&req[i]);                              start
28     }
29
30     for(i=0; i<12; ++i) {
31       int neig_index;
32       /* Wait for all communications */
33       MPI_Waitany(12, req, &neig_index, ...);          wait
34       if (neig_index >=6) {
35         /* Unpack halos if receive request */
36         unpack_halos(neig_index-6);                    halos
37       }
38     }
39     swap_domain_pointers();
40   }
41   MPI_Finalize();
42 }
```

5.4.1 Hybrid RTM-*proto*

In the following section, we give the implementation details of three hybrid versions of RTM-*proto* and we present some guidelines to any developer who would hybridrize

an MPI domain-decomposition code with OpenMP. First, the MPI communications of the master-only version are performed inside an OpenMP `Master` region. Then, the hybrid version with taskification of communications encapsulates each MPI communication into taks, which can be processed in parallel by the OpenMP threads. Finally, the domain-decomposition design proposes a coding style close to the SPMD programming.

Listing 5.2 – `MASTER` version of RTM-*proto* (simplified source code)

```
#pragma omp parallel
for (t=0; t<t_max; ++t) {
  #pragma omp master
  {
    /* Post receive buffers */
    MPI_Startall(6, &req[6]);
  }

  /* Compute stencil blocks */
  #pragma omp for schedule(static)
  for(i=0; i<blocks_max; i++) {
    compute_block(i);
  }
  #pragma omp barrier

  #pragma omp master
  {
    /* Send halos */
    for(i=0; i<6; ++i) {
      pack_halos(i);
      MPI_Start(&req[i]);
    }

    /* Wait for communications */
    for(i=0; i<12; ++i) {
      int neig_index;
      MPI_Waitany(12, req,
        &neig_index, ...);
      if (neig_index >=6) {
        /* Unpack halos if
           receive request */
        unpack_halos(neig_index-6);
      }
    }

    swap_domain_pointers();
  }
  #pragma omp barrier
}
```

Listing 5.3 – `DD` version of RTM-*proto* (simplified source code)

```
#pragma omp parallel
for (t=0; t<t_max; ++t) {
  /* Post receive buffers */
  MPI_Startall(6, &req[6]);

  /* Compute stencil blocks */
  for(i=0; i<blocks_max; i++) {
    compute_block(i);
  }

  /* Send halos */
  for(i=0; i<6; ++i) {
    pack_halos(i);
    MPI_Start(&req[i]);
  }

  /* Wait for all communications */
  for(i=0; i<12; ++i) {
    int neig_index;
    MPI_Waitany(12, req,
      &neig_index, ...);
    if (neig_index >=6) {
      /* Unpack halos if
         receive request */
      unpack_halos(neig_index-6);
    }
  }

  #pragma omp barrier
  #pragma omp master
  {
    swap_domain_pointers();
  }
  #pragma omp barrier
}
```

5.4.1.1 Hybrid Programming with Single-Threaded Communications

The `MASTER` version depicted in listing 5.2 is parallelized using the incremental approach described in section 5.1.1. The main *for*-loop is surrounded by a call to `pragma omp for` construct to distribute stencil blocks among OpenMP threads (line 10). Moreover, we statically decomposed the work using the `schedule(static)` attribute since the workload of RTM-*proto* is well balanced. At the end of the sten-

cil computation, the OpenMP threads are synchronized using a barrier (line 14) and the master thread starts all inter-node communications before waiting for them using a call to `MPI_Waitany` (lines 16 to 35).

One benefit of this version is that it requires a low-level of thread safety (`MPI_THREAD_FUNNELED`). Indeed, only the master thread is allowed to communicate and no concurrent calls to the runtime are performed. Moreover, other threads than the master thread are idle during all MPI communications because no work can be performed until the communications are completed.

In addition to MPI communications, the master thread has also to manage halos packing and unpacking (lines 20 and 32). These operations consequently increase the execution time, especially if they are sequentially executed. To reduce this overhead, the buffer packing and unpacking functions have been parallelized and decomposed into OpenMP tasks (8 tasks per communication buffer). These tasks are generated by the master thread and distributed by the OpenMP runtime to any idle thread.

5.4.1.2 Hybrid Programming with Manual Taskification of Communications

The `TASKS` version is similar to the `MASTER` version, except that every MPI communication is encapsulated inside an OpenMP task region and all these regions can be executed in parallel. Because several OpenMP tasks can be concurrently executed, the `MPI_THREAD_MULTIPLE` is required for this version.

One of the major issues of the OpenMP standard is actually the lack of optimizations for hierarchical memory systems such as NUMA architectures [Jin+11]. For example with the task directive, it is uncertain that the same task will be performed by the same thread between two consecutive stencil iterations. For this hybrid version of RTM-*proto*, the `task` construct has been removed and replaced with a regular `if`-statement. Communication tasks are statically distributed among available OpenMP threads and each thread selects which task it has to perform thanks to its thread ID. This transformation is legitimate as long as they are as many or more physical cores as OpenMP tasks and if no imbalance is noticed between threads entering the communication tasks. For RTM-*proto*, it limits the execution on machines with at least 12 cores per compute node (the 3D-cartesian mesh requires 12 communications tasks). Moreover, a moderate imbalance is observed since the OpenMP threads are explicitly synchronized before entering the `task` directive.

5.4.1.3 Highly Parallel Hybrid Programming using Domain Decomposition

The Domain Decomposition (`DD`) code depicted in listing 5.3 is similar to the SPMD programming previously described in section 5.1.2. The domain is equitably divided into subdomains as for the full MPI version and subdomains are distributed to OpenMP threads according to their thread IDs. At the beginning of the code, each thread determines (according to its ID) the subdomain it has to process and initializes it. Inside a node, the stencil is contiguously allocated in memory and whenever an OpenMP thread needs a value from another subdomain, it directly accesses it by reading the main memory. Previous studies have shown good performance using the SPMD programming where halos are privatized into thread-private

memory regions [KC03; Ber+05]. This approach has however not been investigated in this thesis since it requires as many communication buffers as the full MPI versions.

This hybrid version of RTM-*proto* actually provides the maximum parallelism since all OpenMP threads participate to MPI communications. The only sequential part of the code is a synchronization barrier used to wait the completion of MPI communications and to swap pointers referencing the current and next domains. To maximize locality between OpenMP threads, the subdomains are distributed to the OpenMP threads following a *z-curve* function previously presented in figure 5.7(b). This distribution ensures that the neighborhood of an OpenMP thread is physically close and the memory accesses are consequently more efficient. Finally, the application associates a *tag* to every MPI message in order to distinguish a message from an OpenMP thread to another. Before being sent, MPI messages are tagged with the ID of the remote thread and receiver threads tag the corresponding receive requests with their own ID.

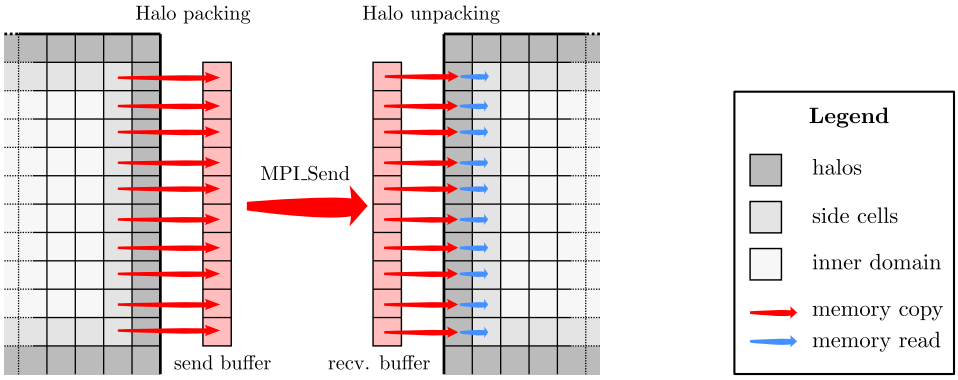
5.4.2 Discussion on Non-Contiguous Data

To deal with non-contiguous messages, the MPI standard provides several methods to define user-level data types, which are called *derived* data types. With derived data types, buffer packing and unpacking operations are not performed by the application anymore and the runtime has now the opportunity to optimize the transmission of non-contiguous messages. For our RTM-*proto* application, we however did not use this interface and halos are manually managed as depicted in figure 5.8(a). They are several reasons to this. First, specifying the layout of the non-contiguous data is usually a laborious work, especially for multi-dimensional cartesian meshes where the gap between segmented data is not always constant. Second, most MPI runtimes do not efficiently implement data types and non-contiguous data are copied into buffers which are internally managed by the runtime. This behavior is consequently similar to the one manually performed by the application and no speedup is expected from derived data types. As depicted in figure 5.8(b), a few MPI runtimes are actually able to suppress two memory copies when using derived data types in a shared-memory context. This is for example the case with thread-based MPI runtimes where an MPI task can directly access the data from another task inside a compute node. However, although data are directly accessible in shared memory, the MPI standard still requires a memory copy. Furthermore, it is uncertain if the network efficiently enables zero-copy mechanism for derived data types [WWP04].

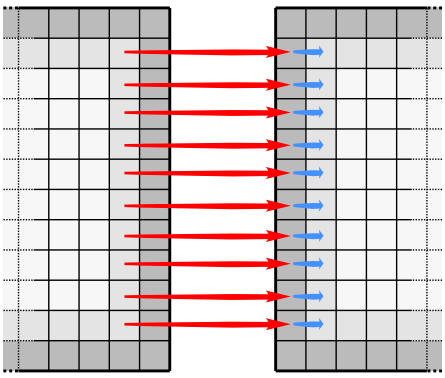
In a hybrid context where one unique MPI task is allocated per compute node, MPI is not used for intra-node communications and the OpenMP threads directly access data using memory read operations (see figure 5.8(c)). As a consequence, the memory traffic is reduced and because no halos are involved for intra-node communications, more memory is available for larger simulations.

5.4.3 Experimental Results

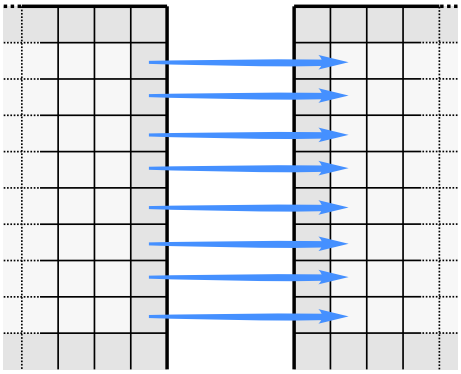
In the following section, we compare the performance of the three hybrid versions and the full-MPI version of RTM-*proto* previously described in section 5.4. We first detail an algorithm for progressing MPI messages in a multi-threaded context. This optimization aims at minimizing the memory traffic while polling, which consequently improves the computation time of memory-bandwidth bound applications



(a) Manual halo packing/unpacking with MPI



(b) Halo swaps with MPI and *derived* data types



(c) Direct access to data with OpenMP

Figure 5.8 – Halo swaps with the domain decomposition method in a shared-memory context. In figure (a), the user manually packs and unpacks halos into contiguous buffers and three memory copies are involved. In figure (b), optimized MPI derived data types are used and the runtime can suppress two memory copies. This is for example the case with thread-based MPI. With OpenMP in figure (c), threads directly access data from the neighborhood and no more halos are allocated for intra-node communications.

like our RTM-*proto* application. We then present the results of RTM-*proto* on top of MPC and Intel MPI, two MPI runtimes implementing the `MPI_THREAD_MULTIPLE` level of thread-safety with a support of high-speed networks.

5.4.3.1 Multi-threaded Message Progression

In hybrid mode with one MPI task per compute node, a unique set of network endpoints is generally allocated for the entire compute node. Moreover and as we will discuss in section 5.5.1, the MPI standard combined with a shared memory programming model makes difficult the efficient use of several *vrails* (concept previously presented in section 4.4.2). As a consequence to a polling-based message progression in hybrid mode, all idle threads attempt to access the same CQs, which leads to a high memory traffic and a high contention on these structures. For example with RTM-*proto*, we have experimented significant slowdown and a large imbalance of the computation functions while using the polling-based message progression.

The Infiniband specifications propose the event-driven method, which does not

require to access the completion queues to detect the arrival of new messages. However, generating an event for each incoming message can lead to significant overhead. First, on fully-subscribed systems, the progression thread, which is scheduled following an interruption can be executed on any core and would potentially de-schedule another computing thread. Then, section 2.2.4 has previously shown that an event driven message progression can be subject to overheads due to a longer reactivity than the polling-based to detect incoming messages.

