12-2012

Kernel-assisted and Topology-aware MPI Collective Communication among Multicore or Many-core Clusters

Teng Ma
tma@utk.edu

To the Graduate Council:

I am submitting herewith a dissertation written by Teng Ma entitled "Kernel-assisted and Topology-aware MPI Collective Communication among Multicore or Many-core Clusters." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

Jack J. Dongarra, Major Professor

We have read this dissertation and recommend its acceptance:

George Bosilca, Jian Huang, Louis J. Gross

Accepted for the Council:

Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

Kernel-assisted and Topology-aware MPI Collective Communication among Multicore or Many-core Clusters

A Thesis Presented for
The Doctor of Philosophy
Degree
The University of Tennessee, Knoxville

Teng Ma

December 2012

Dedication

This dissertation is dedicated to Julie and Emma, who have always been there for me.

Acknowledgements

To this day, I still feel very lucky to have the opportunity to pursue my PH.D. study at the Innovative Computing Laboratory under the guidance of Dr. Jack Dongarra. I still remember the first time I met Jack in Beijing, in 2005. His insights into high performance computing, his humor and intellect, and his inspired speeches immediately captured me. This also leads into the most important decision in my life: pursuing my PH.D. study at UT. During these six years of PH.D. study, I received much help from my advisors, family, colleagues, collaborators, classmates, and friends. The experiences of the past six years have become my most beautiful memories.

First, I would like to thank my advisor, Dr. Jack Dongarra, for his mentoring, support and motivation during my PH.D. study. I was inspired by his technical talks in conferences, lunch talks, and group meetings. His talks always informed me of new technologies and innovative research ideas. His vision into scientific computing widens my knowledge and helps me decide which research topics I should focus on. I would also like to thank Dr. Dongarra for his generous finance support of this research and offering plenty of opportunities to attend conferences and workshops.

Second, I want to thank my project leader, Dr. George Bosilca for mentoring me in these six years. I am so grateful to have Dr. Bosilca as my project leader; I learned so many technical skills from him: all variations of programming tools, debugging, writing research proposals and papers, etc. The collaborative projects that I participated in with him built a solid foundation for my research. Aside from the assistance mentioned above, he also offered wise and helpful advice, on many occasions, throughout my time at UT.

Third, I would like to thank my committee members, Dr. Louis Gross and Dr. Jian Huang for valuable suggestions and for taking the time to participate in this process.

Fourth, I want to thank my best friends, classmates and colleagues: Peng and Fengguang. All the discussions and group studies with them helped my research significantly. Their constructive criticism and helpful insights into my research allowed me to push my research to another level. All discussions in the Claxton conference room have become my best memories during my PH.D. studies. I would also like to thank our administrative staff members: Teresa, Tracy R., Leighanne, Tracy L., Sam and David. Their work and assistance allowed my research and studies to proceed smoothly. Sam Crawford proofread most of my publications and gave me many valuable suggestions. I also want to thank many current and former ICLers for their valuable input and camaraderie: Aurelien, Thomas, Stephanie, Anthony, Piotr, Jakub, Shirley, Stan, Terry, Jeffrey, Yinan, and many others.

I also want to thank many collaborators outside of UT: Dr. Scott Pakin from LANL, Dr. Darren Kerbeson from PNNL, and Dr. Brice Goglin from INRIA. It's so great to collaborate with them, and their research work greatly inspired me. These collaborations widened my vision in research and built a solid foundation for my dissertation.

Finally, I want to give my special thanks to my family. In these six years, my wife, Julie, carried everything in the family on her back to let me focus on my work, especially after 2008 when we had our daughter, Emma. I also want to thank my mother, my father in law, and my mother in law for their understanding and support.

The most beautiful thing we can experience is the mysterious. It is the source of all true art and all science. - Albert Einstein

Abstract

Multicore or many-core clusters have become the most prominent form of High Performance Computing (HPC) systems. Hardware complexity and hierarchies not only exist in the inter-node layer, i.e., hierarchical networks, but also exist in internals of multicore compute nodes, e.g., Non Uniform Memory Accesses (NUMA), network-style interconnect, and memory and shared cache hierarchies.

Message Passing Interface (MPI), the most widely adopted in the HPC communities, suffers from decreased performance and portability due to increased hardware complexity of multiple levels. We identified three critical issues specific to collective communication: The first problem arises from the gap between logical collective topologies and underlying hardware topologies; Second, current MPI communications lack efficient shared memory message delivering approaches; Last, on distributed memory machines, like multicore clusters, a single approach cannot encompass the extreme variations not only in the bandwidth and latency capabilities, but also in features such as the aptitude to operate multiple concurrent copies simultaneously.

To bridge the gap between logical collective topologies and hardware topologies, we developed a distance-aware framework to integrate the knowledge of hardware distance into collective algorithms in order to dynamically reshape the communication patterns to suit the hardware capabilities. Based on process distance information, we used graph partitioning techniques to organize the MPI processes in a multi-level hierarchy, mapping on the hardware characteristics. Meanwhile, we took advantage of the kernel-assisted one-sided single-copy approach (KNEM) as the default shared memory delivering method. Via kernel-assisted memory copy, the collective algorithms offload copy tasks onto non-leader/not-root processes to evenly distribute copy workloads among available cores. Finally, on distributed memory machines, we developed a technique to compose multi-layered collective algorithms together to express a multi-level algorithm with tight interoperability between the levels. This tight collaboration results in more overlaps between inter- and intra-node communication.

Experimental results have confirmed that, by leveraging several technologies together, such as kernel-assisted memory copy, the distance-aware framework, and collective algorithm composition, not only do MPI collectives reach the potential maximum performance on a wide variation of platforms, but they also deliver a level of performance immune to modifications of the underlying process-core binding.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Introduction

While the endless demand for increasing computing power from the domain sciences motivates the deployment of powerful High Performance Computing (HPC) systems, thermal and power consumption concerns have curbed the growth of both compute node count and processor frequency. In recent years, thermic issues have prevented the straightforward performance improvement of increasing processor operating frequency. Maintaining an increase in performance therefore had to be achieved in a different way. Parallelism on a chip – in the form of multicore processors – has been widely adopted as the solution. This trend is even more pronounced when considering the systems in the TOP500 list of supercomputers [46]: multicore clusters have become the most prominent form of HPC systems, and exhibit a rapid increase in the number of cores per compute node. The top ranking machine on the June 2012 TOP500 list, the LLNL Sequoia computer, uses more than one and a half million cores*. While increasing the number of cores raises the theoretical peak performance, keeping that many processing units busy requires a significant amount of data to be transferred to and from main memory. The flat memory bus, as featured in

*http://top500.org/lists/2012/06

many legacy Symmetric Multiprocessing (SMP) *north-bridge* chipsets, is not able to sustain such a bandwidth and requested throughput, practically limiting the achievable performance to a tiny fraction of the computing peak – a problem known as *the memory wall*. To avoid this problem, most recent multicore designs embrace Non Uniform Memory Access (NUMA) and hierarchical memory interconnection to enable core count scalability and adequate bandwidth between the cores and the memory banks. Unfortunately, these new hardware feathers challenge the assumptions made by most current HPC programming models, directly threatening the performance efficiency of the machines. Namely, within compute nodes, NUMA, memory, and shared cache hierarchies weaken the assumptions of regular load balance and uniform link bandwidth and latency. These increased hardware complexities and hierarchies inside compute nodes greatly challenge applications' performance and portability.

In the era of the single-processor cluster, the Message Passing Interface (MPI) standard has enjoyed a wide adoption in the HPC community, thanks to two key features: implementations could provide the highest level of performance while maintaining the application's portability. With respect to portability, not only could an MPI code compile on different machines, but it also exhibits an excellent efficiency, because network topologies and collective communication patterns are accounted for by the MPI library rather than the application itself. With the introduction of multicore compute nodes, both of these features could lose effectiveness in MPI with most implementations treating multicore compute nodes as mere SMP units and ignoring their internal hierarchies. The resulting false assumption is that inter-process distances are the same between processes on the same compute node, which often leads to a severe performance penalty for MPI applications. Another major concern is raised about the quality of MPI implementations, and whether or not current MPI implementations can provide scalable and reliable performance with more CPU cores and memory nodes integrated into compute nodes. To alleviate these issues, some attempts have been made to use hybrid programming model, retaining MPI between computer nodes and a thread-centric approach (pthreads, OpenMP, TBB, etc.) between cores. The hybrid programming model also has several productivity drawbacks: it imposes a significant complexity on programmers, renders explicit management of hierarchies which defeats performance portability, and requires a major rewrite of legacy applications.

We believe that the MPI standard is still a competitive proposition for harnessing the power of multicore clusters. However, the current MPI implementations obviously can't satisfy the requirement of the highest efficiency and portability. As the premise of this work, we have identified three critical issues specific to collective communication on multicore clusters that prevent these systems from reaching the potential maximum performance. The first problem arises from the mismatch of the run-time process placement and underneath hardware topology due to topology-unaware MPI implementations [43]. As most of the collective communication exhibits specific communication patterns, which are allowed to adapt to the underlying architectural features (split binary tree, binomial tree, chain, etc.), there is a mismatch between the MPI internal collective topology (which is usually created based on the processes MPI ranks) and the external process placement decision, leading to a severe performance penalty. Second, current MPI communications lack efficient shared memory message delivering approaches. The usual approach for transporting message payloads between MPI processes across shared memory is based on the copy-in/copy-out (CICO) algorithm (as illustrated by the SM component in Open MPI and the Nemesis [42] device in MPICH2). This algorithm uses an extra pre-allocated shared memory buffer as an exchange zone between processes. Each message is copied into this intermediate zone by the sender process and then copied out to the destination buffer by the receiver process. This CICO approach implies two memory copies to pass a single message. Two memory copies not only greatly waste memory bandwidth and CPU cycles, but also lead the root process or leader processes in the collective communication to be heavily involved in intra-node data transfer. This overhead will be magnified when increasing the number of cores and the levels of memory hierarchy. Therefore, it becomes clear that maintaining the copy-in/copy-out approach as the default communication model for many-core architectures leads to substantial scalability and performance issues [40]. Last, on distributed memory machines, like multicore clusters, a single level approach cannot encompass the extreme variations not only in the bandwidth and latency capabilities, but also in features such as the aptitude to operate multiple concurrent copies simultaneously. Efficient multicore shared memory approaches are so specific, including kernel assisted copies, that they cannot apply to network communications; on the other hand, regular network approaches fail to extract performance out of shared memory links. This calls for a tight collaboration between multiple layers of collective algorithms, dedicated to managing intra- or inter- node communications.

1.1.1 Contribution

The techniques developed in this dissertation improve state-of-the-art MPI collective communication via the following: directly applying kernel assisted direct memory copy technique into shared memory intra-node MPI collectives; building a distance-aware framework to express MPI process distance by hardware distance and reshaping run-time logical collective topologies onto hardware topologies; and composing multi-layered collective algorithms together as a multi-level algorithm, for collectives on distributed memory machines.

Kernel-assisted collectives: The kernel assisted direct memory copy technique is deeply applied into our collective algorithms, as the default data moving mechanism, rather than the classical shared memory double memory copy approach. We developed a new MPI collective component in Open MPI, the KNEM collective, that takes advantage of both the kernel assisted direct memory copy module (KNEM Linux kernel module) and the NUMA memory subsystem. By adjusting stream direction of kernel memory copies, memory copy work-loads are always off-loaded onto non-root processes to evenly distribute memory accesses across all available cores. Both intermediate memory copies and sequential progressing on the root process are eliminated in kernel assisted collectives. Experimental results show that our approach outperforms all other stat-of-the-art MPI collective components on synthetic benchmarks. Our KNEM-enabled MPI communication can increase application performance, and expose a better scalability than the classical shared memory double copy approach when integrating more CPU cores and memory into compute nodes.

Distance-aware framework: We developed a distance-aware framework to integrate the knowledge of hardware distance for collective algorithms. The process distance reflects how many functional units, buses, caches, memory controllers, etc., messages travel through from the source to the destination memory. The process distance is computed under the assumption that each process is bound to a particular computing unit (core), and therefore the process distance is relative to the core placement at the hardware level. As a result, the distances can be measured using the memory and physical hierarchies. With the help of measured distance information, distance-aware collective algorithms dynamically generate collective topologies, which can reflect physical hardware topologies, run-time process placements, and dynamic communicators. Distance-aware framework effectively bridges the gap between logical collective topologies and underneath hardware topologies. We implemented this distance-aware framework into the KNEM collective. Experimental results demonstrate that, on a wide variation of many-core compute nodes, not only can distance-aware KNEM collectives reach the potential maximum performance, but they also deliver a level of performance immune to modifications of the underlying process-core binding.

Extension to distributed memory machines: These techniques are further extended to be applied onto distributed memory machines, like multicore clusters. The distance framework can be integrated to any type of resource allowing data movement between processes, either in terms of latency or bandwidth. At the inter-node level, the distance-aware framework can be extended to network adapters, links, or even switches and routers. To maximize the overlap between inter- and intra-node communication, we composed multi-layered collective algorithms together to form a multi-level algorithm. Similar to the distance framework used in shared memory compute nodes, based on the underneath hardware property (process distance), processes are organized into groups based on the inter-links latency and bandwidth, in most cases, two groups: inter- and intra-node groups. One leader from each compute node is selected among the core-centric collective algorithm, to participate in the inter-node collective. Intra-node communications are managed by offloading memory copies to non-leader processes, taking advantage of the kernel-assisted one- sided single-copy approach to evenly distribute the memory copy workloads among available cores. The resulting scheme enables perfect overlap of intra-node communication time by external communications, thanks to our innovative algorithm composition. We demonstrate experimentally, by considering three collective patterns (one-to-many, many-to-many, and many-to-one), that 1) this approach is immune to modifications of the underlying process-core binding; 2) it outperforms state-of-the-art MPI libraries (Open MPI, MPICH2, and MVAPICH2) demonstrating up to a 30x speedup for messages between 8KB and 256KB in synthetic benchmarks, and up to 3x speedup for a parallel graph application (ASP); 3) it demonstrates a linear speedup with the increase of the number of cores per node, a paramount requirement for scalability on future many-core hardware.

1.2 Dissertation Organization

The rest of this dissertation is organized as follows: Chapter 2 introduces MPI collective operations covered in this dissertation and a survey of the literature for each contribution. Chapter 3 describes an implementation of kernel-assisted collective algorithms (the KNEM collective) in Open MPI, covering the most widely used collective operations: Broadcast, Scatter(v), Gather(v), Alltoall(v) and Allgather(v). The end of Chapter 3 provides performance comparisons between the KNEM collective and other state-of-the-art collective implementations on both synthetic benchmarks, and widely used graph and scientific applications. Next, Chapter 4 discusses the distance-aware framework and how to build a logical collective topology based on the underneath hardware topology. Two distance-aware collective operations, Broadcast and Allgather, are implemented as examples. The evaluation experiments are presented at the end of this chapter to show that our distance-aware (topology-aware) collective operations can always provide the highest performance under any process placement, any underlying hardware architecture, and any run-time communicator. Then, Chapter 5 describes the extension work of kernel-assisted and topology-aware approaches on distributed memory multicore clusters and its corresponding implementation as a collective component in Open MPI: the HierKNEM collective. Finally, Chapter 6 concludes the dissertation and discusses the future work.

Chapter 2

MPI Collective Operations and Literature Review of Related Work

2.1 MPI Collective Operations

MPI collective operations were abstracted from a wide variation of distributed parallel algorithms, representing the most common communication patterns in MPI applications. As early as 1994, collective operations have been included into the first MPI standard (MPI-1.1) [1] as standard application interfaces (APIs). As well as MPI point-to-point communication, collective operations are frequently used in all kinds of MPI applications. For example, the Fast-Fourier-Transform (FFT) algorithm depends heavily on All-to-all communication [39]; Broadcast is widely used for data distribution like the All-Pairs-Shortest-Path (ASP) algorithm [51]; and Reduce or Allreduce is often used to collect errors. Both point-to-point and collective communication build a foundation for the MPI standard. Different from point-to-point communication, collective communication must involve all processes in the scope of a communicator. MPI collective operations could be categorized into the following types [1]:

1. Broadcast from one member to all members of a group (MPI_Bcast).

2. Gather data from all group members to one member (MPI_Gather(v)).

3. Scatter data from one member to all members of a group (MPI_Scatter(v)).

4. A variation on Gather where all members of the group receive the gathered result (MPI_Allgather(v)).

5. Scatter/Gather data from all members to all members of a group (MPI_Alltoall(v)).

6. Global reduction operations such as sum, max, min, or user-defined functions, where the result is returned to all group members: MPI_Allreduce or the result is returned to only one member: MPI_Reduce.

7. A combined reduction and scatter operation (MPI_Reduce_scatter).

8. Scan across all members of a group (MPI_Scan).

9. Barrier synchronization across all group members (MPI_Barrier).

We focus on the first 6 types of the above collective operations in this dissertation, because these operations represent the most used algorithms in MPI applications, and these operations' performance is greatly threatened by increasing amounts of data and hardware complexity. MPI applications' performance is often sensitive to the quality of these operations' implementations.

Depending on whether or not the 'root' process existed, these six types of collective operations can be further typed into 'rooted' or 'non-rooted' operations. For example, operations, like Broadcast, Gather, Scatter, and Reduce, all belong to rooted operations; non-rooted operations include Allgather, Alltoall, and Allreduce. The following subsection 2.1.1 and subsection 2.1.2 describe the API definition of rooted or non-rooted operations in the MPI standard [1].

2.1.1 Rooted Operations

1. Broadcast:

```
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype,
              int root, MPI_Comm comm)
```

MPI_Bcast broadcasts a message from the process with rank 'root' to all processes in the communicator. This function is called by all processes in the communicator. Once returning from this function, the content in the root process's buffer has been copied into all processes' buffers.

2. Scatter(v):

```
int MPI_Scatter(void *sendbuf, int sendcnt,
                MPI_Datatype sendtype, void *recvbuf,
                int recvcnt, MPI_Datatype recvtype,
                int root, MPI_Comm comm)
```

MPI_Scatter splits a message beginning from the send buffer (sendbuf) in the root process into N equal segments (each segment has the same count of 'sendcnt' of data). The ith segment is sent to the ith process in the communicator, and each process (including the root process) receives this message, and stores the received message in the receive buffer (recvbuf).

```
int MPI_Scatterv(void *sendbuf, int *sendcnts, int *displs,
                 MPI_Datatype sendtype, void *recvbuf,
                 int recvcnt, MPI_Datatype recvtype,
                 int root, MPI_Comm comm)
```

MPI_Scatterv is a vector version of MPI_Scatter, which allows a varying count of data to be sent to each process. The count of data sent to each process is put into an integer array 'sendcnts'.

3. Gather(v):

```
int MPI_Gather(void *sendbuf, int sendcnt,
               MPI_Datatype sendtype, void *recvbuf,
               int recvcnt, MPI_Datatype recvtype,
               int root, MPI_Comm comm)
```

MPI_Gather is the inverse operation of MPI_Scatter. Each process (include the root process)sends its send buffer (sendbuf) to the root process. The root process stores these received messages into its receive buffer (recvbuf) in rank order.

```
int MPI_Gatherv(void *sendbuf, int sendcnt,
                    MPI_Datatype sendtype, void *recvbuf,
                    int *recvcnts, int *displs,
                    MPI_Datatype recvtype,
                    int root, MPI_Comm comm)
```

MPI_Gatherv is a vector version of MPI_Gather and is also the inverse operation of MPI_Scatterv. Each process sends 'sendcnt' count to the root process. Root process stores these data according to the integer array displs and the corresponding rank.

4. Reduce:

```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count,
                    MPI_Datatype datatype, MPI_Op op, int root,
                    MPI_Comm comm)
```

MPI_Reduce combines data provided in the send buffer (sendbuf) of each process by using the operation 'op', and returns the combined result into the receive buffer (recvbuf) of the root process.

2.1.2 Non-rooted Operations

1. Allgather:

```
int MPI_Allgather(void *sendbuf, int sendcount,
                  MPI_Datatype sendtype, void *recvbuf,
                  int recvcount, MPI_Datatype recvtype,
                  MPI_Comm comm)
```

MPI_Allgather can be thought of as a combination of MPI_Gather and then MPI_Bcast. In MPI_Allgather, all processes receive the result instead of just the root process.

2. Alltoall:

```
int MPI_Alltoall(void *sendbuf, int sendcount,
                 MPI_Datatype sendtype, void *recvbuf,
                 int recvcount, MPI_Datatype recvtype,
                 MPI_Comm comm)
```

MPI_Alltoall is an extension of MPI_Allgather in which each process sends a distinct data to each process in the communicator. After an Alltoall operation, the ith block data sent from process j is received by process i and placed into jth block.

2.2 Literature Review

Clusters of multi- and many-core nodes are currently the most popular platform in high performance computing. According to underlying link properties between processes, collective communication on multicore clusters can be transformed into one of the following three scenarios: inter-node collectives, intra-node collectives, and inter- and intra-node collectives. When one MPI process from each compute node participates in collective communication, and any pairs of MPI processes in collective are between a distance of inter-node links, it becomes an inter-node collective; when all MPI processes reside inside a single shared memory compute node, it transforms into an intra-node collective; the inter- and intra-node collective is a hybrid form of the above two scenarios: processes on the same compute node are across intra-node links and processes residing on different compute nodes are across inter-node links.

A wide variant of latency and bandwidth from different underlying links makes it almost impossible for a single collective component to tackle all scenarios very well. Most MPI libraries provide several collective components selected for each scenario. For example, in Open MPI, the *SM* or *Tuned* collective component can be selected for intra-node collectives; the *Tuned* collective component can be used as inter-node collectives; and the *Tuned* or *Hierarch* collective component can be selected to tackle inter- and intra-node collectives. However, these classic collective components are far from optimal solutions on the new hardware architecture. With increased hardware resources integrated into compute nodes, these collective components easily become performance bottlenecks. There is an urgent demand for reworking these collective components by using innovative techniques to fit onto new hardware architectures.