To minimize the memory traffic while waiting for network messages, we designed the multi-threaded progression policy depicted in algorithm 2, which optimizes the polling of completion queues. The CQ polling is moved to a critical section only accessible by a unique thread and which extends from line 7 to line 25. As for Collaborative-Polling presented in chapter 4, each MPI task implements a pending list which stores the CQEs waiting for being processed by the task. Moreover, if several threads (e.g., OpenMP threads) from the same MPI tasks are idle, multiple CQEs from the same pending list can be processed in parallel.

When a thread waits a message, it first tries to poll the pending list of the MPI tasks it belongs to (line 2). If no messages are found and if the critical section is free (line 7), the polling thread accesses the CQs (line 10) and disseminates CQEs into the appropriate pending lists. Otherwise the critical section is busy and the polling thread goes to sleep (line 27). In line 11, some CQEs may be waiting in the pending lists. In this case, the polling thread releases other threads waiting on the conditions (line 27) and leaves the polling function since the incoming CQEs may be for any threads. Otherwise if no CQEs are pending and if the polling thread has called the progress function in a blocking-mode (line 17), it returns on line 10 and polls again the CQs. This technique aims at maximizing cache reuse as the same thread successively accesses the CQs and data related to polling are already cached. In those cases where the polling thread calls the progression function in a non-blocking mode, a signal is emitted on the condition (line 22), the next thread to poll is scheduled and the polling thread leaves the progression function.

5.4.3.2 Conditions of the Evaluation

In the following section, we detail the environment of the evaluation. Both MPI runtimes and all versions of RTM-*proto* have been compiled with the same Intel Compiler and the same OpenMP runtime provided by the Intel Compiler suite version 13.0.0.079. Moreover, we enabled the `-O3` compilation flag and `-xHost`/`-xAVX` flags are selected according to the underlying architecture. Concerning MPC, its scheduler has been disabled in hybrid mode and the scheduling of the OpenMP threads is now only managed by the OS. All RTM-*proto* versions have been instrumented with RDTSC counters[4], which register the start and the end of each regions into thread-private arrays. Furthermore, experiments have proven that the instrumentation interface has a negligible impact on performance (1% overhead).

[4]The Read Time Stamp Counter (RTSC) counts the number of CPU cycles since reset

Algorithm 2 Multi-threaded message progression for hybrid programming

Input:

- *busy*: shared variable to determine if the current task is the first to enter the polling function.
- *cond*: shared condition variable used for signaling threads waiting on the polling function.
- *mutex*: shared mutex associated to the condition variable *cond*.
- *retry*: shared variable used for indicating if the polling function must be re-executed by the next thread to poll ($retry = 1$).
- *blocked*: integer that indicates whether the polling function is called in blocking mode (e.g., `MPI_Wait`) or not (e.g., `MPI_Test`).

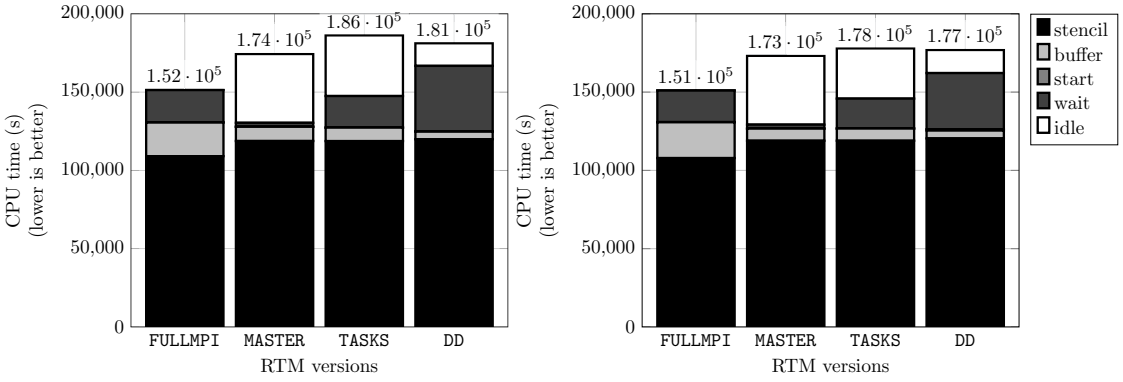
```
1: function PROGRESS_FUNCTION
2:     POLL_PENDING_LISTS
3:     if message found then
4:         return
5:     end if
6:     MUTEX_LOCK(mutex)
7:     if busy == 0 then                          ▷ If nobody is currently polling
8:         busy = 1
9:         MUTEX_UNLOCK(mutex)
10:        POLL_CQ                          ▷ Poll Send and Recv Completion Queues
11:        if at least 1 CQE waiting on pending lists then
12:            retry = 0
13:            MUTEX_LOCK(mutex)
14:            COND_BROADCAST(cond)
15:            MUTEX_UNLOCK(mutex)
16:        else
17:            if blocking = 1 then              ▷ Function called in blocking mode
18:                go to line 10
19:            else
20:                retry = 1
21:                MUTEX_LOCK(mutex)
22:                COND_SIGNAL(cond)          ▷ Release the next thread to poll
23:                MUTEX_UNLOCK(mutex)
24:            end if
25:        end if
26:    else
27:        COND_WAIT(cond)                    ▷ Non-polling threads are waiting
28:        if retry == 1 then
29:            go to line 7
30:        end if
31:        MUTEX_UNLOCK(mutex)
32:    end if
33: end function
```

Figures 5.9 to 5.11 decompose the CPU time of RTM-*proto* into six different regions. These regions are represented in listing 5.1 and are detailed as follows:

- **stencil**: stencil update. With hybrid versions, this timer implicitly includes the time spent by OpenMP threads for accessing neighbor halos in shared memory;
- **buffer**: halo packing and unpacking;
- **start**: MPI buffer posting (send and receive requests initiated with `MPI_Start(all)`;
- **wait**: idle time due to MPI communications (e.g., `MPI_Waitall`, `MPI_Wait`, `MPI_Waitany`);
- **idle**: idle time due to OpenMP synchronization directives (e.g., `Barrier`, `Master`). This time is actually the total time subtracted from the time inside each individual regions and may include the overhead due to the instrumentation;

`MASTER` (see section 5.4.1.1), `TASKS` (see section 5.4.1.2), and `DD` (see section 5.4.1.3), are respectively referring to as the version where only the master thread communicates using MPI, the version encapsulating MPI communications into OpenMP tasks and the hybrid domain decomposition version where OpenMP threads independently communicate inter-node halos using MPI.

The reference point to compare hybrid versions is the full MPI version that is denoted as `FULLMPI`.



(a) MPC results (4 *vrails*, 1 HCA) **(b)** Intel MPI results

Figure 5.9 – Comparison between RTM-*proto* `FULLMPI`, `MASTER`, `TASKS` and `DD` versions on *Thin Cluster*, 2,048 cores (128 MPI tasks, 16 OpenMP threads per MPI task, $2,560^3$ domain size, 2,000 iterations)

5.4.3.3 Thin Compute Nodes

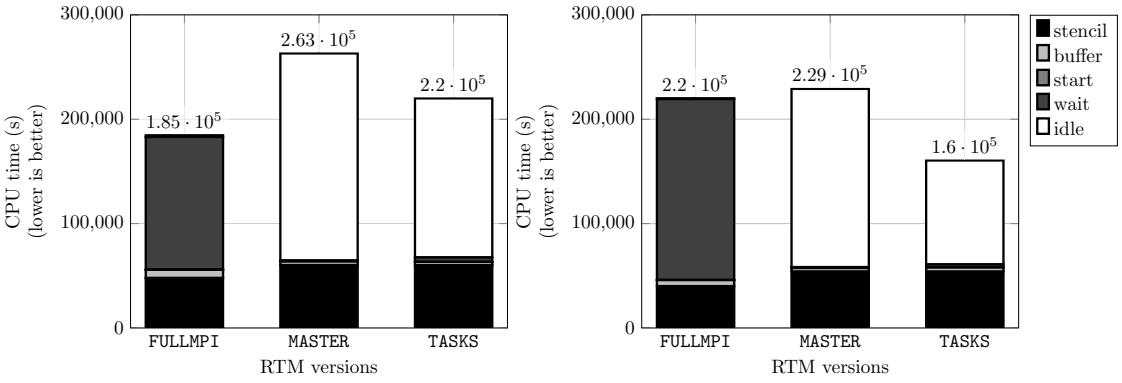
Figure 5.9 reports the results of all RTM-*proto* versions with MPC and Intel MPI on 128 compute nodes from the *Thin Cluster* for a total of 2,048 physical cores and a domain of size $2,560^3$. As we can see, both runtimes perform similarly on all versions of the code and the introduction of OpenMP has two direct implications compared to the `FULLMPI` version. First, the time inside halo packing and unpacking is significantly reduced since there is no halo for intra-node communications. Second,

we notice a slight overhead in stencil computation in hybrid versions. It is due to the OpenMP threads that require distant NUMA accesses while updating cell on the subdomain edges. This observation is not true for the `FULLMPI` version since no data are shared and memory accesses are exclusively local to the NUMA node.

We now compare the three hybrid versions together. The figure shows that the `MASTER` version dramatically reduces the time in MPI *wait* functions. Nevertheless, as only one thread participates to MPI communications, it represents a large portion of code that cannot be parallelized. As a result, the Amdhal's law does apply on MPI communications and the idle time increases. The `TASKS` version should *a priori* outperform the `MASTER` version as it parallelizes communications on 12 threads and reduces the sequential portion of the code. This expectation is actually not true for the two runtimes and the `TASKS` version gets a high overhead in the *wait* functions. On 16-core compute nodes, the `DD` version also fails to outperform the `FULLMPI` version. Although the time spent inside buffer packing/unpacking and idle regions is low with the `DD` version, the communication time is larger than the `FULLMPI` version due to the overhead generated by multi-threaded accesses to the MPI runtime.

5.4.3.4 Large Compute Nodes with Single HCA

Figure 5.10 reports the results of RTM-*proto* on 16 compute nodes from the *Large Cluster* with 2,048 MPI tasks, a domain size of $2,560^3$ and 500 iterations. Although the cluster embeds 4 HCA, we only use a single HCA for these experiments and multirail configurations will later be discussed in section 5.5. MPC sets up 16 *vrails* since this configuration gives the best performance and the runtime was configured for using one HCA. Moreover, the multi-threaded algorithm presented in Listing 2 was implemented into MPC and activated for this evaluation on these large compute nodes. Regarding Intel MPI, it accesses the Infiniband network through the DAPL fabric and uses the same HCA than MPC.



(a) MPC results (16 *vrails*, 1 HCA) **(b)** Intel MPI results

Figure 5.10 – Comparison between RTM-*proto* `FULLMPI`, `MASTER` and `TASKS` versions on the *Large Cluster*, 2,048 cores (16 MPI tasks, 128 OpenMP threads per MPI task, $2,560^3$ domain size, 500 iterations).

When focusing on `MASTER` and `TASKS` versions, both versions significantly reduce the time inside *wait* and buffer packing/unpacking functions. However, as expected with the increasing number of cores, these two hybrid versions generate a lot of imbalance as only a minor subset of OpenMP threads communicates using MPI. It results in a high idle time, even for the `TASKS` version that only parallelizes

communications on 12 tasks while the 116 other cores are idle. We now compare the `MASTER` and `TASKS` versions to the `FULLMPI` version. As we can see on the figure, Intel MPI outperforms MPC on the `MASTER` and `TASKS` versions respectively with a speedup of 1.15 and 1.38. We suppose it is due to Infiniband communications that are better optimized by the Intel runtime than they are for MPC. As an example with the `MASTER` version, an overhead of x seconds in the MPI communications consequently results in $127 \times x$ seconds of additional idle time. In this case and compared to Intel MPI, MPC gets an overhead of 17% in the *wait* functions, which in part contributes to the 16% overhead of the idle time. This observation is not valid for the `FULLMPI` version where MPC outperforms Intel MPI certainly because of the single copy of intra-node communications (see section 5.4.2).