2.2.1 Inter-node Collective Communication

Inter-node collective communication is widely used in today's hybrid programming models: using the message passing technique to communicate between computer nodes; and pthreads, TBB, or OpenMP to communicate between cores inside compute nodes. Inter-node collective communication often inherits collective algorithms designed for single-processor clusters. Plenty of research efforts have been done to improve collective algorithms on single-processor clusters, e.g., making collective algorithms be aware of network topologies and hierarchies, smart network interfaces (NICs), multipath links between processes, etc. The focus of inter-node collectives is toward tackling network complexity and hierarchies among inter-node links.

Most MPI libraries provide host-based and software-based collective components, like Open MPI's *Tuned* collective. The *Tuned* collective component uses send-receive primitives provided by point-to-point communication, adopts different communication topologies (linear, chain, split binary tree, binomial tree, etc.), [31] and enables parallel treatment of the message through pipelining. In the *Tuned* collective component, a run-time decision is used to select the best algorithms and parameters according to message size, communicator size, and other parameters [20]. As an example, for a Broadcast in the *Tuned* collective component, a binomial algorithm is used to deliver small messages, a split binary tree algorithm is selected for intermediate messages, and large messages are transferred by a pipeline algorithm in which the pipeline size varies with the message and communicator size. Most MPI libraries select this *Tuned*-style collective component by default, e.g., MPICH2 [57] and Open MPI [20]. This software-based collective communication was originally designed for general purpose single-processor clusters. It reuses the highest efficient implementation of point-to-point communication to fully utilize underneath inter-node links, e.g., high speed Ethernet, InfiniBand, etc. Tuning among a large set of algorithms and parameters can help the *Tuned* collective to deliver a level of performance close to potential hardware capacities.

Another effort to optimize inter-node collective communication focuses on hardware-based or NIC-based collective communication. With the development of interconnect networking technologies, some specific high performance networks, e.g., Myrinet, InfiniBand [4], Quadrics [9, 50], IBM PERCS network [53], Cray SeaStar [11], are equipped with smart Network Interface Cards (NICs). Programmable processors integrated inside smart NICs can be utilized to offload collective communication from host CPUs, like Mellanox FCA (Fabric Collective Accelerate) on Mellanox connect-3x InfiniBand cards [54], the Elan library on Quadrics networks [2], hardware-assisted collectives on IBM Blue Gene/L/P/Q systems [8, 15, 19] and etc.

The early efforts were toward building NIC-based collective operations on top of hardware-based collective-style operations. For example, in Quadrics, NIC-based barrier and broadcast operations are based on hardware-based multicast primitives provided by the Elan library [17]. A similar approach is applied to Myrinet, like NIC-based multicast [62] and all-to-all broadcast [63]. The early smart NICs are less powerful than host CPUs and some of them do not have floating-point units like Quadrics [47].

IBM keeps advancing hardware-based collective communication on their Blue Gene systems (Blue Gene/L/P/Q). A collective network is used for optimized collectives and communication with I/O nodes. Most MPI collective operations are built on hardware-based collective primitives from the collective network. For example, on a Blue Gene/P, Broadcast is based on the fast hardware Allreduce primitive provided by the collective network: the root process injects data while other processes inject zeros in a global OR Allreduce operation [21].

Mellanox is another industry vendor advancing hardware-based collective technology on their InfiniBand networks. The latest effort is the Mellanox Fabric Collective Accelerate (FCA [54]) library on their connect-3x InfiniBand cards.

Compared with software-based collective communication, hardware-based collective communication has the following advantages: 1) MPI applications can directly call hardware-based collective communication instead of forwarding users' requests to hardware by operating system (OS). This OS-bypassing design can eliminate intermediate memory copies and deliver a lower latency than through operating systems. 2) It is more convenient for hardware-based collective communication to fully utilize hardware architecture to reach potential peak performance. For example, hardware-based all-to-all operations on IBM Blue Gene can easily use multipath links inside a torus network to achieve a high bandwidth. 3) It can reduce the impacts of OS jitter over collective performance thanks to the OS-bypassing design. [22] 4) Host processors can dedicate themselves to computation by offloading collective communication onto smart NICs, leading to more overlaps between communication and computation. This offloading feather plays an important role in non-blocking collective communication proposed in the latest MPI 3.0 standard [7].

The limitation of hardware-based collective communication mainly comes from its platform-specific characteristics: 1) Most hardware-based collective components are tightly coupled with specific hardware. These components often require support of specific hardware firmware and device drivers, making it almost impossible to deploy them on a wide variation of commodity HPC platforms. 2) Putting these collective communication functions into hardware often leads to more hardware complexity and increases hardware costs, such as the message units (MU) of IBM Blue Gene/P systems. [8, 21]. Integrating more software functions into hardware is still controversial. 3) Most hardware-based collective components only provide a subset of collective operations. For example, the Quadrics Elan library only provides elan_bcast, elan_hbcast, elan_gsync, elan_reduce, elan_gather, elan_alltoall, etc. These collective functions are much less than collective operations defined in the MPI standard, and also requested by MPI applications. Some hardware-based collective functions' signatures are completely different than the definition in the MPI standard, which requires extra coding work when called by MPI applications.

2.2.2 Intra-node Collective Communication

Different than inter-node collective communication, intra-node collective communication focuses on collective communication on a shared memory multicore compute node. With increasing CPU cores and memory nodes integrated into compute nodes, intra-node collectives become equally important as inter-node collectives. Several optimizations have been used to maximize the throughput of intra-node collective communication.

One of the most recent of those efforts is due to Richard Graham et al., who proposed a shared memory-based fan-in/fan-out implementation for multicore MPI collectives, implemented in the Open MPI SM collective component [27]. Their optimization focuses on lightweight synchronization, reducing memory copy times, increasing parallelism by copying messages in a pipelined way, and controlling working set size to fit into caches by building a logical fixed degree tree. This shared memory based approach simply treats a multicore/many-core compute node as an SMP system and ignores other architecture characteristics, such as NUMA, memory hierarchy, and core distance. The fixed degree tree is built following the logical ranks layout, which cannot always reflect architecture characteristics. With more heterogeneity in modern NUMA multi-core designs, it is hard to optimally tune a shared memory based implementation for different platforms.

Most state-of-the-art MPI libraries also select a *Tuned*-style collective component as a default intra-node collective component on a shared memory compute node. Although *Tuned*-style collective components were originally designed for single-processor clusters, they can still deliver good performance on SMP-style compute nodes with a careful tuning on shared memory links. However, tuning for an unknown platform is not a trivial task, even for expert users. Indeed, there are too many parameters such as pipeline size, thresholds, etc., and any wrong selection might ruin the overall performance of the *Tuned* collective. Another severe issue for *Tuned* collectives is, their logical collective topologies are built based on MPI ranks, which can't reflect underneath multicore or many-core topologies. The mismatch between collective topologies and hardware topologies often causes performance penalties [43]. To further exacerbate this issue, intelligent process placements, used as a bridge between applications and MPI libraries, e.g., MPIPP [16] or TreeMatch [33], often break regular process-core binding patterns and schedule continuous processes (in the way of MPI ranks) to cores between a long physical distance. This irregular mapping leads into a further mismatch between collective topologies and underneath hardware topologies [43].

Another orthogonal work is toward improving the efficiency of shared memory inter-process communication. Copy-in/copy-out in a shared memory segment is still the most common approach for transferring data between processes on a shared memory platform. The copy-in/copy-out approach implies two memory copies to pass a single message, greatly wasting memory bandwidth and CPU cycles. More severely, when applying the copy-in/copy-out approach into rooted collective operations like Gather(v) or Scatter(v), the 'copy-in' or 'copy-out' operation in the root process is executed sequentially by N times (N is often the number of cores per compute node).

These sequential memory copies on the root process cause a serious scalability issue with more CPU cores integrated into compute nodes.

Several platform-specific efforts offered single-copy inter-process communication (IPC). For instance BIP-SMP implemented such an optimization for Myrinet based clusters [24]. This idea has spread into most vendor specific HPC stacks, such as Myricom MX, Qlogic IPath, and Bull MPI. Some lightweight kernels enabled an even bigger rework of the model on Cray platforms, thanks to the ability of the operating system to make all processes address spaces accessible to any of them. This unusual feature enabled single-copy RMA-based communication (SMARTMAP [10]), which greatly reduces memory copies needed by intra-node message passing, especially for collectives. Recent Linux kernels support the remapping of other processes' memory thanks to the XPMEM module [49] which enables similar optimizations but is restricted to SGI or Cray machines.

Other efforts towards single-copy IPC on general purpose platforms is the kernel-assisted approach. Lei Chai et al. introduced a kernel memory copy module interface called LiMIC [34]. This kernel-based approach can reduce the number of necessary memory copies to one. Another similar kernel module called by CMA (cross memory attach) has been integrated into the latest Linux kernel 3.1. KNEM [6, 26] is another similar kernel module used in MPICH2 and Open MPI. KNEM offers additional features such as an asynchronous copy model, vectorial buffer support, and copy offload on dedicated hardware [59]. This approach has already proved to be valuable to increase point-to-point bandwidth between processes communicating over shared memory compute nodes [13]. Stephanie Moreaud et al. [48] proved collective communication can get free upgraded performance by using single-copy kernel-assisted memory copy instead of a shared memory copy-in/copy-out approach as underneath point-to-point communication. However, even this upgrading is still not enough to reach full potential performance. Collective algorithms need more control over the underlying memory copy mechanism.

2.2.3 Inter- and Intra-node Collective Communication

Multicore clusters are a distributed extension of multicore compute nodes interconnected by networks. Compared with shared memory multicore compute nodes, distributed multicore clusters include both inter-node and intra-node links, which requires that, an efficient collective algorithm must consider multi-layer network hierarchies and memory hierarchies into building correct collective topologies. As mentioned above, Open MPI's *Tuned* collective component was designed for 'single-processor' clusters and is unaware of these network or memory hierarchies from multiple layers. So tuning itself can not help the *Tuned* collective component find optimal algorithms or parameters on each multicore cluster.

Therefore, there is a strong demand for collective algorithms to be aware of underneath hardware topologies and to map logical collective topologies onto hardware topologies. Leader-based hierarchical collective algorithms become very popular on distributed multicore clusters [35, 36, 44, 45, 52, 64]. The early trying of the hierarchical approach focuses on collective communication on clusters of SMPs [36] or Grids [37] to reduce the message amount crossing high latency and low bandwidth links. Combining with the SMP-aware method, leader-based hierarchical algorithms were widely applied into collective communication on multicore clusters, e.g., MPI on Quadrics networks [52], Open MPI's Hierarch collective and ML collective [28], and MVAPICH2 on InfiniBand networks [35, 44, 45]. In these SMP-aware methods, multicore nodes are often treated as shared memory nodes. The layered collective components handle inter- or intra-node communications respectively, and MPI processes in each layer are interconnected by the same links: inter- or intra-node links. As a consequence, Open MPI's *Tuned* collective can be used again to find the best algorithms and parameters for each layer. Composing these *Tuned* collective algorithms from each layer together can indeed achieve better performance than tuning a single algorithm for the whole multicore cluster. However, the layered collective components that handle inter- or intra-node communications do not cooperate tightly, leading to suboptimal pipelining and sometimes contradictory tuning choices. This results in another difference with our proposed work: the intra-node communication is mainly implemented by a copy-in/copy-out (CICO) approach using a shared memory segment.

The CICO approach requires double memory copies for each message, greatly wasting memory bandwidth and CPU cycles. When applying this approach into leader-based hierarchical algorithms, leader processes are heavily involved in intra-node data movement, resulting in serializing the inter- and intra-node communications. Most of the intra-node communication overhead accumulates and results in a significant overhead that cannot benefit from overlap by inter-node communications. Obviously, such overhead is bound to increase with the number of cores: the copy-in/copy-out at the leader process has to be sequentially executed once for each of the processes participating in the collective communication.

Chapter 3

Kernel-Assisted Shared Memory Collective Communication

3.1 Introduction

To satisfy the increasing demands of computing power from domain sciences, multicore processors have been widely adopted in HPC systems. Processors with 8 to 12 cores are very popular today, and it is very common to install such processors into multiple socket boards featuring 8 to 96 cores per compute node, with network-style interconnection between caches or to the memory banks (e.g., Intel QPI or AMD Hyper-transport). These systems are expected to feature, in the near future, *fat* many-core compute nodes composed of more than 100 cores.

These new hardware feathers greatly challenge performance of state-of-the-art MPI libraries on shared memory multicore and many-core compute nodes, especially their collective communication.

Although researchers have been advancing the state-of-the-art in collective communication performance for years, the investigative focus has typically been on the complexity and performance difficulties posed by inter-node network interconnects. We believe that these collective communication techniques used with hierarchical network interconnects should now also be applied to multicore compute nodes or clusters featuring many cores and NUMA architectures.

The usual approach for transporting message payloads between MPI processes across shared memory is based on the copy-in/copy-out (CICO) algorithm. This algorithm uses an extra pre-allocated shared memory buffer as an exchange zone between processes. Each message is copied into the intermediate zone by the sender process and then copied out to the destination buffer by the receiver process. This CICO approach implies two memory copies to deliver a single MPI message, which greatly wastes CPU cycles and memory bandwidth, and increases memory usage and cache pollution. More severely, when applying this CICO approach into some rooted collective operations, e.g., Gather or Scatter, N times of copy-in or copy-out will be sequentially executed on the root process. This serial progressing causes unacceptable performance losses on collective communication on multicore compute nodes [42]. With the emergence of many-core compute nodes, there are strong demands of alternative efficient methods of inter-processes communication. One-sided sing-copy inter-process communication techniques have emerged in the form of kernel assisted memory copy (such as KNEM [6, 13], LiMIC [34], XPMEM [49], CMA [61], etc.). These kernel assisted approaches utilize single-copy transfers, and have proved beneficial in the context of point-to-point communications [13, 41]. The focus of this Chapter will be how to directly apply kernel assisted memory copy into shared memory collective algorithms.

First, we identified three critical issues specific to collective communication between cores that prevent fully utilizing the benefits of kernel assisted memory copies. The first problem arises from the contentions imposed on the root process in one-to-many or many-to-one collective operations. Every process has to wait for the progression of the core hosting the root process in the copy of data to or from the intermediate buffer. This effect actually prevents any potential acceleration from having multiple cores available to undertake multiple copies simultaneously. Second, many algorithms do not take into account temporal locality, which results in cache-ready data being discarded and then reloaded multiple times. More cache pollution, in turn, leads to a plummeting memory bandwidth as more data lines are reclaimed from the slow and contention-prone memory banks. Last, many implementations ignore topological characteristics such as NUMA and network-style processor interconnects. The blind application of the *one size fits all* collective algorithm can lead to unnecessary traffic between sockets, potentially overwhelming some memory links while under-utilizing others. These issues can be solved via 1) extending the function interfaces (APIs) of kernel assisted memory copy software modules to allow the specification of copy direction and granularity, and 2) reworking the collective algorithms themselves to detect and exploit locality in NUMA architectures.

We develop new MPI collective communication algorithms that take advantage of the NUMA memory subsystem to avoid memory contention and to maximize the overall sustained bandwidth. Our approach is based on the KNEM [6] Linux kernel module – a software that enables single memory copy between processes. We investigate several different optimizations to collective algorithms that maximize both parallelism and pipelining. A key point in our design is that multiple processes can access the same buffer – or different parts of the same buffer – simultaneously, without the need for more than a single memory copy between processes. Moreover, stream direction control enables the collective algorithm to select sender-writing or receiver-reading according to the communication pattern (many-to-one or one-to-many) to avoid the root process bottleneck. Last, our collective algorithms can detect distance between hardware units to build an optimized communication topology that minimizes inter-socket traffic. Each of these approaches are evaluated experimentally on a variety of different hardware setups, exhibiting a better scalability when increasing the number of cores, leading to substantial performance gains on many-core hardware.

The rest of this chapter is organized as follows: Section 3.2 introduces some key concepts of the KNEM kernel assisted memory copy. Next, Section 3.3

discusses a kernel-assisted and NUMA-aware Broadcast algorithm on large NUMA compute nodes: combining KNEM kernel memory copies with a NUMA topology aware hierarchical layout. Then, Section 3.4 describes the KNEM collective algorithms: one-to-many (Broadcast and Scatter), many-to-one (Gather), many-to-many (Alltoall and Allgather), and their corresponding implementations in a new collective component for Open MPI: the KNEM collective. All those algorithms are compared experimentally with state-of-the-art MPI implementations: Open MPI and MPICH2, on both synthetic benchmarks and scientific applications in Section 3.7. Finally, Section 3.8 concludes the chapter with a discussion of the results.

3.2 Kernel Assisted Memory Copy

Copy-in/copy-out (CICO) in a shared memory segment is still the most common approach for transferring data between MPI processes on a shared memory platform. Most MPI libraries provide efficient implementation of a shared-memory based copy-in/copy-out algorithm as default shared memory inter-process communication (IPC), like the SM (Shared Memory) BTL (Byte Transfer Layer) in Open MPI [23] and the Nemesis [14] device in MPICH2 [30]. The CICO algorithm uses an extra pre-allocated shared memory buffer as an exchange zone between processes. Each message is copied to this intermediate zone by the sender process and then copied to the destination buffer by the receiver process. The most prominent drawback is the necessity to copy every data twice.

There are some efforts toward single-copy inter-process communication technique on special platforms with operating system extensions. For example, SMARTMAP enables direct access to other processes' memory space in the Catamount system with a light weight kernel [10]. However, it can only be used on Cray XT platforms. Another platform-specific technique is XPMEM (Cross-Process Memory Mapping) [49] Linux kernel module, which implements a single-copy mechanism via remapping other processes' memory. But it can only be deployed on some platforms from SGI or Cray.

Another important alternation is kernel assisted direct memory copy provided by a Linux kernel module like KNEM [6], LiMIC [34], or CMA [61]; because the operating system kernel has complete access to the memory space of both sender and receiver processes, it can perform the copy from the source buffer in the sender's address space directly to the target buffer in the receiver's address space without the need for an intermediate buffer.

KNEM is an example of such a Linux kernel module that enables high-performance, inter-process, one-copy memory copies. It offers support for asynchronous and vector data transfers. KNEM can also offload memory copies to a hardware DMA engine (such as the Intel I/O AT hardware [59]), if available. More importantly, KNEM not only supports send/receive function interfaces for point-to-point communication, but also provides primitives supporting collective communication since its version 0.7.1, and declared memory regions can be operated simultaneously by multiple remote processes.

LiMIC is a similar Linux kernel module designed by MVAPICH2 communities for kernel assisted direct memory copy between MPI processes. But it only offers send/receive primitives used for MVAPICH2's large message's point-to-point transport (LMT). CMA (Cross Memory Attach) provides similar features with LiMIC by using two new operating system system calls in Linux Kernel 3.2. Due to only supporting send/receive primitives, it is very challenging to adapt CMA or LiMIC directly into collective algorithms. So, we select the KNEM kernel module as our kernel assisted memory copy approach to develop kernel-assisted collective algorithms.

3.2.1 MPI Integration of KNEM Memory Copy Module

It was not long before the above-cited features raised interest for improving MPI intra-node communications. Open MPI v1.5 includes KNEM support in its shared memory point-to-point communications component. MPICH2 v1.1.1 implements a KNEM LMT (large message transfer) to improve large message performance within a shared memory compute node. The general operating principles of the integration between KNEM and MPI point-to-point communication are the following (more details can be found in [13]): 1) The sender process declares a send buffer to KNEM. The kernel module saves the list of virtual segments contained in the buffer and associates them with a unique cookie. 2) The sender passes the cookie to the process which is interested in this buffer by an out-of-band transfer. 3) The receiver gives this incoming cookie to KNEM along with a receive buffer. 4) The KNEM module copies the data from the send buffer to the receive buffer within the kernel. 5) The initiator (sender or receiver process) will notify the remote peer after KNEM memory copy is finished. 6) The sender process deregisters the mapping between sender buffer and the KNEM kernel device. The cookie here is actually an opaque identifier which is used to tell the OS kernel about the send buffer layout from the receiver side [26]. With memory buffer information on both sender and receiver processes, a direct memory copy can be triggered from the OS kernel.

The security model of this strategy is similar to System V IPC shared memory segments. Declaring a memory buffer to the KNEM driver makes it available to any other process. However, other memory regions cannot be accessed unless explicitly declared to KNEM as well. A malicious user modifying a cookie value would either get an invalid parameter error or get a valid access to a previously properly declared buffer.

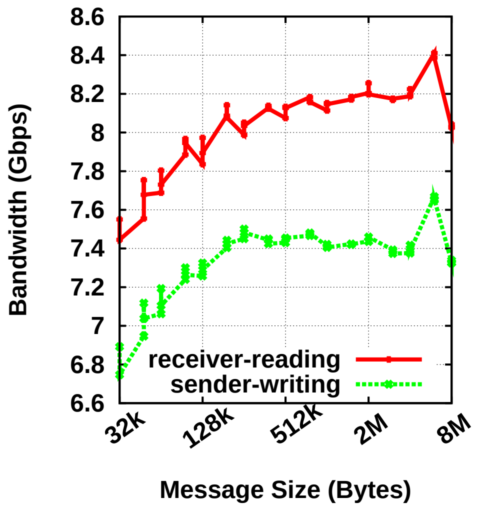
3.2.2 Issues with MPI Collective Operations

While the beneficial effect of KNEM on point-to-point performance also translates into collective improvements [48], we have identified a series of additional optimizations that further boost collective communication performance. They require that the collective component has more control over kernel assisted direct memory copy instead of simply using kernel assisted point-to-point primitives: 1) If control of the kernel module is delegated to the point-to-point MPI message passing engine, using inter-process kernel-assist memory copies results in the same data region being registered multiple times when sent to different destination processes. The overhead of synchronizing and exchanging cookies therefore cannot be amortized. The collective component knows when the same buffer is used with multiple recipients, and can therefore eliminate redundant registrations. 2) Many collective algorithms exhibit a one-to-many communication pattern. Such a communication pattern stresses the root process resources. Although in kernel execution space when using KNEM, only the core hosting the root process is performing all the data movements. This actually serializes all memory copies and limits the progression of the collective to the speed of only a single core (i.e., where the root process is executing), even though multiple cores are available (and probably waiting for the collective operation to progress). Attempting to alleviate this issue by inverting the point-to-point algorithm – i.e., having the receiver(s) make the copies – simply results in degrading many-to-one communication patterns, instead. Thus, it is desirable for the collective algorithm to be able to express the direction of data transfer, independently of concurrent point-to-point strategies.