Figure 5.11 focuses on the results from the `DD` version. The `IMPI/WAIT` runtime refers to as Intel MPI with the *WAIT* mode activated. According to the documentation from Intel, this mode allows the MPI tasks to wait the reception of messages without polling the network endpoints. The `IMPI/SOCKET` version corresponds to Intel MPI configured with one MPI rank per level-2 NUMA node for a total of 64 MPI ranks and 32 OpenMP threads per MPI rank (`I_MPI_PIN_DOMAIN=omp:compact`).

As shown on the figure, MPC reduces by a factor of 3.1 the time inside the `MPI_Wait` functions from the `FULLMPI` version to the `DD` version. In the *Large Cluster*, NUMA effects are substantial, especially for communications traversing the BCS (see figure 1.5 in section 1.3.6). Without optimized derived data types, section 5.4.2 has previously established that the communication model of MPI requires at least three memory copies for swapping halos in a shared-memory context. We believe that the `FULLMPI` version generates much more memory traffic than the `DD` version. Thus, the `FULLMPI` consequently leads to a larger portion of time in `MPI_Wait` functions due to the saturation of the memory bandwidth. Moreover, the `DD` version fully parallelizes the network communications compared to other hybrid versions. Indeed, it has to be noticed that the same amount of data is exchanged in the network between the three hybrid versions. The only difference is that messages are shorted and in largest number for the `DD` version. Consequently to the higher number of messages, communications benefit from a higher potential of overlap.

With Intel MPI in its default configuration (denoted as `IMPI` in figure 5.11), the `DD` version gets a huge overhead due to the 128 OpenMP threads simultaneously accessing the MPI runtime. We believe that Intel MPI does not efficiently allow several threads to process message reception in parallel, which leads to significantly delay the message progression. The `WAIT` mode, generally used for saving CPU time in a *fully-subscribed* mode also preserves the memory bandwidth. At some point, the `WAIT` mode improves the execution time and a speedup of 1.16 is observed inside *wait* functions. Despite this improvement, Intel MPI however runs the `DD` version 4.32 times slower than MPC. Regarding Intel MPI with one rank per level-2 NUMA node (`IMPI/SOCKET`), it solves a large part of the performance issue and enhances the execution of the `DD` version by a factor of 4.7. This optimization is nevertheless insufficient to reach the performance of MPC. It is due to a larger amount of time inside *wait* functions because of intra-node MPI communications between shared memory tasks.

Table 5.1 finally reports the physical memory used for the `FULLMPI` and `DD` versions of RTM-*proto* with MPC and Intel MPI. From the total memory used, the table extracts the memory dynamically allocated by the application, which includes various buffers, array of MPI requests and the stencil grid. Regarding the mem-

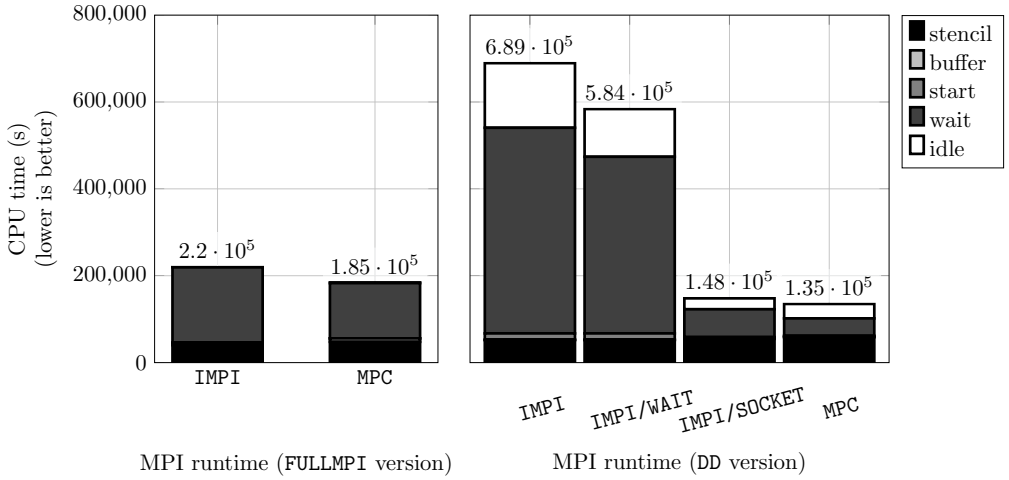


Figure 5.11 – Comparison between Intel MPI in various configurations and MPC on `FULLMPI` and `DD` versions with 2,048 cores from the *Large Cluster* (16 MPI tasks, 128 OpenMP threads per MPI task, $2,560^3$ domain size, 500 iterations). `IMPI/WAIT` refers to as Intel MPI with `WAIT`-mode activated whereas `IMPI/SOCKET` creates one MPI task per level-2 NUMA node

ory allocated by the application, 17.2% of memory is saved by mixing MPI with OpenMP compared to the full-MPI version of the code. This gain is due to halos, which are not duplicated for intra-node communications between OpenMP threads. Furthermore, we notice that more memory than 17.2% is actually saved with 19.9 % of total memory for MPC and 28.5 % for Intel MPI compared to the full-MPI RTM-*proto* code. Indeed, combining MPI with OpenMP reduces by a factor of 128 the total number of MPI runtimes instantiated.

One would also notice that MPC consumes more memory than Intel MPI in hybrid mode. It is because, in hybrid mode, MPC allocates some resources that scale according to the number of cores on the machine. It includes for examples network buffers over all NUMA nodes (see section 3.3.3). As far as we know, the MPI runtime from Intel keeps a constant amount of memory per process, whatever the underlying architecture and the number of cores available in hybrid mode.

Intel MPI with one MPI rank allocated per level-2 NUMA node (Intel MPI/`SOCKET`) is a relevant trade-off. Indeed, the total memory saved in this configuration is 26.6 % against 28.5 % for the regular `DD` version and the application gets a speedup of 4.67 compared to the full-MPI version. This configuration is however suboptimal in both execution time and memory since it implies additional buffers for halos and intra-node communications using MPI.

5.5 Limitations to Hybrid Mode

For runtime designers, an efficient hybrid support implies to carefully consider the overhead due to concurrent accesses to the runtime internals. This point includes for example the queues for message matching, the MPI requests (since two threads cannot simultaneously use the same request ID) and operations on the registration cache and the pool of network buffers.

		Memory (total)		Memory (application)	
Version	Runtime	MB	% saved	MB	% saved
FULLMPI	MPC	168,462		159,448	
	Intel MPI	187,114			
DD	MPC	134,937	19.9	132,053	17.2
	Intel MPI	133,734	28.5		
DD	Intel MPI/SOCKET	137,425	26.6	136,460	14.4

Table 5.1 – Comparison of the physical memory used for the FULLMPI and DD versions of RTM-*proto* with MPC and Intel MPI

Regarding the MPI standard, it was not originally defined to allow multiple threads to communicate independently: they have no MPI rank and they constantly need to access the context of the MPI task they belong to for communicating. This limitation is actually due to the matching semantics of MPI that ignores the notion of threads. Let us consider the hybrid configuration depicted in figure 5.12. Two OpenMP threads share the same address space and concurrently communicate independent messages to the same remote MPI task but for two different destination threads. This communication pattern is what is typically happening in our DD version of RTM-*proto*. In such a case, the two messages conflict since they access the context of the same MPI task while thread-to-thread communications are independent. As a consequence, the critical path involves multiple synchronization points, which results in overhead.

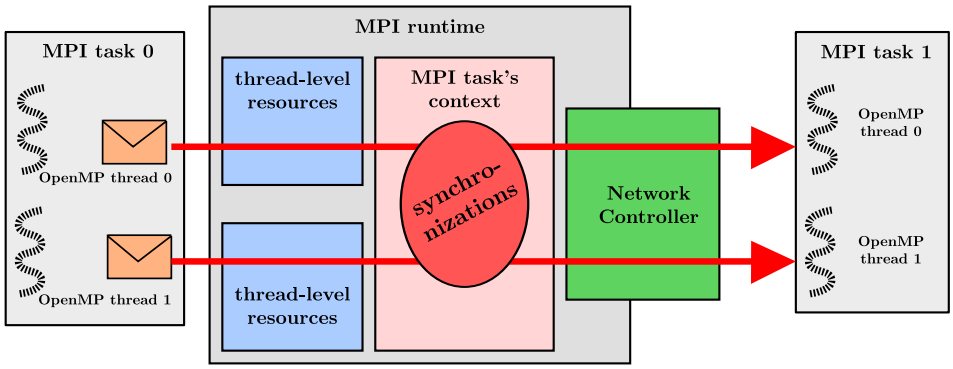


Figure 5.12 – Thread-to-thread communications using the actual interface of the MPI standard. A synchronization is required between thread 0 and thread 1 to access the context of the MPI task they belong to.

Another drawback of the current MPI interface concerns multi-HCA and multi-port configurations. In the previous section 5.4.3, the evaluation was performed using only one HCA and, in this case, the DD version obtained the best results. Figure 5.13 now compares the FULLMPI version of RTM-*proto* with 1 and 4 HCA(s) (both using 16 *vrails*) to the hybrid DD version. As we can see, the FULLMPI version using 4 HCAs outperforms the one using 1 HCA with a speedup of 4 in *wait* functions. Compared to the DD version, the FULLMPI version with 4 HCAs is now the configuration that gives the best results compared to other hybrid versions.

With the current MPI interface, providing an efficient multi-rail configuration for hybrid applications is not an easy operation. First, it is impossible for the runtime to determine if two messages being exchanged by two different threads are

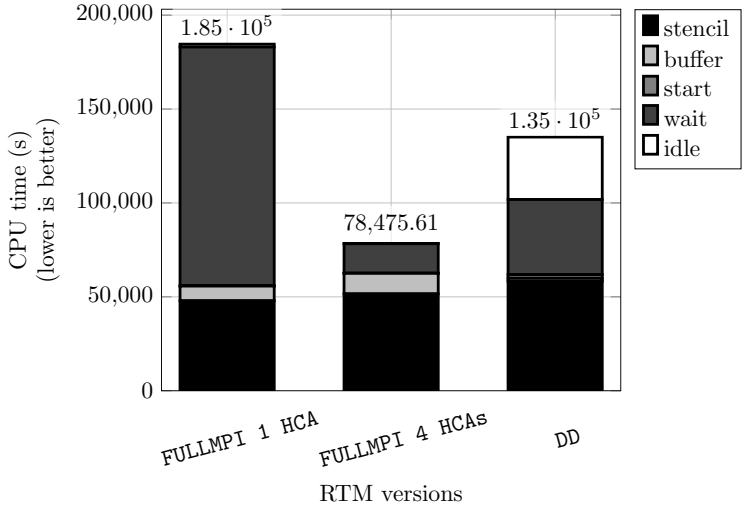


Figure 5.13 – Comparison between the `FULLMPI` version of RTM-*proto* with 1 HCA, the `FULLMPI` version with 4 HCAs and the DD hybrid code on 2,048 cores from the *Large Cluster* (16 MPI tasks, 128 OpenMP threads per MPI task)

independent. Indeed, this information is manually handled by the user who generally tags MPI messages with a unique value for each pair of threads. A second solution which is barely used consists in allocating one distinct communicator for each pair of threads. One runtime aware of these user-specific information would be able to independently route messages to different HCAs according to the MPI tag or the value of the communicator. It is however evident that the two solutions are both (1) not portable from an MPI runtime to another and, (2) inconvenient for users since applications must be rewritten to integrate such a design.

A second issue is the lack of topology information on where the threads are executing in hybrid mode. As an example, the current OpenMP standard [Ope13] does not expose any interface that could be used by MPI runtimes to topologically locate OpenMP threads on the compute nodes. Focusing on our *vrail* contribution proposed in section 4.4.2, the receiver-driven routing strategy requires the runtime to collect where the remote threads are running. Since the MPI runtime cannot identify a thread among others, this solution simply becomes unenforceable. As a second proposition, the sender-driven message routing strategy could be used to send messages through the closest HCA from the sender thread. This solution has however been previously discarded because non-optimal. In fact, the only way for achieving multi-rail optimizations for hybrid applications is to split every single MPI message in as many fragments as HCAs available on the compute node and send each fragment through different HCAs. This optimization is already available in mainstream MPI runtimes and efficiently improves the communications of multi-rail configurations [LVP04; MGN10]. Stripping messages however requires to consider some overhead because the receiver has to poll all HCAs to recompose the previously split message.

5.5.1 MPI Endpoints and Unified Runtimes

The concept of MPI endpoints was originally designed as a part of the MPI 3.0 standard. Several interfaces were explored for integrating MPI endpoints to the standard but the MPI Forum finally delayed this concept.

We define as an MPI endpoint an entity that is able to directly and independently communicate MPI messages. Conceptually, an MPI endpoint corresponds to an MPI rank attached into a specific communicator. Once the communicator created and the endpoints allocated, the threads can attach to endpoints and make MPI calls using the resources of this endpoint. Let us consider this concept with a 1:1 mapping, where an MPI endpoint is accessible by a unique OpenMP thread. In such a configuration, an OpenMP thread could directly communicate data to a remote OpenMP thread and thread-to-thread communications could independently perform in parallel. Thus, MPI runtime becomes aware of the overlying running threads (e.g., OpenMP threads) and considers them as fully-fledged MPI ranks. As a consequence, matching and sequence numbering occur at a thread level and the overheads spotted in section 5.3.2 are eliminated.

Recently, Dinan *et al.* exposed a new dynamic semantics that does not restrain the maximum number of threads to a static value [Din+13]. This powerful interface keeps the backward compatibility with the current MPI interface and only requires a few new MPI functions to be implemented. In the research paper, the authors give an example of their interface using OpenMP. This example is depicted in listing 5.4 and detailed bellow.

For each parallel region, a communicator inherited from `MPI_COMM_WORLD` is created by calling `MPI_Comm_create_endpoints`. It should be noted that this operation could slow down the parallel region since (1) it involves a collective operation and, (2) this communication needs to be protected by a `master` directive. As a consequence, this solution would not be efficient with the fine-grain parallelization described in section 5.1.1 where parallel sections are generally short and numerous. Once the creation of endpoints successfully accomplished, *nt* endpoints are created and *ep_comm* contains as many communicators as the number of endpoints (i.e., *n*). Finally, each thread attaches to the communicator associated to the endpoint and this communicator is used to distinguish an OpenMP thread among others in MPI calls (line 21 and 23).

Listing 5.4 – Example of a hybrid MPI+OpenMP program where endpoints are used to enable all OpenMP threads to participale in a collective `MPI_Allreduce` operation (listing extracted from [Din+13])

```
1  int main(int argc, char **argv) {
2    int world_rank, tl;
3    int max_threads = omp_get_max_threads();
4    MPI_Comm ep_comm[max_threads];
5
6    MPI_Init_thread(&argc, &argv, MULTIPLE, &tl);
7    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
8
9  #pragma omp parallel
10   {
11     int nt = omp_get_num_threads();
12     int tn = omp_get_thread_num();
13     int ep_rank;
14 #pragma omp master
15     {
16       MPI_Comm_create_endpoints(MPI_COMM_WORLD,
17         nt, MPI_INFO_NULL, ep_comm);
18     }
```

```
19 #pragma omp barrier
20         MPI_Comm_attach(ep_comm[tn]);
21         MPI_Comm_rank(ep_comm[tn], &ep_rank);
22         ... // divide up work based on 'ep_rank'
23         MPI_Allreduce(..., ep_comm[tn]);
24
25         MPI_Comm_free(&ep_comm[tn]);
26       }
27       MPI_Finalize();
28     }
```

A unified runtime combines several programming models into the same execution model [PJN08]. Programming models can consequently collaborate together, share the same resources in memory or communicate informations concerning their own behavior.

We believe that a unified MPI+OpenMP would give a convenient solution for designing the concept of MPI endpoints. First and in order to use our *vrail* design with the receiver-driven routing strategy depicted in section 4.4.2, threads must not migrate between NUMA nodes. Indeed, if a thread migrates and in the meanwhile, it receives a message in the wrong NUMA node, this message could never be received. In this case, a unified runtime would allow the OpenMP runtime to communicate the thread binding policy and the topological location of every OpenMP threads to the MPI runtime. Thus, our *vrail* design with the receiver-driven routing strategy could be used in this case. Second, MPI endpoints using the interface presented in 5.4 must be re-created for each new parallel region. It is because the placement and the number of OpenMP threads can change between consecutive parallel regions. In this context, if the OpenMP runtime assumes the same configuration of threads from a parallel region to another, the MPI runtime could reuse the same endpoints and skip their creation.

As a future work, we plan to implement MPI endpoints through the dynamic interface proposed by Dinan *et al.* [Din+13]. Because MPC provides a unified scheduler for hybrid MPI+OpenMP programming, we plan to develop this interface inside MPC. The section 5.3.2 has however demonstrated that a unique matching queue shared between several OpenMP threads may be more efficient since an OpenMP thread can progress communications from another thread. Thus, MPI endpoints may prevent this effect from happening because every OpenMP thread has now its own matching queue. Finally, the Collaborative-Polling presented in chapter 4 would be an efficient solution to develop MPI endpoints while keeping the ability for an OpenMP thread to progress messages from other threads when idle.

Chapter 6

Conclusion and Future Work

The thesis is focused on improving the memory consumption and the performance scalability of network communications in HPC clusters of multiprocessors. It is concentrated on high-performance interconnects namely Infiniband and targets the MPI interface. With the purpose of showing the relevance of the contributions, the thesis relies on the multi-threading of MPI tasks inside an MPI runtime called MPC (MultiProcessor Computing).

Chapter 1 presented the design of nowadays supercomputers and the programming models commonly used to develop parallel applications. The chapter concluded on the six more important challenges an MPI runtime should overcome to efficiently use current HPC clusters and upcoming new platforms. Chapter 2 focused on interconnection networks for HPC. After introducing the overhead generated by the kernel-level communication libraries, we described some capabilities of recent interconnects for optimizing network performance and memory scalability. We then examined the high-speed Infiniband network and provided an in-depth analysis of the low-level `verbs` interface. More specifically on the Infiniband network, we finally emphasized that (1) the network is unreliable and only supports reliable communications through the connection-based protocols and, (2) the current programming interfaces do not support a fully independent progression of MPI messages.

6.1 Summary of the Research Contributions

The first contribution proposed new methods for enhancing scalability in memory of MPI runtimes while providing competitive results compared to the related work. Chapter 3 started with the demonstration that network endpoints and network buffers are by far the two predominant factors involved in the memory consumption.

To reduce the number of network endpoints required by connection-oriented high-speed networks, we proposed a signalization network that virtually and reliably exposes a fully-connected topology. Messages are routed through a torus topology and the design provides an extensible interface for developing user-level communication algorithms. Implemented inside MPC, the signalization network allows fast peer connections over Infiniband and drives control messages of the `eager` RDMA protocol while maintaining a constant amount of per-core memory.

The multi-threaded virtual rails (*vrails*) proposed a novel strategy for sharing network endpoints between MPI tasks in a shared-memory MPI runtime. The contribution focused on multi-rail configurations where several HCAs are symmetrically connected in pairs and one *vrail* is created per HCA. Two routing strategies were examined and the thesis established that the receiver-driven polling strategy (i.e., the *vrail* to use is determined according to the location of the receiver) provides the best results because it preserves the data locality of polling-related structures. Furthermore, we proposed a new strategy for `rendezvous` messages by designing a receiver-driven routing strategy in a multi-*vrail* context. On 128-core compute nodes

equipped with 4 HCAs, our approach reduces by 16,384 the number of network endpoints compared to the multi-rail strategy commonly embraced by mainstream MPI runtimes. Moreover, and on the same architecture, the contribution has demonstrated performance improvements using a single HCA. On mono-HCA configurations, we proposed to duplicate network structures across the compute node and to allocate one *vrail* per NUMA node. Although this strategy slightly increases the memory consumption, it significantly accelerates network communications. With 16 *vrails* per compute node and a micro-benchmark performing AllToAll communications, small and large messages respectively get speedups of 20 and 2 compared to one unique *vrail* shared by the whole compute node. This improvement is due to network resources that are locally accessed by the MPI tasks and because less contention is observed on synchronization points. More specifically, the BCS bandwidth is preserved and network communications passing this NUMA interconnect are less affected. We finally evaluated the contribution on a real-world astrophysical application where we show both a near perfect memory scaling and results comparable to the related-work up to 6,144 cores.

Auto-reshaping of `eager` RDMA buffers established new protocols for dynamically re-adjusting the memory attributed to network buffers during execution. This runtime optimization aims at reserving the precise amount of buffer memory corresponding to the communication volume of the MPI application being executed. We first proposed a protocol for enlarging memory regions of RDMA buffers for accelerating communications if a previous allocation has been under-evaluated. The experimental evaluation shows good results and a memory consumption reduced by a factor of 1.7 compared to the best configuration achieved by manually tweaking RDMA buffers. We then focused on an AMR application from CEA and demonstrated that the memory consumed (1) continuously increases during execution and (2) briefly grows during compute phases. As a result, previously allocated `eager` RDMA buffers could require the job to abort due to the shortage of memory following a memory increase. Thus, with the purpose of releasing memory, we finally introduced three different algorithms for reducing and disconnecting RDMA channels if the free memory becomes short.

The second contribution proposed a runtime optimization for improving overlap capabilities of MPI applications. Chapter 4 investigated several threaded message progressions and concluded that such progressions may generate an overhead. We then proposed a Collaborative-Polling approach where an idle MPI task can assist the message progression of any other busy MPI task (e.g., a task performing a computation loop). The experiments on the NAS Parallel Benchmarks have demonstrated a significant improvement of MPI *wait* functions due to a large number of messages stolen. Moreover, the Collaborative-Polling is able to give a speedup of 2 on real-world scientific applications and an in-depth study has proven the ability of the Collaborative-Polling to improve the asynchronous progression of control messages involved in the `rendezvous` protocol. Finally, this contribution led to one publication in a scientific conference [Did+12] and another in a journal [Did+13].

The third contribution evaluated the multi-threaded capability of MPI runtimes in hybrid MPI+threads context. Chapter 5 emphasized the need of an MPI runtime where several threads can concurrently and efficiently access the MPI runtime. We implemented our multi-threaded communication layer in MPC and compared its performance to Intel MPI. Indeed, both the runtimes combine both (1) the highest level of thread-safety and (2) an access to high-speed networks including Infini-

band. On hybrid MPI+PThreads latency benchmarks with 96 cores, MPC shows a speedup of 3.5 and 34.8 respectively on `eager` and `rendezvous` protocols compared to Intel MPI. We then evaluated both MPI runtimes on a seismic modeling application hybridized with OpenMP. Different hybrid versions of the code were examined and the domain decomposition method is the only version that fully parallelizes MPI communications. With one Infiniband HCA and 128-core compute nodes, MPC with the hybrid domain decomposition (`DD`) method outperforms the full-MPI version of the code with a speedup of 1.37. As regards Intel MPI, we highlighted a thread concurrency issue with the `DD` method and one MPI task per compute node. With one MPI task per level-2 NUMA node (4 MPI tasks per compute node), Intel MPI enhances the execution time of the `DD` version by a factor of 4.7 but the execution time is still 9% slower than MPC. In addition to reducing the execution time, the `DD` version with 1 MPI task per compute node saves 17.2% of memory compared to the full-MPI version. Overall, 19.9% and 28.5% of memory is saved respectively with MPC and Intel MPI due to the lower number of MPI runtimes instantiated. Finally, we presented some limitations of the hybrid mode due to the current MPI interface and we showed how the MPI endpoints could solve these limitations.

6.2 Scope of the Contributions

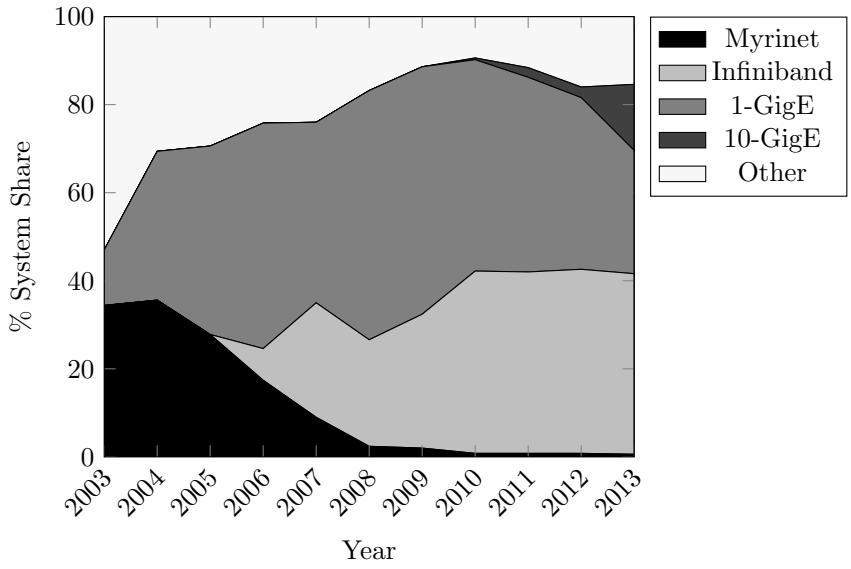


Figure 6.1 – Interconnect family system share over ten years, from 2003 to 2013 (from TOP500's website [Top])

Over a short period of ten years in High Performance Computing, interconnect's landscape has been considerably evolving. First and as depicted on figure 6.1, Myrinet was leading the market up to 2004 and began falling to give the way to Ethernet and Infiniband interconnects. Furthermore, although the graph seems to stabilize over the last three years, it is uncertain if Infiniband will remain a valuable network for future HPC platforms. As a consequence, it is important to propose contributions that will serve future supercomputers based on different hardware and software solutions.