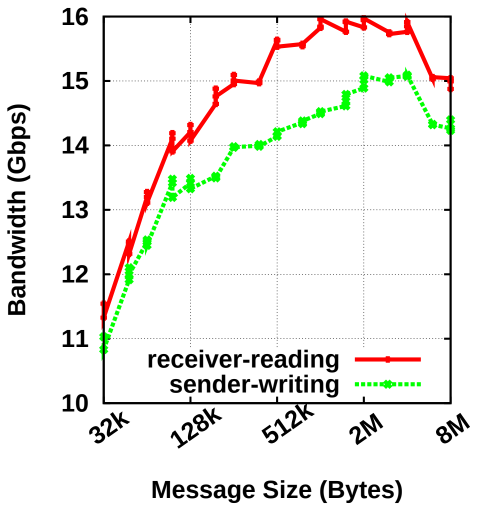
Although using kernel-assisted direct memory copy as underneath point-to-point communication has successfully improved the performance of applications [13, 48], a tight collaboration between collective algorithms and the kernel assisted memory copy module is required to further optimize performance.

3.2.3 KNEM Direction and Granularity Control

Implementing collective operations directly on top of the original KNEM send/receive interfaces greatly wastes system resources. Collective patterns such as one-to-many or many-to-one would declare the same multiply-accessed memory buffer multiple times and pass multiple KNEM cookies between processes for synchronization. To



(a) on Tigerton



(b) on Istanbul

Figure 3.1: Point-to-Point communication bandwidth with KNEM on Intel Tigerton and AMD Istanbul platforms.

solve this issue, we introduced (available since KNEM 0.7) extended programming interfaces designed to address the needs of collective operations. Instead of only offering a point-to-point send/receive interface, KNEM now offers the ability to register persistent memory regions and access them multiple times from different processes. Such accesses may either touch all or only part of the region, enabling the actual copy a single message at once or as multiple chunks simultaneously. This model avoids wasting system resources and reduces the overhead of synchronizing processes when creating and passing region identifiers.

Another extension added to KNEM is the ability to read (receiver-reading) or write (sender-writing) to each region, enabling effective direction control of the data transfer. Figure 3.1 presents the bandwidth of KNEM point-to-point communication on two very different platforms, Zoot (Intel Tigerton) and IG (AMD Istanbul). These two platforms are depicted in Section 3.7.1. This experiment uses the KNEM module in NetPIPE release 3.7.2 [3] with the off-cache option enabled. By default, the KNEM backend in NetPIPE uses a receiver-reading method, but we added support for sender-writing to compare the performance of both flow directions. We can see that receiver-reading performance is a slightly better than sender-writing on both platforms. One reason for this difference lies in the actual copy implementation in the Linux kernel which is more optimized in the receiver-reading scheme. While this experiment only shows the case when the two communicating processes are on the same socket, similar behaviors have been observed independent of the processes placement. One can see that for solely point-to-point communication, direction control can't provide extra benefits, as one direction (receiver-reading) always provides better performance. However, direction control is a very important feature to unleash the performance of collective operations. The effective direction control of the copies can be decided by the collective component to match the communication pattern (one-to-many or many-to-one), with the goal of maximizing the number of cores participating to the progression of the data transfers, and parallelize the progression of the collective algorithm as much as possible.

Those two novel features introduced into the KNEM kernel module are used by our KNEM collective component in Open MPI. Unlike previous components that only used kernel-assisted copies simply through the point-to-point interfaces [48], this component takes advantage of directional control and persistent registrations to further increase the performance achieved on collective communication.

3.3 NUMA aware collective communication

A significant amount of work has been done over the past decade toward improving the collective communication performance by taking into account the network features. Some algorithms take advantage of the low latency of some specific high-performance, while others capitalize on the bandwidth capabilities. Additionally, the network topology (butterfly, torus) has been another important factor in re-designing collective algorithms. While this work is significant, and has clearly influenced our approach,

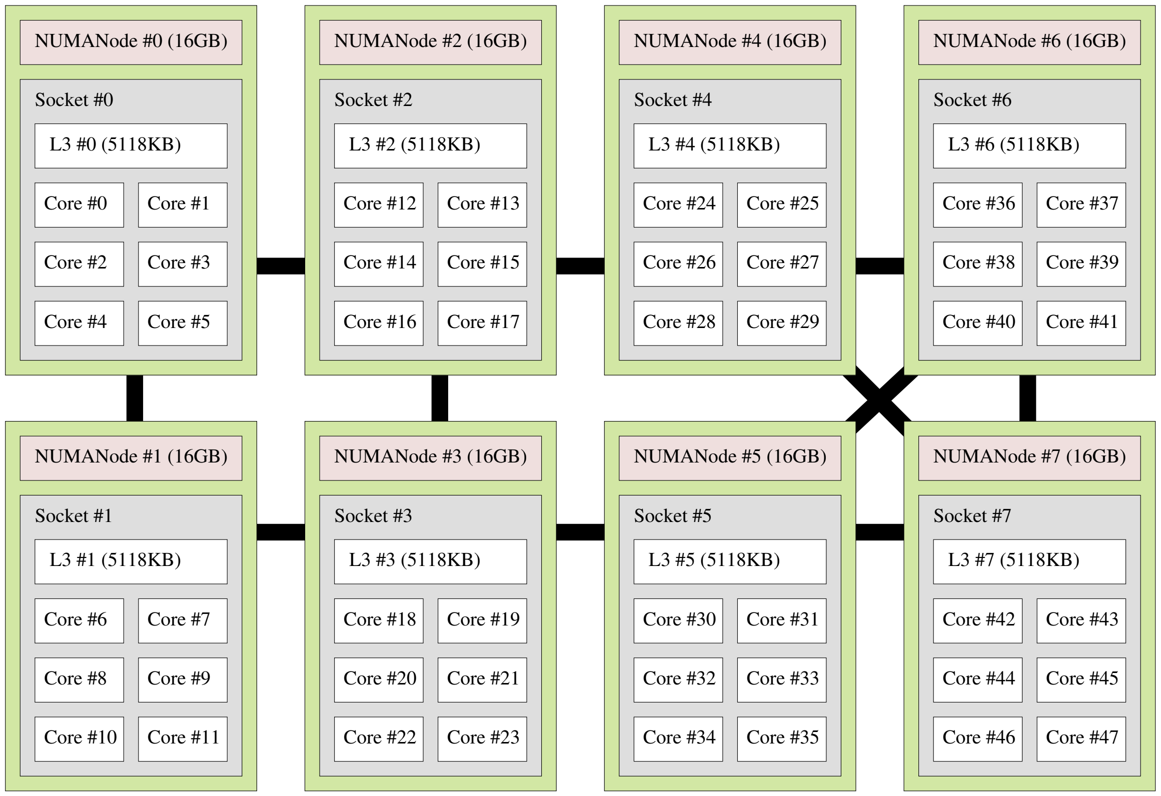


Figure 3.2: Architecture of IG. 8 NUMA nodes, containing 6 cores and 16GB each, are interconnected through HyperTransport links. Each core also has its own 64kB L1 and 512kB L2 cache.

one has to keep in mind the tight conditions required for collective algorithms in shared memory. The memory access latency and bandwidth are important, but not more important than the topology of the links to and from the memory banks. Therefore, the algorithms designed in this context are significantly more complex than the usual two-level hierarchical collective algorithms [38], and take into account the topology as well as the memory access performance, that usually exhibit several order of magnitude performance differences for different levels in the hierarchy. In order to explain how the collective algorithm will be affected by the hardware architectural features, let us take our large NUMA node (IG) as an example. This machine consists of 8 NUMA nodes, each of them containing a six-core AMD Opteron processor and 16GB of local memory. 8 NUMA nodes are interconnected by AMD HyperTransport (HT). The Figure 3.3 shows how a 48 processes broadcast will unfold on this large

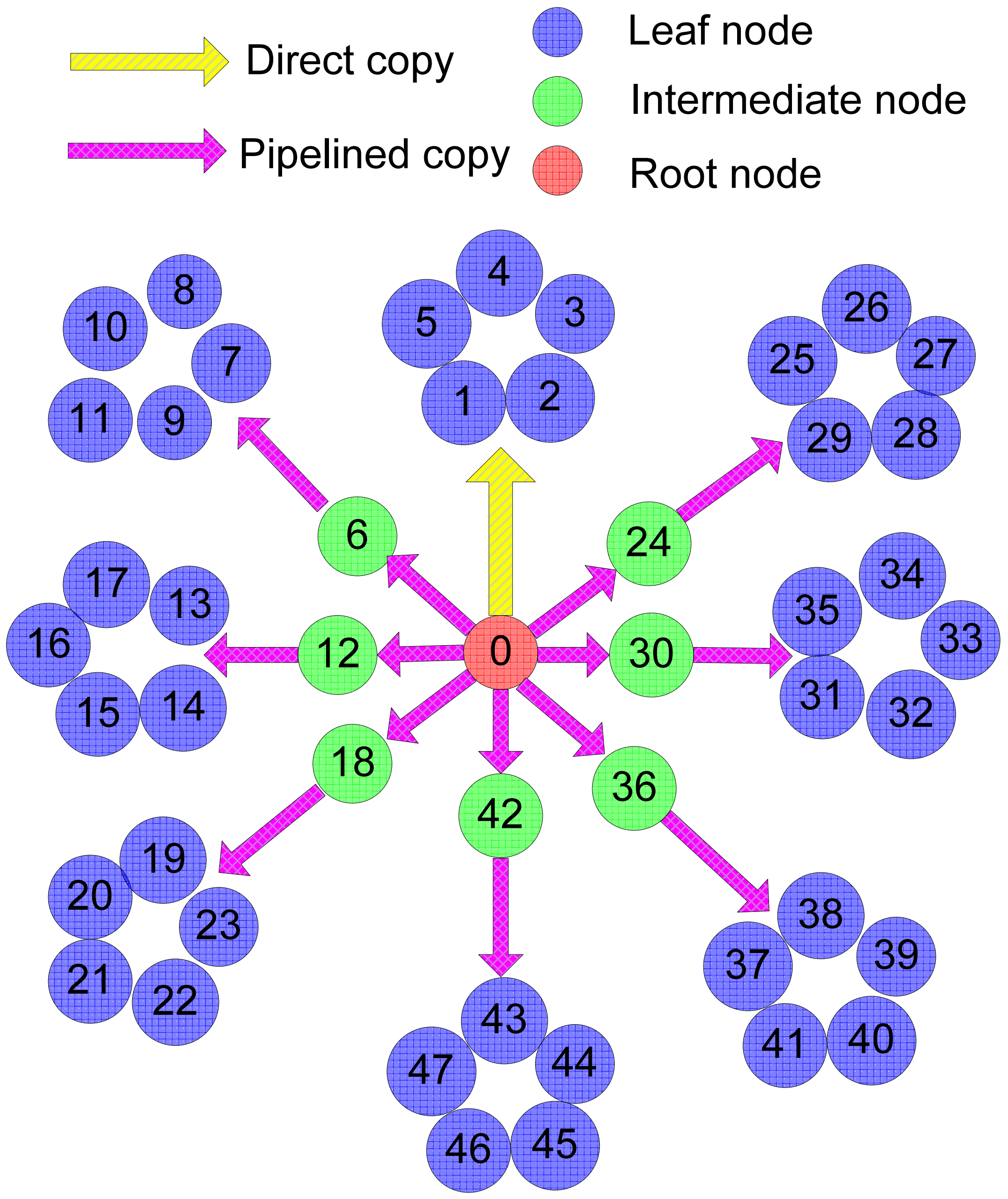


Figure 3.3: Progression of the hierarchical pipeline KNEM Broadcast

NUMA node depicted in Figure 3.2. Processes are split into 8 sets according to their NUMA locality information, which means processes within the same socket and NUMA node are in the same set. A two-level tree is then built accordingly: one process per NUMA node will belong to the first tree level (the green background circles), while all the remaining processes per NUMA node will belong to the second level (the blue background circles), and behave as leafs to the tree. This tree structure reflects the architecture topology such as core distances and relationships which can be retrieved thanks to the *Hardware Locality*[12] software.

Dividing the processes based on NUMA architecture has the advantage of reducing inter-socket traffic since a single remote memory data transfer is performed towards each NUMA memory node. Moreover, since a cache is shared between all processes inside the same NUMA node, multiple copies between processes in the same set benefit from cache hits. However, one of the disadvantages of such an algorithm is the reduction in the degree of parallelism between the memory copies toward NUMA nodes, and between the leaf nodes and their corresponding intermediary node (a leaf process cannot start a memory copy until the intermediary process received the entire data). To alleviate this strong synchronization requirement, and therefore minimize the unnecessary waiting on the leaf nodes, we divide the data in several segments and pipeline the copy operations corresponding to each one of these segments (pink arrows). Finding a suitable pipeline size is critical to the performance of hierarchical pipelined algorithms. This parameter is influenced by several factors: CPU cores sharing one NUMA memory node, cache size, memory hierarchy, and memory bandwidth.

3.4 Kernel-Assisted MPI Collective Communication Framework

We implemented the KNEM-based collectives as a new component named the KNEM collective in Open MPI. Open MPI is based on a flexible component architecture, collective algorithms are placed under the *COLL* framework (as depicted in Figure 3.4). Multiple collective components are available (*Tuned*, *Basic*, *SM*, and our proposed *KNEM*), and can be selected at runtime. In Open MPI's component

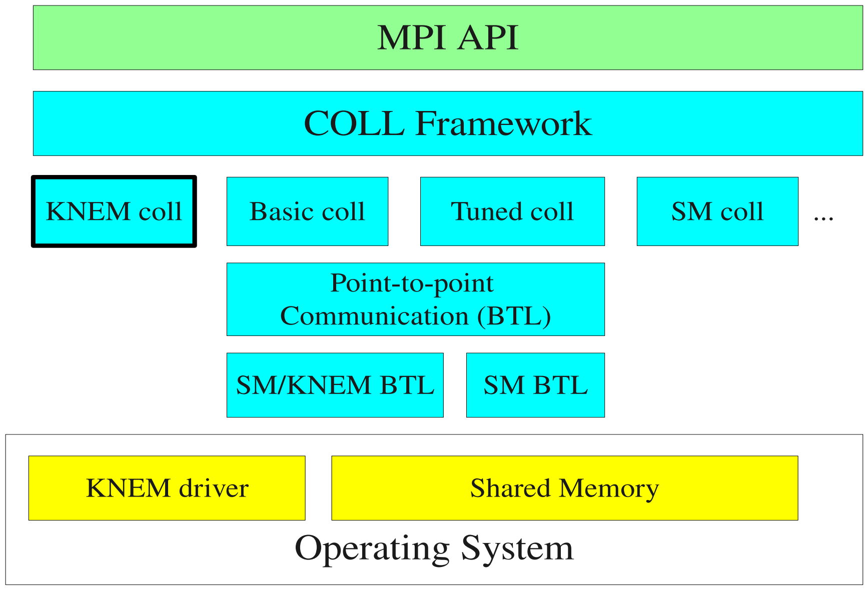


Figure 3.4: Open MPI collective communication framework

architecture, different collective components can use different underneath point-to-point communication components (Byte Transfer Layer, BTL) [56]. By default, the *Tuned* collective component and the SM BTL component are teamed to provide for MPI collective operations, resulting in the default setup to use the best collective algorithms with the copy-in/copy-out point-to-point transport. In previous works the *SM/KNEM BTL* were introduced. This BTL enhances shared-memory point-to-point operations by using the KNEM kernel memory copy for large messages. While this is not a default setup, the *Tuned* collective component can be teamed with this BTL to benefit from KNEM kernel memory copy to some extent for large messages.

Unlike other collective components, the KNEM collective component uses the shared memory BTL only as an out of band channel for synchronization or delivering KNEM's "cookie" data (128bits). All data movements are handled inside the collective component by resorting to direct calls to the KNEM memory copy module. While the KNEM kernel module is used directly, the collective algorithms themselves remain implemented in user-space. Direct calls to KNEM within the collective component are required to have enough flexibility to express the new algorithms intended to avoid unnecessary buffer registrations, handle directional copies, and fragment the messages to establish a pipeline suitable for the most complex memory hierarchies. The KNEM function interfaces (APIs) (see Section 3.2) are used to create memory regions dynamically and read or write all or part of them multiple times, whenever needed. However, trapping into kernel mode has a non-negligible overhead (about 100 ns on modern processors) when delivering small messages. So, we only consider KNEM for optimizing collectives for intermediate and large messages (larger than 16KBytes). We started by adopting the most useful algorithms first (Gather(v), Scatter(v), AlltoAll(v), AllGather(v) and Broadcast, v stands for vector variants of corresponding collective operations). For smaller messages, or unimplemented collective calls, the operation is delegated to the original Open MPI component.

3.5 Rooted Operations: Broadcast, Scatter and Gather

The implementation of KNEM Broadcast is a straightforward adoption of the KNEM point-to-point model: 1) The root process declares a send buffer to KNEM and gets the corresponding cookie in return. 2) It passes the cookie to all non-root processes in the communicator through several out-of-band transfers (via Open MPI SM BTL). 3) Each receiver process passes the incoming cookie to the KNEM kernel device along with its receive buffer. 4) KNEM triggers a memory copy from the send buffer to receive buffer within the kernel. The copy is performed by each receiver core in parallel. 5) Each receiver process sends back a synchronization message to root process after the completion of KNEM copy. 6) After the root process receives all synchronization messages from non-root processes, it deregisters the send buffer from the KNEM driver. The KNEM Broadcast also features a hierarchical pipelining algorithm (as sketched in Section 3.3), that can be turned on or off depending on the properties of the hardware. The topology mapping here is static and manual in this chapter, but there will be a dynamic approach presented in Chapter 4.

The KNEM Scatter implementation is overall similar to the KNEM Broadcast, except that each non-root process reads only parts of the root buffer. Their starting offset is calculated from their ranks and the data type.

The KNEM Gather consists of the opposite communication pattern of Scatter. Therefore it can benefit from the direction control of the new KNEM version to declare the root process memory as a write buffer. Unlike the regular Gather, all non-root processes can write simultaneously to that buffer.

3.6 AllGather and Alltoall

The KNEM AllGather is an assembly of a gather followed by a broadcast. During the first step, all processes do a KNEM Gather operation to rank 0. In the second stage, the root process performs a KNEM Broadcast. This ad-hoc method is pretty good on SMP compute nodes because it can capitalize on the improvements achieved in the KNEM Broadcast and Gather. However this method is far from being optimal on a large NUMA compute node, because it puts the memory controller of the core hosting the root process under a high pressure. The overall bandwidth of this ad-hoc Allgather operation is restricted by the memory bandwidth of the NUMA memory node hosting the root process. Chapter 4 presents a ring-style algorithm for the KNEM Allgather operation on large NUMA compute nodes.

Our AlltoAll algorithm focuses on avoiding bandwidth sharing, by rotating the access pattern so that at any instant, a core is sending and receiving exactly one fragment of data. First, each process declares its send buffer to KNEM and gets back the corresponding cookie. Those cookies are exchanged by doing an out of band AllGather operation based on shared memory (not KNEM AllGather); it is necessary to pre-allocate an integer array the size of the communicator to store cookies from all other processes. A loop of KNEM copies is performed to fetch the corresponding

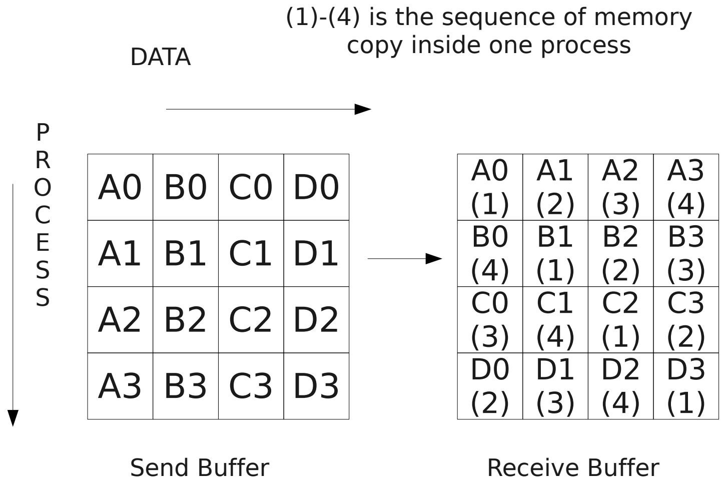

(1)-(4) is the sequence of memory copy inside one process
DATA
PROCESS
A0 B0 C0 D0
A1 B1 C1 D1
A2 B2 C2 D2
A3 B3 C3 D3
A0 (1) A1 (2) A2 (3) A3 (4)
B0 (4) B1 (1) B2 (2) B3 (3)
C0 (3) C1 (4) C2 (1) C3 (2)
D0 (2) D1 (3) D2 (4) D3 (1)
Send Buffer
Receive Buffer

Figure 3.5: An example of a copy sequence for KNEM AlltoAll on four processors.

messages from other nodes (receiver-reading). Each process offsets the starting point of this loop in a round-robin manner. Finally, each process deregisters memory buffer from the KNEM driver after a barrier operation. Figure 3.5 gives an example of this communication scheduling; as one can see, for every step of the algorithm (marked between parenthesis on the right buffer), the memory belonging to a particular sender is accessed only once, and the workload is evenly spread over the entire duration of the collective.