First, the Collaborative-Polling relies on a limitation of actual Infiniband HCAs that provide only a few mechanisms to progress MPI messages in hardware. With

the offload of collective communications on the HCA, only Mellanox's ConnectX products actually propose a hardware progression. This interface is however very limited as it only supports calculation operations of scalar values [Tec11]. More generally, we observe that most current network interconnects do not propose hardware facilities for message progression and the table 2.1 previously presented clearly shows this trend. Indeed, offloading the progression engine of MPI requires to move many MPI-specific structures from the runtime to the network card (e.g., the MPI matching lists). This would first limit the network card to the very specific message matching of the MPI standard. Finally, it would definitively increase the complexity and so the market price of network controllers.

Second, the thesis targets the `verbs` interface because it supports a wide range of Infiniband hardware. Originally designed to unify Infiniband hardware under a unique and standardized interface, the `verbs` make now difficult the support of new hardware functionalities. The proprietary Mellanox VAPI/MXM and the Cisco usNIC interfaces are some example of this observations[1]. Moreover the `verbs` is the single low-level interface to access all Infiniband networks. Contrary to MX library for Myrinet and Tports for Quadrics, Infiniband does not facilitate the development of message passing implementations with specialized programming interface. In the light of these limitations, it is possible that the `verbs` interface will be reconsidered in the next years. We however consider that the contributions could be extended to a variety of communication libraries since most network concepts seen in the `verbs` also exist in other programming interfaces.

Then, the design of future exascale machines is not clearly defined as the time of writing. Some design attempts have been proposed by several exascale laboratories in 2010 (the US Department of Energy [Ash+10] and the Defense Advanced Research Projects Agency [Ama+09]) but since, no updates are available. Thus, it is uncertain if the next machine will express different levels of memory architecture. As an example, the Intel MIC architecture is one representative hardware of future supercomputers but the actual products released with this architecture only expose uniform memory accesses.

Although our contributions propose optimizations for NUMA architectures, they are especially based on the expectation that the intra-node concurrency of future architectures will increase. Moreover, we agree that the Bull's BCS architecture is not representative of mainstream supercomputers. Indeed, the BCS compute nodes were originally deployed on Curie for parallel applications developed using shared-memory programming models. We however think that the BCS is a good illustration of the large compute nodes that are expected for the upcoming exascale machines.

6.3 Future Work

In the following section we propose different research axes to extend the proposed contributions.

The section 4.5.3 concluded that the main CPU is still required for completing collective communications that involve complex operation. We plan to integrate the NBC library [HLR07] to MPC and we believe that the Collaborative-Polling could improve the non-blocking collective communications of MPI 3.0 without involving the overheads generated by a thread-based progression.

[1] Cisco modified the `verbs` interface to allow packets with arbitrary sizes like Ethernet

With the decrease in the amount of per-core memory and the increase of cores per compute node, we believe that MPI runtimes should be more dynamic and should adapt their memory consumption to the free memory in the compute nodes. More precisely, we plan to investigate disconnection of network endpoints (using the software solution presented in section 3.4.2.4 or using the Dynamically Connected Transport (DCT) protocol recently proposed by Mellanox and presented in section 2.2.5) for automatically resizing the communication buffers. The purpose here is to design a degraded mode where the runtime is able to run with the lowest memory footprint. In this mode, performance would not the priority and the runtime would do everything possible to prevent the job to be killed. As soon as the free memory increases, resources would be reallocated and the runtime performance would be finally restored.

Chapter 5 has demonstrated the efficiency of our multi-threaded communication layer combined with OpenMP. We plan to extend our previous evaluation to other shared-memory programming models. More specifically, we plan to look toward Intel Concurrent Collections (CnC [SBK13]). This recent programming model uses MPI in back-end for performing inter-node communications and threads for communicating data inside a node. We believe that our work could directly contribute to CnC since in the current implementation, multiple threads concurrently access the MPI interface. Additionally, we noted a genuine interest in MPI endpoints for supporting multi-rail configurations in an hybrid context [Din+13]. As a future work, we intend to integrate the MPI endpoints in MPC. Indeed, MPC would be an ideal candidate for integrating the MPI endpoints because the runtime efficiently supports multi-threaded MPI ranks and provides a unified scheduler for MPI and OpenMP.

Finally, while this thesis has targeted the MPI interface, other programming models like Chapel, Fortress, X10, UPC or Charm++ [KK93] also require messaging between nodes. We plan to investigate how our contributions would support these programming models.

Bibliography

[AA04] G. Amerson and a. Apon. “Implementation and design analysis of a network messaging module using virtual interface architecture”. In: *International Conference on Cluster Computing* (2004) (see p. 79).

[Ama+09] Saman Amarasinghe et al. “Exascale software study: Software challenges in extreme scale systems”. In: *DARPA IPTO, Air Force Research Labs, Tech. Rep* (2009) (see p. VIII, 5, 14, 35, 120).

[Ash+10] Steve Ashby et al. “The opportunities and challenges of exascale computing—summary report of the advanced scientific computing advisory committee (ASCAC) subcommittee. US Department of Energy Office of Science”. In: *US Department of Energy Office of Science* (2010) (see p. VIII, 5, 14, 35, 120).

[Atc+11] Scott Atchley et al. “The Common Communication Interface (CCI)”. In: *Hot Interconnects*. 2011, pp. 51–60 (see p. 25).

[Aug+12] Cédric Augonnet et al. *StarPU-MPI: Task Programming over Clusters of Machines Enhanced with Accelerators*. conference proceeding. 2012 (see p. 95).

[Aum+07] Olivier Aumage et al. “NEW MADELEINE: a Fast Communication Scheduling Engine for High Performance Networks”. In: *International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2007, pp. 1–8 (see p. 36, 41, 51, 67).

[Bai+95] David Bailey et al. *The NAS Parallel Benchmarks 2.0*. 1995 (see p. 69, 84).

[Bal+09] Pavan Balaji et al. “MPI on a Million Processors”. In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface (PVM/MPI)*. Ed. by Matti Ropo, Jan Westerholm, and Jack Dongarra. Sept. 2009 (see p. 36, 92).

[Bal+10] Pavan Balaji et al. “PMI: A Scalable Parallel Process-Management Interface for Extreme-Scale Systems”. In: *Recent Advances in the Message Passing Interface (EuroMPI)*. Ed. by Rainer Keller et al. Vol. 6305. Lecture Notes in Computer Science (LNCS). 2010, pp. 31–41 (see p. 45).

[Bar+03] Paul Barham et al. “Xen and the art of virtualization”. In: *ACM SIGOPS Operating Systems Review* 37.5 (2003), pp. 164–177 (see p. 67).

[BEA09] J. Mark Bull, James P. Enright, and Nadia Ameer. “A Microbenchmark Suite for Mixed-Mode OpenMP/MPI”. In: *Evolving OpenMP in an Age of Extreme Parallelism (IWOMP)*. Ed. by Matthias S. Muller, Bronis R. de Supinski, and Barbara M. Chapman. Vol. 5568. Lecture Notes in Computer Science (LNCS). June 2009, pp. 118–131 (see p. 96).

[Bee+03] Jon Beecroft et al. “QsNet-II: An Interconnect for Supercomputing Applications”. In: *The Proceedings of Hot Chips*. 2003 (see p. 23).

[Bel+06] Christian Bell et al. “Optimizing bandwidth limited problems using one-sided communication and overlap”. In: *International Parallel and Distributed Processing Symposium (IPDPS)*. 2006 (see p. 78).

[Ber+05] M. J. Berger et al. “Performance of a New CFD Flow Solver using a Hybrid Programming Paradigm”. In: *Journal of Parallel and Distributed Computing* 65.4 (Apr. 2005) (see p. 94, 96, 105).

[Blu+96] R. Blumofe et al. “Cilk: An Efficient Multithreaded Runtime System”. In: *Journal of Parallel and Distributed Computing* 37.1 (1996), pp. 55–69 (see p. 9).

[BMG06] Darius Buntinas, Guillaume Mercier, and William Gropp. “Design and evaluation of Nemesis, a scalable, low-latency, message-passing communication subsystem”. In: *Cluster Computing and the Grid (CCGRID)*. Vol. 1. IEEE. 2006, 10–pp (see p. 13, 15, 36, 93).

[Bod+95] Nanette J. Boden et al. “Myrinet: A Gigabit-per-Second, Local Area Network”. In: *IEEE Micro* 15.1 (Feb. 1995), pp. 29–36 (see p. 23, 28).

[Bon02] Dan Bonachea. *GASNet Specification, v1.1*. Tech. rep. 2002 (see p. 25).

[Bos+] George Bosilca et al. “MPICH-V: toward a scalable fault tolerant MPI for volatile nodes”. In: pp. 1–18 (see p. 11).

[BP03] Darius Buntinas and Dhabaleswar K Panda. “NIC-based reduction in Myrinet clusters: is it beneficial?” In: *In SAN-02 Workshop (in conjunction with HPCA)*. 2003 (see p. 22).

[BP11] Ron Brightwell and Kevin Pedretti. “An Intra-Node Implementation of OpenSHMEM Using Virtual Address Space Mapping”. In: *Fifth Partitioned Global Address Space Conference*. 2011 (see p. 84).

[Bri+] R. Brightwell et al. “Portals 3.0: Protocol Building Blocks for Low Overhead Communication”. In: *International Parallel And Distributing Processing Symposium (IPDPS)*. Ed. by Bob Werner, pp. 164–164 (see p. 25).

[Bri+05] Ron Brightwell et al. “A Hardware Acceleration Unit for MPI Queue Processing”. In: *International Parallel and Distributed Processing Symposium*. 2005 (see p. 22).

[Bro+10a] François Broquedis et al. “ForestGOMP: An Efficient OpenMP Environment for NUMA Architectures”. In: *International Journal of Parallel Programming* 38.5-6 (2010), pp. 418–439 (see p. 96).

[Bro+10b] François Broquedis et al. “hwloc: A generic framework for managing hardware affinities in hpc applications”. In: *Parallel, Distributed and Network-Based Processing (PDP)*. IEEE. 2010, pp. 180–186 (see p. 56).

[BRU05] Ron Brightwell, Rolf Riesen, and Keith D. Underwood. “Analyzing the Impact of Overlap, Offload, and Independent Progress for Message Passing Interface Applications”. In: *IJHPCA* (2005) (see p. 78).

[Bul] Bull. *Bullx MPI*. http://bull.com (see p. 11).

[CE00] Franck Cappello and Daniel Etiemble. “MPI versus MPI+OpenMP on IBM SP for the NAS Benchmarks”. In: *Proceedings of Supercomputing*. LRI. Nov. 2000 (see p. 96).

[Cha+13] Sanjay Chatterjee et al. “Integrating Asynchronous Task Parallelism with MPI”. In: *Department of Computer Science, Rice University, Technical Report TR12-07* (2013) (see p. 95).

[Che+12] Dong Chen et al. "Looking Under the Hood of the IBM Blue Gene/Q Network". In: *SC'12 CD-ROM: Conference on High Performance Computing Networking, Storage and Analysis*. Salt Lake City, UT, USA: ACM SIGARCH/IEEE Computer Society, Nov. 2012 (see p. 22).

[CIY05] Wei-Yu Chen, Costin Iancu, and Katherine A. Yelick. "Communication Optimizations for Fine-Grained UPC Applications". In: *IEEE PACT*. IEEE Computer Society, 2005, pp. 267–278 (see p. 9).