3.7 Experimental Evaluation

3.7.1 Experimental Conditions

Our experimental platform is composed of two multicore/many-core machines that cover the spectrum of typical current commodity high performance compute nodes, but also expected designs for the upcoming years. They feature Intel and AMD processors, and represent a wide variety of setups, from SMP to massive NUMA machines. Because a vector version of collective operations has a similar implementation with the corresponding collective operations, we just select one operation from each type to present performance improvement to save space. Broadcast, Gather, Scatter, Alltoallv, and Allgather are selected as examples to present a performance comparison with other state-of-the-art implementations.

Zoot is a 16 core machine with 32GB of memory. The system has four sockets with a quad-core 2.40 GHz Intel Xeon Tigerton E7340 featuring 4 MB L2 caches shared between pairs of cores. A single memory controller in the north-bridge chipset connects all the sockets with the global shared memory. Because all CPU cores share a single memory control, it is an SMP-style compute node.

IG is a 48 cores machine with 128GB of memory. The system is composed of 8 sockets with a six-core 2.8 GHz AMD Opteron 8439 SE, 5 MB L3 caches and 16 GB memory per NUMA node. The sockets are further divided as two sets of 4 sockets on two separate boards connected by a low performance interlink. IG is a large NUMA compute node. The architecture is depicted on Figure 3.2.

Software setup includes KNEM version 0.9.5 [6]. The Intel MPI benchmark suite IMB-3.2 [5] was used to assert the difference between the collective components with the *off-cache* option enabled to avoid cache reuse. Open MPI version 1.7a1 and MPICH2-1.3.1, both properly tuned, are compared to our approach. Tuning parameters between all components based on Open MPI are identical, unless stated. On a particular machine, the mapping between physical cores and MPI processes is identical, regardless of the MPI implementation used. Due to the large number of combinations, we restrict the discussion only to the most meaningful algorithms, but still cover the entire spectrum of collective patterns.

Because our own component (referred to as KNEM-Coll from now on) is inside Open MPI, we undergo a deeper comparison with its default collective component: the *Tuned* component. It does not use kernel assisted copies by default, but uses the SM BTL as underneath point-to-point communication that relies on copy-in/copy-out; results obtained with this setup are called *Tuned-SM*. To clearly assert what the extra benefits of our approach are, we also present the results obtained when simply using the *Tuned* component on top of KNEM point-to-point messages (*Tuned-KNEM*). *MPICH2-SM* and *MPICH2-KNEM* are similar to *Tuned-SM* and *Tuned-KNEM*, with respectively shared-memory or KNEM as the underneath point-to-point communication transport.

3.7.2 Hierarchical effect and tuning the pipeline size

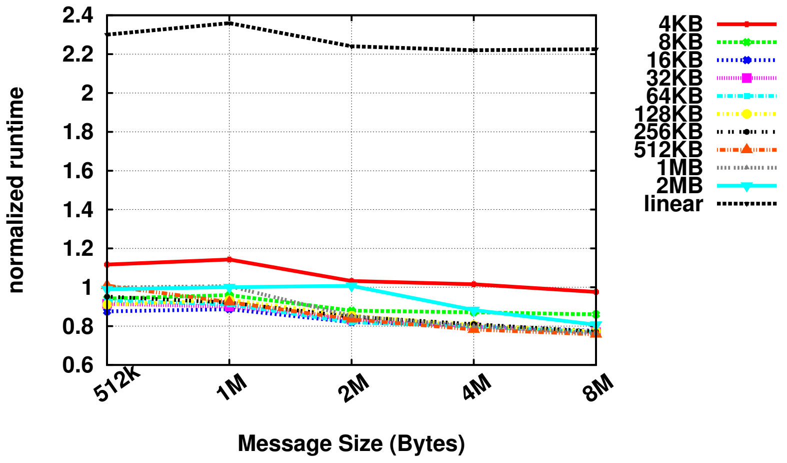


Figure 3.6: Performance comparison between linear KNEM Broadcast, hierarchical KNEM without pipeline, and different pipeline sizes in the hierarchical pipelined KNEM Broadcast on the IG platform. Results are normalized to the runtime of hierarchical KNEM Broadcast without pipeline (lower is better).

Figure 3.6 presents the effect of the pipeline size on the hierarchical pipelined KNEM Broadcast on the large NUMA machine IG. The pipeline sizes range from 4KB to 2MB. The execution time of different pipeline sizes is normalized to the execution time of a hierarchical KNEM Broadcast without pipeline. Compared with the linear KNEM broadcast, the hierarchical approach itself contributes a 2.2× to 2.4× speedup on this large NUMA node. And the pipelining provides hierarchical KNEM Broadcast extra up to 1.25× speedup. One can see that, except for using too small pipeline size (4KB), hierarchical KNEM Broadcast can always get significant benefits from pipelining.

Selecting a suitable pipeline size can be a challenging problem for the hierarchical pipelined strategies. A too small pipeline size induces more synchronization between each segment and makes the transfer efficiency suffer due to KNEM copy startup overhead when delivering small messages (4KB and 8KB in Fig 3.6), while a too large pipeline size (2MB in Fig 3.6) leads to a long initialization time for the pipeline algorithm to take effect. One can see that the best pipeline size is 16KB for intermediate message sizes (smaller than 2MB), and 512KB for large message sizes. In the rest of this chapter, we settled the pipeline size according to this tuning on IG: 16KB for intermediate message size and 512KB for large message size.

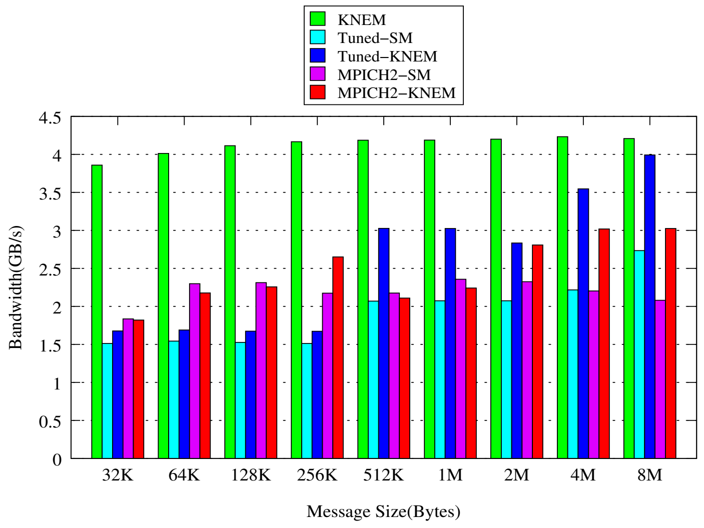
3.7.3 Rooted Operations

Figure 3.7 shows the performance comparison of the Broadcast implementations on all platforms. The KNEM Broadcast outperforms other collective components in all cases. Compared with Open MPI's best collective component, the KNEM Broadcast can provide a speedup of about 1-2.5× on Zoot and 1.5-2.1× on IG.

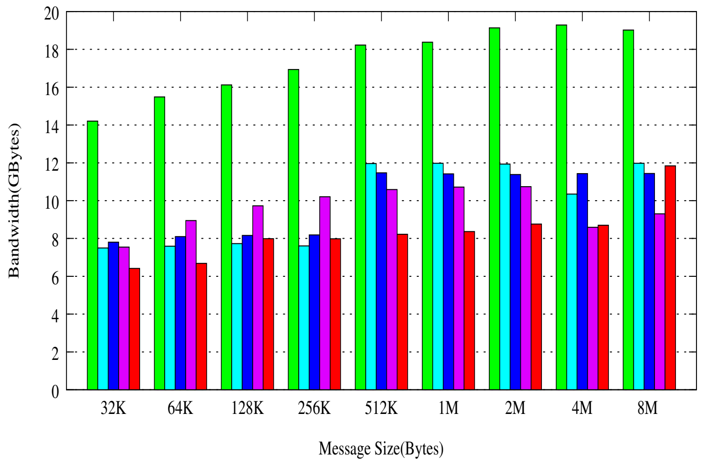
Figure 3.8 presents the performance comparison of the Gather operation. The KNEM Gather tremendously outperforms all other components in all cases. Compared with the best Gather in Open MPI and MPICH2, the maximum speedup, thanks to the KNEM collective component, is 3.1× on Zoot, and 3.2× on IG.

Figure 3.9 shows the performance comparison of the Scatter operation. KNEM Gather and Scatter operations are very similar, except from the different copy directions: sender-writing for Gather and receiver-reading for Scatter. Consequently, those two operations exhibit very similar performance profiles. Compared with Open MPI's best *Tuned* Scatter implementation, the maximum speedup of KNEM Scatter is about 3.2× on Zoot, and 4.3× on IG.

The KNEM collective component has a huge performance speedup in these "rooted" collective operations, thanks to unleashing parallel access to the buffer of the root process. Compared with the approach of indirectly using KNEM copy as



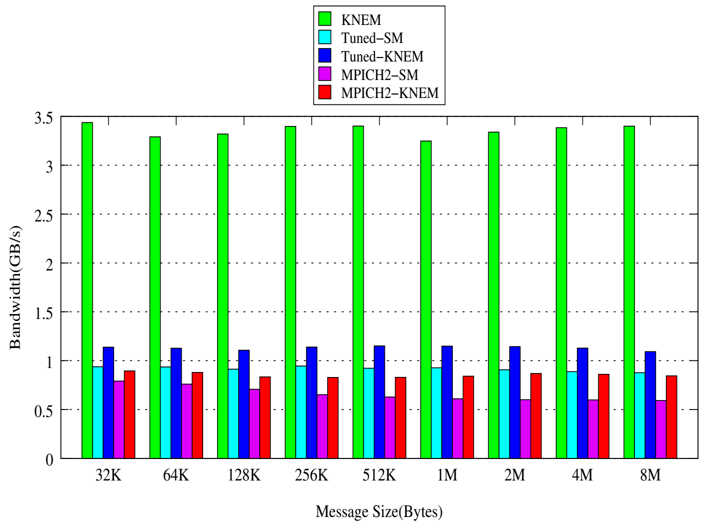
(a) on Zoot



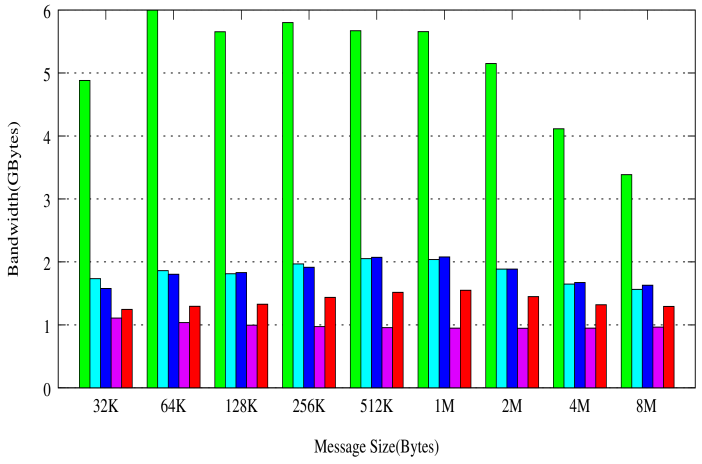
(b) on IG

Figure 3.7: Aggregate Bandwidth Comparison of Broadcast operations between Open MPI Tuned components, MPICH2, and the KNEM collective component.

underneath communication (Tuned-KNEM), the KNEM collective component can provide more reliable improvement in all cases, benefiting from KNEM drivers' new features we mentioned in Section 3.2.