[Col+03] Salvador Coll et al. "Using multirail networks in high-performance clusters". In: *Concurrency and Computation: Practice and Experience* 15.7-8 (2003), pp. 625–651 (see p. 67).

[CPJ11] Patrick Carribault, Marc Pérache, and Hervé Jourdren. "Thread-Local Storage Extension to Support Thread-Based MPI/OpenMP Applications". In: *7th International Workshop on OpenMP (IWOMP)*. Ed. by Barbara M. Chapman et al. Lecture Notes in Computer Science. 2011, pp. 80–93 (see p. 12, 80).

[CPP01] Barbara M. Chapman, Amit Patil, and Achal Prabhakar. "Performance Oriented Programming for NUMA Architectures". In: *OpenMP Shared Memory Parallel Programming, International Workshop on OpenMP Applications and Tools, WOMPAT 2001*. Ed. by Rudolf Eigenmann and Michael Voss. Vol. 2104. Lecture Notes in Computer Science (LNCS). West Lafayette, IN, USA: Springer-Verlag (New York), July 2001, pp. 137–154 (see p. 94).

[CRS09] Jie Cai, Alistair P. Rendell, and Peter E. Strazdins. "Non-threaded and Threaded Approaches to MultiRail Communication with uDAPL". In: *NPC*. IEEE Computer Society, 2009, pp. 233–239 (see p. 50).

[Dem97] E. Demaine. "A Threads-Only MPI Implementation for the Development of Parallel Programming". In: *Proceedings of the 11th International Symposium on High Performance Computing Systems*. 1997 (see p. 11).

[Den11] Alexandre Denis. *A High-Performance Superpipeline Protocol for InfiniBand*. proceeding with peer review. Aug. 2011 (see p. 41).

[Did+12] Sylvain Didelot et al. "Improving MPI Communication Overlap with Collaborative Polling". In: *Recent Advances in the Message Passing Interface (EuroMPI)*. Ed. by Jesper Larsson Träff, Siegfried Benkner, and Jack J. Dongarra. Vol. 7490. Lecture Notes in Computer Science. Springer, 2012, pp. 37–46 (see p. 18, 118).

[Did+13] Sylvain Didelot et al. "Improving MPI communication overlap with collaborative polling". In: *Computing* (2013), pp. 1–16. DOI: 10.1007/s00607-013-0327-z. URL: http://dx.doi.org/10.1007/s00607-013-0327-z (see p. 18, 118).

[Din+10] James Dinan et al. "Hybrid parallel programming with MPI and unified parallel C". In: *Proceedings of the 7th ACM international conference on Computing frontiers*. ACM. 2010, pp. 177–186 (see p. 16).

[Din+13] James Dinan et al. "Enabling MPI Interoperability Through Flexible Communication Endpoints". In: (2013) (see p. 96, 115, 116, 121).

[DK04] Nikolaos Drosinos and Nectarios Koziris. “Performance Comparison of Pure MPI vs Hybrid MPI-OpenMP Parallelization Models on SMP Clusters”. In: *International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, 2004 (see p. 96).

[Don88] Jack J Dongarra. “The LINPACK benchmark: An explanation”. In: *Supercomputing*. Springer. 1988, pp. 456–474 (see p. 6, 7).

[EgCD03] Tarek A. El-ghazawi, William W. Carlson, and Jesse M. Draper. *UPC Language Specification v1.1.1*. Oct. 2003 (see p. 9).

[EGS07] Tarek El-Ghazawi and Vivek Sarkar. “Programming using the Partitioned Global Address Space (PGAS) Model”. In: *SC'07 USB Key*. Reno, NV: ACM/IEEE, Nov. 2007 (see p. 9).

[Eic+92] Thorsten von Eicken et al. “Active Messages: a Mechanism for Integrated Communication and Computation”. In: *International Symposium on Computer Architecture*. 1992 (see p. 25).

[FD00] Graham E. Fagg and Jack J. Dongarra. *FT-MPI: Fault Tolerant MPI, supporting dynamic applications in a dynamic world*. Oct. 2000 (see p. 11).

[Foo+03] Annie P. Foong et al. “TCP performance re-visited”. In: *ISPASS*. IEEE Computer Society, 2003, pp. 70–79 (see p. 19).

[Fre+04] Felix Freitag et al. “Predicting MPI Buffer Addresses”. In: *ICCS*. Ed. by Marian Bubak et al. Vol. 3036. Lecture Notes in Computer Science (LNCS). June 2004, pp. 10–17 (see p. 40).

[Fri+07] Andrew Friedley et al. “Scalable high performance message passing over InfiniBand for Open MPI”. In: *Proceedings of 2007 KiCC Workshop, RWTH Aachen*. 2007 (see p. 37, 50).

[Fri+13] A. Friedley et al. “Hybrid MPI: Efficient Message Passing for Multi-core Systems”. In: IEEE/ACM International Conference on High Performance Computing, Networking, Storage and Analysis (SC13). 2013 (see p. 11, 15, 93).

[Gab+04] Edgar Gabriel et al. “Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation”. In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface(PVM/MPI)*. Ed. by Dieter Kranzlmüller, Péter Kacsuk, and Jack Dongarra. Vol. 3241. Lecture Notes in Computer Science. 2004, pp. 97–104 (see p. 40).

[GM12] Brice Goglin and Stéphanie Moreaud. “KNEM: A generic and scalable kernel-assisted intra-node MPI communication framework”. In: *Journal of Parallel and Distributed Computing* (2012) (see p. 11, 13, 93).

[Goo+11] David Goodell et al. “Scalable Memory Use in MPI: A Case Study with MPICH2”. In: *Recent Advances in the Message Passing Interface (EuroMPI)*. Lecture Notes in Computer Science. 2011 (see p. 36).

[Gra+10] Richard L Graham et al. “Overlapping computation and communication: Barrier algorithms and ConnectX-2 CORE-Direct capabilities”. In: *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*. IEEE. 2010, pp. 1–8 (see p. 47).

[Gro+96] William Gropp et al. "A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard". In: *Parallel Computing* 22.6 (1996), pp. 789–828 (see p. 11).

[GT06] William D. Gropp and Rajeev Thakur. "Issues in Developing a Thread-Safe MPI Implementation". In: *PVM/MPI*. 2006, pp. 12–21 (see p. 96, 99).

[Gup+03] Rinku Gupta et al. "Efficient collective operations using remote memory operations on VIA-based clusters". In: *Parallel and Distributed Processing Symposium*. IEEE. 2003, 9–pp (see p. 22).

[GWS06] Richard L. Graham, Timothy S. Woodall, and Jeffrey M. Squyres. "OpenMPI: A Flexible High Performance MPI". In: *Parallel Processing and Applied Mathematics*. 2006 (see p. 11).

[Hag+] Georg Hager et al. "Prospects for Truly Asynchronous Communication with Pure MPI and Hybrid MPI/OpenMP on Current Supercomputing Platforms". In: () (see p. 95).

[Hil+03] Jeff Hilland et al. *RDMA Protocol Verbs Specification*. http://www.rdmaconsortium.org/home/draft-hilland-iwarp-verbs-v1.0-RDMAC.pdf. Apr. 2003 (see p. 23).

[HJR09] Georg Hager, Gabriele Jost, and Rolf Rabenseifner. "Communication Characteristics and Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-core SMP Nodes". In: *Proceedings of Cray User Group*. 2009 (see p. 78).

[HL08] Torsten Hoefler and Andrew Lumsdaine. "Message progression in parallel computing – to thread or not to thread?" In: *International Conference on Cluster Computing*. 2008 (see p. 80).

[HLK04] Chao Huang, Orion Lawlor, and L. V. Kalé. "Adaptive MPI". In: *Languages and Compilers for Parallel Compution (LCPC)*. 2004 (see p. 11).

[HLR07] Torsten Hoefler, Andrew Lumsdaine, and Wolfgang Rehm. "Implementation and Performance Analysis of Non-blocking Collective Operations for MPI". In: *SC*. Nov. 2007 (see p. 90, 120).

[Hoe+10] Torsten Hoefler et al. "Efficient MPI support for advanced hybrid programming models". In: *Recent Advances in the Message Passing Interface*. Springer, 2010, pp. 50–61 (see p. 96).

[HS11a] T. Hoefler and M. Snir. "Generic Topology Mapping Strategies for Large-scale Parallel Architectures". In: *Proceedings of the 2011 ACM International Conference on Supercomputing (ICS)*. June 2011, pp. 75–85 (see p. 48).

[HS11b] Torsten Hoefler and Marc Snir. "Writing parallel libraries with MPI-common practice, issues, and extensions". In: *Recent Advances in the Message Passing Interface*. Springer, 2011, pp. 345–355 (see p. 36).

[Hua+06] Wei Huang et al. "Design of High Performance MVAPICH2: MPI2 over InfiniBand". In: *CCGRID*. 2006, pp. 43–48 (see p. 11, 40, 41).

[IB99] J. B. White Iii and S. W. Bova. *Where's the Overlap? - An Analysis of Popular MPI Implementations*. Tech. rep. Aug. 1999 (see p. 78).

[IBM] IBM. *IBM Platform MPI*. http://www-03.ibm.com/systems/technicalcomputing/platformcomputing/products/mpi/ (see p. 11).

[Inc] Myricom Inc. *MPICH-MX*. https://www.myricom.com/support/downloads/mx/mpich-mx.html (see p. 11).

[Inc10] Cray Inc. *The Gemini Network, v1.1*. 2010 (see p. 22).

[Inc11] Cray Inc. *Using the GNI and DMAPP APIs*. 2011 (see p. 22).

[Inta] Intel. *Intel Math Kernel Library (Intel MKL)*. http://software.intel.com/en-us/intel-mkl (see p. 9).

[Intb] Intel. *Intel MPI benchmarks (IMB)*. http://software.intel.com/en-us/articles/intel-mpi-benchmarks (see p. 40).

[Intc] Intel. *Intel MPI Library*. http://software.intel.com/en-us/intel-mpi-library (see p. 11).

[Intd] Intel. *Intel Threading Building Blocks (Intel TBB)*. http://software.intel.com/en-us/intel-tbb?wapkw=tbb (see p. 9, 16).

[Inte] Intel. *Intel Xeon Phi Coprocessor 5110P*. http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html (see p. 14).

[JG04] M Jette and M Grondona. *SLURM: Simple Linux Utility for Resource Management*. June 2004 (see p. 32).

[Jin+07] Hyun-Wook Jin et al. "Lightweight kernel-level primitives for high-performance MPI intra-node communication over multi-core systems". In: *Cluster Computing, 2007 IEEE International Conference on*. IEEE. 2007, pp. 446–451 (see p. 11, 93).

[Jin+11] Haoqiang Jin et al. "High performance computing using MPI and OpenMP on multi-core parallel systems". In: *Parallel Computing* 37.9 (Sept. 2011), pp. 562–575 (see p. 96, 104).

[JM10] Emmanuel Jeannot and Guillaume Mercier. *Near-Optimal Placement of MPI processes on Hierarchical NUMA Architectures*. proceeding with peer review. Aug. 2010 (see p. 37).

[Jos+10] Jithin Jose et al. "Unifying UPC and MPI runtimes: experience with MVAPICH". In: *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*. ACM. 2010, p. 5 (see p. 16).

[Jou05] Hervé Jourdren. "HERA: A Hydrodynamic AMR Platform for Multi-Physics Simulations". In: *Adaptive Mesh Refinement - Theory and Application, LNCSE*. 2005 (see p. 43, 72).

[Jow+09] M. Jowkar et al. *D6.4 Report on approaches to Petascaling*. PRACE, Technical Report. 2009 (see p. VIII, 5, 91, 92).

[Kan+11] Krishna Kandalla et al. "High-performance and scalable non-blocking all-to-all with collective offload on InfiniBand clusters: a study with parallel 3D FFT". In: *Computer Science-Research and Development* 26.3-4 (2011), pp. 237–246 (see p. 22, 47).

[Kan+12] Krishna Chaitanya Kandalla et al. "Designing Non-blocking Allreduce with Collective Offload on InfiniBand Clusters: A Case Study with Conjugate Gradient Solvers". In: *International Parallel and Distributed Processing Symposium (IPDPS)*. Shanghai, China: IEEE Computer Society, May 2012, pp. 1156–1167 (see p. 22, 90).

[KC03] Géraud Krawezik and Frank Cappello. "Performance comparison of MPI and three OpenMP programming styles on shared memory multiprocessors". In: *Proceedings of the 15th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM SIGACT, ACM SIGARCH. 2003, pp. 118–127 (see p. 94, 96, 105).

[Key+00] D Keyes et al. "A Parallel Computing Framework for Dynamic Power Balancing in Adaptive Mesh Refinement Applications". In: *Parallel Computational Fluid Dynamics' 99: Towards Teraflops, Optimization and Novel Formulations* (2000), p. 249 (see p. 92).

[KHS12] Fredrik Kjolstad, Torsten Hoefler, and Marc Snir. "Automatic datatype generation and optimization". In: *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*. Ed. by J. Ramanujam and P. Sadayappan. 2012, pp. 327–328 (see p. 36).

[Kim+10] Keiji Kimura et al. "OSCAR API for real-time low-power multicores and its performance on multicores and SMP servers". In: *Languages and Compilers for Parallel Computing*. Springer, 2010, pp. 188–202 (see p. 8).

[Kiv+07] Avi Kivity et al. "kvm: the Linux virtual machine monitor". In: *Proceedings of the Linux Symposium*. Vol. 1. 2007, pp. 225–230 (see p. 67).

[KJP07] Matthew J. Koop, Terry Jones, and Dhabaleswar K. Panda. "Reducing Connection Memory Requirements of MPI for InfiniBand Clusters: A Message Coalescing Approach". In: *CCGRID*. 2007, pp. 495–504 (see p. 41).

[KJP08] Matthew J. Koop, Terry Jones, and Dhabaleswar K. Panda. "MVAPICH-Aptus: Scalable high-performance multi-transport MPI over InfiniBand". In: *International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2008, pp. 1–12 (see p. 38).

[KK93] L. V. Kale and Sanjeev Krishnan. "CHARM++ : A Portable Concurrent Object-Oriented System Based on C++". In: *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA)*. Ed. by Andreas Paepcke. ACM Press, Sept. 1993, pp. 91–108 (see p. 61, 121).

[KKB14] Gurkirat Kaur, Manoj Kumar, and Manju Bala. "Comparing Ethernet & Soft RoCE over 1 Gigabit Ethernet". In: *International Journal of Computer Science and Information Technologies (IJCSIT)* 5.1 (Feb. 2014), pp. 323–327 (see p. 23).

[KL98] JunSeong Kim and David J. Lilja. "Characterization of Communication Patterns in Message-Passing Parallel Scientific Application Programs". In: *Lecture Notes in Computer Science* 1362 (1998), 202–?? (See p. 39).

[Koo+07] Matthew J. Koop et al. "High performance MPI design using unreliable datagram for ultra-scale InfiniBand clusters". In: *Proceedings of the 21th Annual International Conference on Supercomputing (ICS)*. Ed. by Burton J. Smith. June 2007, pp. 180–189 (see p. 37).

[KSP07] Matthew J. Koop, Sayantan Sur, and Dhabaleswar K. Panda. "Zero-copy protocol for MPI using infiniband unreliable datagram". In: *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER)*. Sept. 2007, pp. 179–186 (see p. 37).

[KSP08] Matthew J. Koop, Jaidev K. Sridhar, and Dhabaleswar K. Panda. "Scalable MPI design over InfiniBand using eXtended Reliable Connection". In: *CLUSTER*. 2008, pp. 203–212 (see p. 31, 37).

[KSP09] MJ Koop, AP Sampat, and D. K. Panda. *Veloblock: Efficient and Scalable RDMA Fast Path for InfiniBand*. Tech. rep. 2009 (see p. 42, 68).

[Kum+08] Rahul Kumar et al. "Lock-Free Asynchronous Rendezvous Design for MPI Point-to-Point Communication". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface (PVM/MPI)*. 2008 (see p. 31, 79).

[Kum+12] Sameer Kumar et al. "PAMI: A Parallel Active Message Interface for the Blue Gene/Q Supercomputer". In: *International Parallel and Distributed Processing Symposium (IPDPS)*. Shanghai, China: IEEE Computer Society, May 2012, pp. 763–773 (see p. 23).

[KW12] Humaira Kamal and Alan Wagner. "Added Concurrency to Improve MPI Performance on Multicore". In: *ICPP*. 2012, pp. 229–238 (see p. 11, 78).

[Laba] Sandia National Laboratories. *Portals 4 Reference Implementation*. https://code.google.com/p/portals4/ (see p. 26).

[Labb] Argonne National Laboratory. *MPICH2*. http://www.mcs.anl.gov/mpi/mpich2 (see p. 11).

[Labc] Argonne National Laboratory. *Using the Hydra Process Manager*. URL: http://wiki.mpich.org/mpich/index.php/Using_the_Hydra_Process_Manager (see p. 45).

[Lei85] Charles E Leiserson. "Fat-trees: universal networks for hardware-efficient supercomputing". In: *Computers, IEEE Transactions on* 100.10 (1985), pp. 892–901 (see p. 7, 31).

[Len+03] James Lentini et al. "Implementation and Analysis of the User Direct Access Programming Library". In: *2nd Workshop on Novel Uses of System Area Networks, SAN*. Vol. 2. 2003 (see p. 26).

[Liu+03] Jiuxing Liu et al. *Design and Implementation of MPICH2 over InfiniBand with RDMA Support*. Comment: 12 pages, 17 figures. Oct. 2003 (see p. 40).

[Luo+11] Miao Luo et al. "Multi-threaded UPC runtime with network endpoints: Design alternatives and evaluation on multi-core architectures". In: *HiPC*. IEEE, 2011, pp. 1–10 (see p. 49).

[LVP04] Jiuxing Liu, Abhinav Vishnu, and Dhabaleswar K. Panda. "Building Multirail InfiniBand Clusters: MPI-Level Design and Performance Evaluation". In: *SC*. Nov. 2004 (see p. 50–52, 67, 114).

[LWP04] Jiuxing Liu, Jiesheng Wu, and Dhabaleswar K. Panda. "High Performance RDMA-Based MPI Implementation over InfiniBand". In: *International Journal of Parallel Programming (IJPP)* 32.3 (June 2004), pp. 167–198 (see p. 68).

[Mar+10] Vladimir Marjanovic et al. "Effective communication and computation overlap with hybrid MPI/SMPSs". In: *ACM SIGPLAN Notices* 45.5 (May 2010), pp. 337–338 (see p. 95).

[MG+07] Stéphanie Moreaud, Brice Goglin, et al. "Impact of NUMA effects on high-speed networking with multi-opteron machines". In: *PDCS*. 2007 (see p. 51).

[MGN10] Stéphanie Moreaud, Brice Goglin, and Raymond Namyst. "Adaptive MPI Multirail Tuning for Non-Uniform Input/Output Access". In: (2010) (see p. 51, 67, 114).

[Mie+06] Frank Mietke et al. "Analysis of the Memory Registration Process in the Mellanox InfiniBand Software Stack". In: *Euro-Par*. Ed. by Wolfgang E. Nagel, Wolfgang V. Walter, and Wolfgang Lehner. Vol. 4128. Lecture Notes in Computer Science (LNCS). Aug. 2006, pp. 124–133 (see p. 30).

[MJ11] Guillaume Mercier and Emmanuel Jeannot. "Improving MPI Applications Performance on Multicore Clusters with Rank Reordering". In: *Recent Advances in the Message Passing Interface (EuroMPI)*. Ed. by Yiannis Cotronis et al. Vol. 6960. Lecture Notes in Computer Science. 2011, pp. 39–49 (see p. 37).

[Moo+65] Gordon E Moore et al. *Cramming more components onto integrated circuits*. 1965 (see p. 6).

[Mun+09] Aaftab Munshi et al. "The opencl specification". In: *Khronos OpenCL Working Group* 1 (2009), pp. l1–15 (see p. 16).

[Ngu+12] Tan Nguyen et al. "Bamboo – Translating MPI Applications to a Latency-Tolerant, Data-Driven Form". In: *SC'12 CD-ROM: Conference on High Performance Computing Networking, Storage and Analysis*. Salt Lake City, UT, USA: ACM SIGARCH/IEEE Computer Society, Nov. 2012 (see p. 78).

[NR98] R. Numrich and J. Reid. "Co-Array Fortran for Parallel Programming". In: *ACM Fortran Forum* 17.2 (Aug. 1998), pp. 1–31 (see p. 9).

[Nvi08] CUDA Nvidia. *Programming guide*. 2008 (see p. 16).

[Ort+08] Francisco Ortigosa et al. "Evaluation of 3D RTM on HPC platforms". In: *2008 SEG Annual Meeting*. 2008 (see p. 100).

[PCJ09] Marc Pérache, Patrick Carribault, and Hervé Jourdren. "MPC-MPI: An MPI Implementation Reducing the Overall Memory Consumption". In: *PVM/MPI*. 2009 (see p. VIII, 5, 11, 91).

[Pen+10] Brad Penoff et al. "Employing transport layer multi-railing in cluster networks". In: *J. Parallel Distrib. Comput* 70.3 (2010), pp. 259–269 (see p. 67).

[Pet+01] Fabrizio Petrini et al. “The Quadrics Network (QsNet): High-Performance Clustering Technology”. In: 2001 (see p. 28).

[Peñ+13] Antonio J. Peña et al. “Analysis of topology-dependent MPI performance on Gemini networks”. In: *Recent Advances in the Message Passing Interface (EuroMPI)*. Ed. by Jack Dongarra, Javier García Blas, and Jesús Carretero. ACM, 2013, pp. 61–66 (see p. 48).

[PG07] Jelena Pjesivac-Grbović. “Towards Automatic and Adaptive Optimizations of MPI Collective Operations”. PhD thesis. The University of Tennessee, Knoxville, Dec. 2007 (see p. 43).

[PGB11] Howard Pritchard, Igor Gorodetsky, and Darius Buntinas. “A uGNI-Based MPICH2 Nemesis Network Module for the Cray XE”. In: *Recent Advances in the Message Passing Interface (EuroMPI)*. Ed. by Yiannis Cotronis et al. Vol. 6960. Lecture Notes in Computer Science. Springer, 2011, pp. 110–119 (see p. 22).

[PJN08] Marc Pérache, Hervé Jourdren, and Raymond Namyst. “MPC: A Unified Parallel Runtime for Clusters of NUMA Machines”. In: *Euro-Par*. 2008 (see p. 116).

[PS98] Boris V. Protopopov and Anthony Skjellum. *A multi-threaded Message Passing Interface (MPI) architecture: performance and program issues*. Sept. 1998 (see p. 95).

[QA08] Ying Qian and Ahmad Afsahi. “Efficient shared memory and RDMA based collectives on multi-rail QsNet^{II} SMP clusters”. In: *Cluster Computing* 11.4 (2008), pp. 341–354 (see p. 67).

[RGM11] Juan A. Rico-Gallego and Juan Carlos Díaz Martín. “Performance Evaluation of Thread-Based MPI in Shared Memory”. In: *Recent Advances in the Message Passing Interface (EuroMPI)*. 2011 (see p. 11).

[RHJ09] Rolf Rabenseifner, Georg Hager, and Gabriele Jost. “Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes”. In: *Parallel, Distributed and Network-based Processing*. IEEE. 2009, pp. 427–436 (see p. VIII, 5, 93, 96).

[Ric98] Olivier Richard. *Intra node parallelization of MPI programs with OpenMP*. Jan. 1998 (see p. 94).

[RW03] Rolf Rabenseifner and Gerhard Wellein. “Communication and Optimization Aspects of Parallel Programming Models on Hybrid Architectures”. In: *IJHPCA* 17.1 (2003), pp. 49–62 (see p. 95, 96).

[SB01] Lorna Smith and Mark Bull. “Development of mixed mode MPI / OpenMP applications”. In: *Scientific Programming* 9.2-3 (2001). EPCC, pp. 83–98 (see p. 96).

[SBK13] Frank Schlimbach, James C Brodman, and Kath Knobe. “Concurrent Collections on Distributed Memory Theory Put Into Practice”. In: *Parallel, Distributed and Network-Based Processing (PDP)*. IEEE. 2013, pp. 225–232 (see p. 95, 121).

[SGY10] M. Small, Z. Gu, and X. Yuan. “Near-optimal Rendezvous protocols for RDMA-enabled clusters”. In: *International Conference on Parallel Processing (ICPP)*. 2010 (see p. 36).

[Shi+06] Galen M. Shipman et al. "Infiniband scalability in Open MPI". In: *International Parallel and Distributed Processing Symposium (IPDPS)*. 2006 (see p. 31).

[Shi+07] Galen M. Shipman et al. "Investigations on InfiniBand: Efficient Network Buffer Utilization at Scale". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface (PVM/MPI)*. Ed. by Franck Cappello, Thomas Herault, and Jack Dongarra. Vol. 4757. Lecture Notes in Computer Science (LNCS). Oct. 2007, pp. 178–186 (see p. 41, 47).

[Shi+08] Galen M. Shipman et al. "X-SRQ– Improving Scalability and Performance of Multi-core InfiniBand Clusters". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface (PVM/MPI)*. Ed. by Alexey L. Lastovetsky, M. Tahar Kechadi, and Jack Dongarra. Vol. 5205. Lecture Notes in Computer Science (LNCS). Sept. 2008, pp. 33–42 (see p. 31, 37).

[Spr05] Volker Springel. "The cosmological simulation code gadget-2". In: *Monthly Notices of the Royal Astronomical Society* 364 (2005) (see p. 84).

[SPT98] Edward S Smyth, Jonathan S Parker, and Ken T Taylor. "Numerical integration of the time-dependent Schrödinger equation for laser-driven helium". In: *Computer physics communications* 114.1 (1998), pp. 1–14 (see p. 91).

[Sto+08] James M Stone et al. "Athena: a new code for astrophysical MHD". In: *The Astrophysical Journal Supplement Series* 178.1 (2008), p. 137 (see p. 58, 64).

[Sub+11] Vladimir Subotic et al. "The Impact of Application's Micro-Imbalance on the Communication-Computation Overlap". In: *Parallel, Distributed and Network-based Processing (PDP)*. 2011 (see p. 78, 80).

[Sun90] Vaidy S. Sunderam. "PVM: A framework for parallel distributed computing". In: *Concurrency: practice and experience* 2.4 (1990), pp. 315–339 (see p. 9).

[Sur+05] Sayantan Sur et al. "Can memory-less network adapters benefit next-generation infiniband systems?" In: *High Performance Interconnects*. IEEE. 2005, pp. 45–50 (see p. 49).

[Sur+06a] Sayantan Sur et al. "RDMA Read Based Rendezvous Protocol for MPI over InfiniBand: Design Alternatives and Benefits". In: *Alternatives* (2006) (see p. 31, 79).

[Sur+06b] Sayantan Sur et al. "Shared receive queue based scalable MPI design for InfiniBand clusters". In: *International Parallel and Distributed Processing Symposium (IPDPS)*. 2006 (see p. 31).

[SVP13] Hari Subramoni, Jerome Vienne, and Dhabaleswar K DK Panda. "A scalable infiniband network topology-aware performance analysis tool for MPI". In: *Euro-Par 2012: Parallel Processing Workshops*. Springer. 2013, pp. 439–450 (see p. 48).

[TCP12] Marc Tchiboukdjian, Patrick Carribault, and Marc Pérache. "Hierarchical Local Storage: Exploiting Flexible User-Data Sharing Between MPI Tasks". In: *International Parallel and Distributed Processing Symposium (IPDPS)*. 2012, pp. 366–377 (see p. 35, 91).

[TD09] François Trahay and Alexandre Denis. "A scalable and generic task scheduling system for communication libraries". In: *International Conference on Cluster Computing*. 2009 (see p. 80).

[Tec11] Mellanox Technologies. *Collectives Offload - API (revision 1.3)*. 2011 (see p. 26, 90, 120).

[Tec13a] Mellanox Technologies. *Connect-IB: Architecture for Scalable High Performance Computing*. 2013 (see p. 26, 31).

[Tec13b] Mellanox Technologies. *RDMA Aware Networks Programming User Manual*. 2013 (see p. 26).

[Tec14] Mellanox Technologies. *MellanoX Messaging Library User Manual*. 2014 (see p. 26).

[Tez+98] Hiroshi Tezuka et al. "Pin-Down Cache: A Virtual Memory Management Technique for Zero-Copy Communication". In: *IPPS/SPDP*. 1998, pp. 308–314 (see p. 30, 83).

[TG07] Rajeev Thakur and William Gropp. "Test Suite for Evaluating Performance of MPI Implementations That Support MPI_THREAD_MULTIPLE". In: *PVM/MPI*. 2007, pp. 46–55 (see p. VIII, 5, 16, 78, 95–97).

[Tha+10] Rajeev Thakur et al. "MPI at Exascale". In: *Procceedings of SciDAC* 2 (2010) (see p. 14, 35).

[Top] *Top500 Supercomputer Sites*. http://top500.org. June 2013 (see p. 5, 6, 13, 23, 119).

[Tra03] Jesper Larsson Traff. "SMP-aware message passing programming". In: *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*. IEEE. 2003, 10–pp (see p. 101).

[Tre89] Lloyd N. Trefethen. *SCPACK User's Guide*. Numerical Analysis Report 89-2. (An earlier edition appeared as an ICASE internal report in 1983.) Dept. of Mathematics, MIT, 1989 (see p. 9).

[TY01] Hong Tang and Tao Yang. "Optimizing Threaded MPI Execution on SMP Clusters". In: *International Conference on Supercomputing (ICS)*. 2001 (see p. 11).

[UB04] Keith D. Underwood and Ron Brightwell. "The Impact of MPI Queue Usage on Message Latency". In: *ICPP*. IEEE Computer Society, 2004, pp. 152–160 (see p. 22).

[Vie+12] Jerome Vienne et al. "Performance Analysis and Evaluation of InfiniBand FDR and 40GigE RoCE on HPC and Cloud Computing Systems". In: *High-Performance Interconnects (HOTI)*. IEEE. 2012, pp. 48–55 (see p. 25).

[Vis+06] Abhinav Vishnu et al. "Supporting MPI-2 One Sided Communication on Multi-rail InfiniBand Clusters Design Challenges and Performance Benefits". In: *High Performance Computing – (12th HiPC'05), Proceedings 12th International Conference*. Ed. by David A. Bader et al. Vol. 3769. Lecture Notes in Computer Science (LNCS). Goa, India: Springer-Verlag (New York), Dec. 2006, pp. 137–147 (see p. 67).

[VKB11] Abhinav Vishnu, Manojkumar Krishnan, and Pavan Balaji. "Dynamic Time-Variant Connection Management for PGAS Models on InfiniBand". In: *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*. IEEE. 2011, pp. 740–746 (see p. 75).

[VM03] Vetter and Mueller. "Communication Characteristics of Large-scale Scientific Applications for Contemporary Cluster Architectures". In: *JPDC: Journal of Parallel and Distributed Computing* 63 (2003) (see p. 38).

[VU05] Theewara Vorakosit and Putchong Uthayopas. "Building a Highly Scalable MPI Runtime Library on Grid using Hierarchical Virtual Cluster Approach". In: *International Conference on Parallel and Distributed Computing Systems (PDCS'05)*. Ed. by S. Q. Zheng. Phoenix, AZ, USA: IASTED/ACTA Press, Nov. 2005, pp. 524–529 (see p. 37).

[Wan+11] Hao Wang et al. "MVAPICH2-GPU: optimized GPU to GPU communication for InfiniBand clusters". In: *Computer Science-Research and Development* 26.3-4 (2011), pp. 257–266 (see p. 16).

[Wol+12] Marc Wolff et al. "High-order dimensionally split Lagrange-remap schemes for ideal magnetohydrodynamics". In: *Discrete and Continuous Dynamical Systems - Series S* (2012) (see p. 84).

[Woo+06] Timothy S. Woodall et al. "High Performance RDMA Protocols in HPC". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface (PVM/MPI)*. Ed. by Bernd Mohr et al. Vol. 4192. Lecture Notes in Computer Science (LNCS). Sept. 2006, pp. 76–85 (see p. 41, 66).

[Wu+02] Jiesheng Wu et al. "Impact of On-Demand Connection Management in MPI over VIA". In: *CLUSTER*. 2002, pp. 152–159 (see p. 38).

[WWP04] Jiesheng Wu, Pete Wyckoff, and Dhabaleswar K. Panda. "High Performance Implementation of MPI Derived Datatype Communication over InfiniBand". In: *International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, 2004 (see p. 36, 105).

[YGP06] Weikuan Yu, Qi Gao, and Dhabaleswar K. Panda. "Adaptive connection management for scalable MPI over InfiniBand". In: *International Parallel and Distributed Processing Symposium (IPDPS)*. 2006 (see p. 29, 38).

[Yu+05] Weikuan Yu et al. "Design and Implementation of Open MPI over Quadrics/Elan4". In: *International Parallel and Distributed Processing Symposium (IPDPS)*. Apr. 2005 (see p. 24).

[Inf] InfiniBand Trade Association. *InfiniBand Architecture Specification*. http://www.infinibandta.com (see p. 24, 84).

[MPI93] MPI Forum. “MPI: A Message Passing Interface”. In: *Proceedings of Supercomputing ’93*. Nov. 1993, pp. 878–883 (see p. 9).

[Ope13] OpenMP Architectural Board. “OpenMP Application Program Interface (version 4.0)”. In: July 2013 (see p. 9, 92, 114).

List of Figures

List of Tables

Glossary

CPU (Central Processing Unit) The Central Processing Unit is a chip that carries out the instructions of a computer program.

CQ (Completion Queue) A queue (FIFO) which contains CQEs.

CQE (Completion Queue Entry) An entry in the CQ that describes the information about the completed WR (its status and size, value of the immediate data, etc. . .) .

HCA (Host Channel Adapter) An HCA provides the point at which an Infiniband end node (compute node) connects to an Infiniband network. They are the equivalent of the Ethernet (NIC) card.

Infiniband Verbs Low-level end-user API for programming Infiniband HCAs. Provided by the OpenFabrics Entreprise Distribution software stack.

IP (Internet Protocol) A protocol used for communicating data across a packet-switched internetwork.

lkey A number that is received upon registration of MR is used locally by the WR to identify the memory region and its a ssociated permissions.

MTU (Maximum Transfer Unit) The maximum size of a packet payload (not including headers) that can be sent /received from a port.

NIC (Network Interface Controller) A NIC is a computer hardware component that connects a computer to a computer network.

OFED (OpenFabrics Entreprise Distribution) The OFED stack distributed by the OpenFabrics Alliance includes software-drivers, core kernel-code, middleware and user-level interfaces for accessing the three major RDMA fabric technologies – Infiniband, iWARP and RDMA over Converged Ethernet (RoCE).

OS (Operating System) An Operating System is a collection of software that manages computer hardware resources. To access hardware, operating systems expose a set of services that computer programs can use.

QP (Queue Pair) The pair (Send Queue and Receive Queue) of independent WQs packed together in one abstract object. These network endpoints (similar to TCP sockets) aim at transferring data between nodes of a network. There are two major types of QP: Unreliable Datagram and Reliable Connection..

RDMA (Remote Direct Memory Access) Accessing memory in a remote side without involvement of the remote CPU.

Remote Key (rkey) A number that is required to remotly access a memory region. It is used to enforce permissions on incoming RDMA operations.

RoCE (RDMA over Converged Ethernet) A network protocol that allows remote direct memory access over an Ethernet network.

RQ (Receive Queue) A Work Queue which holds RRs posted by the user.

RR (Receive Request) A WR which was posted to an RQ and describes where incoming data (Send-Receive and Memory channels) is going to be written. Also note that a RDMA Write with immediate will consume a RR.

RTR (Ready To Receive) A QP state in which an RR can be posted and be processed.

RTR (Ready To Send) A QP state in which an SR can be posted and be processed.

SQ (Send Queue) A Work Queue which holds SRs posted by the user.

SR (Send Request) A WR which was posted to an SQ and describes how much data is going to be transferred, its address, and which channel (Send-Receive and Memory).

SRQ (Shared Receive Queue) A queue which contains WQEs for incoming messages from any QP which is associated with it. More than one QPs can be associated with one SRQ.

WQ (Work Queue) A Send Queue or Receive Queue.

WQE (Work Queue Element) An element in a Work Queue.

WR (Work Request) A request which was posted by a user to a work queue.