(a) on Zoot

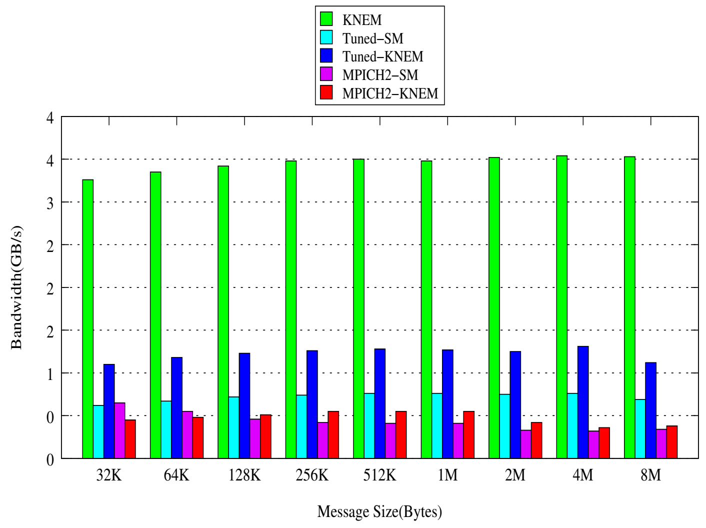


(b) on IG

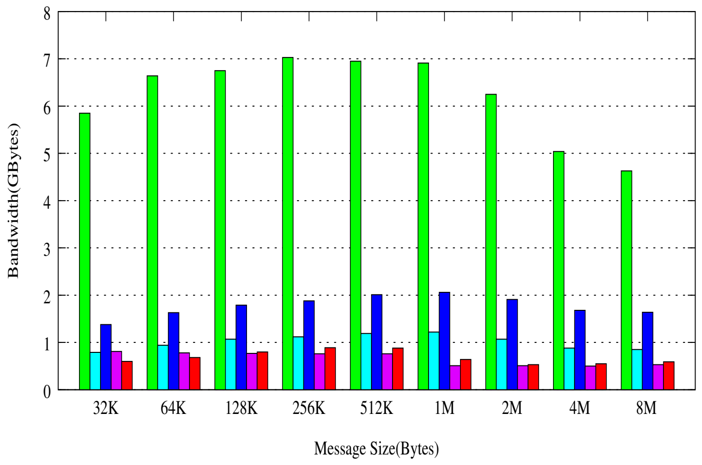
Figure 3.8: Aggregate Bandwidth Comparison of Gather operations between Open MPI Tuned components, MPICH2, and the KNEM collective component.

3.7.4 Many-to-many Operations

Figure 3.10 shows the performance comparison for the AlltoAllv collective operation. The performance of these many-to-many collective operations is restricted by the press over memory buses due to delivering too many messages simultaneously. As a consequence, the relative performance benefits are smaller when compared to Tuned-KNEM than for the one-to-many or many-to-one rooted operations presented in



(a) on Zoot

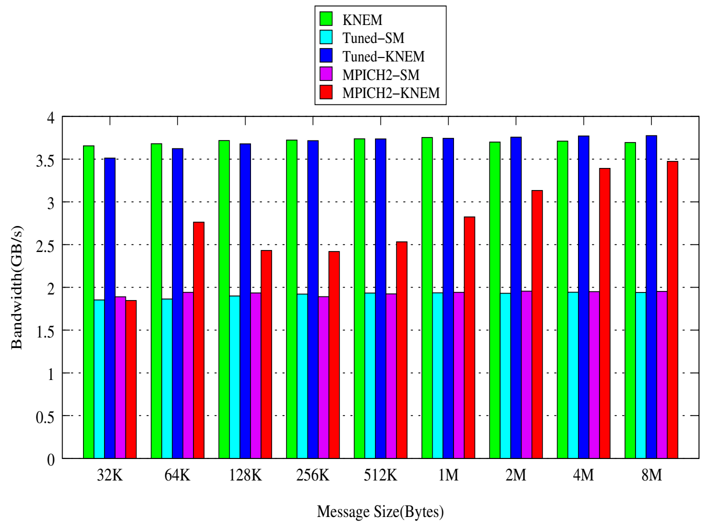


(b) on IG

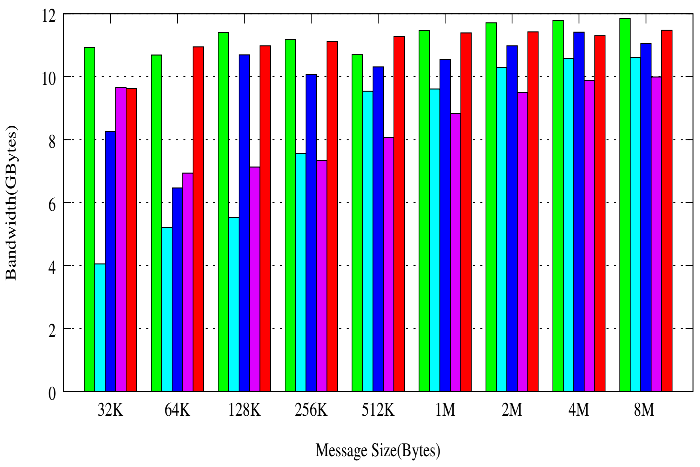
Figure 3.9: Aggregate Bandwidth Comparison of Scatter operations between Open MPI (Tuned-KNEM), MPICH2, and KNEM collective modules.

the previous paragraphs. However, compared with Tuned-SM based on the shared memory approach, the KNEM AlltoAllv can still show significant improvement, with a maximum speedup of 2× on Zoot, and 2.7× on IG.

Finally, Figure 3.11 presents the performance of the collective components on the AllGather operation. On the SMP machine (Zoot), the KNEM AllGather has the best performance among all collective components. On the large NUMA node (IG), Tuned-KNEM performs better than KNEM AllGather by up to 25%. The



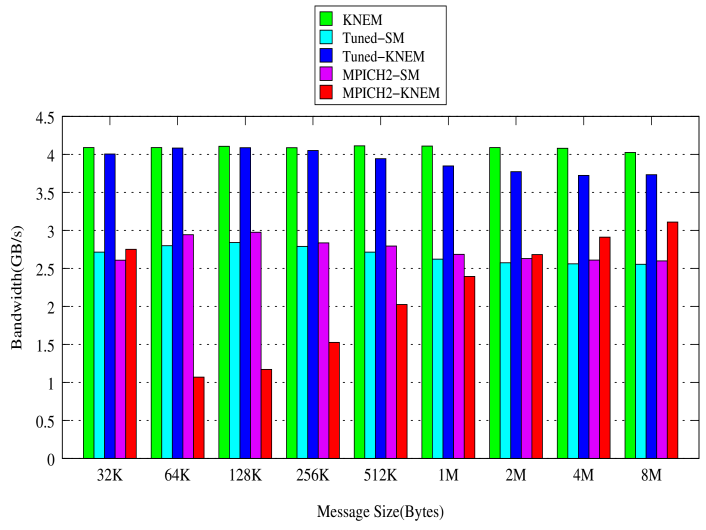
(a) on Zoot



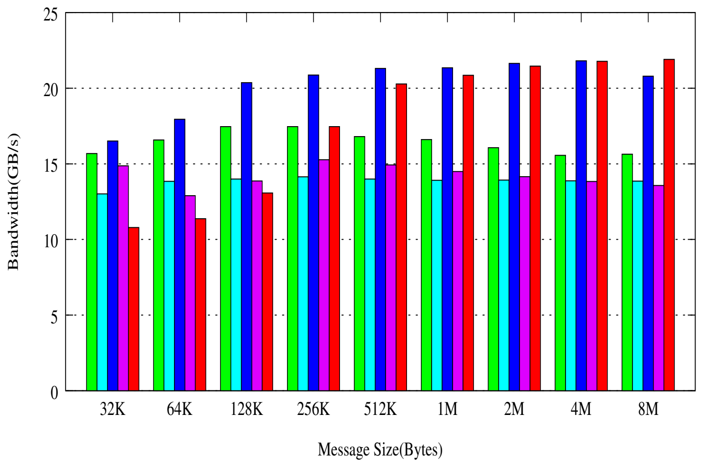
(b) on IG

Figure 3.10: Aggregate Bandwidth comparison of AlltoAllv operations between Open MPI Tuned components, MPICH2, and the KNEM collective component.

loss of KNEM AllGather's performance on large NUMA nodes lies in the KNEM AllGather operation not being optimized as explained in Section 3.6. The operation is split into two separated stages. Although the KNEM Gather and Broadcast are optimized in each stage, overlapping between these two stages is eliminated by this simple concatenation of the KNEM Gather and Broadcast algorithms. And the selected root process forms a single point in the KNEM AllGather implementation, forcing the throughput of the AllGather operation, which is restricted by the limited



(a) on Zoot



(b) on IG

Figure 3.11: Aggregate Bandwidth Comparison of AllGather operations between Open MPI Tuned components, MPICH2, and the KNEM collective component.

memory bandwidth of the NUMA node owning the root process. This is also the reason why the KNEM AllGather's performance suffers more on the large NUMA nodes than on SMP or small NUMA nodes. However, even on a large NUMA node IG, the KNEM AllGather still performs better than Tuned-SM and MPICH2-SM, which are based on the shared memory approach. The implementation of these two many-to-many collective operations (e.g., AlltoAllv and AllGather) benefits greatly from the adoption of KNEM copy, thanks to reducing memory copies and cache pollution in these communication-intensive operations. But the kernel-assisted memory copy technique itself can't solve all difficult cases, e.g., Allgather on large NUMA compute nodes here. In addition to NUMA-aware kernel-assisted Broadcast, other collective operations also require combining kernel-assisted memory copy and hardware architecture to reach potential peak performance. Chapter 4 will present how these techniques are used together under a distance-aware framework.

3.7.5 Application Performance

ASP

To evaluate the impact of the improvement due to using the KNEM collective operations on real application performance, we use the ASP [51] application, a parallel implementation of the Floyd-Warshall algorithm used to solve the all-pairs-shortest-path problem. The main MPI collective operation used in this application is MPI_Bcast. We tested this application on two machines: Zoot and IG, the two extreme platforms regarding the degree of complexity of the core hierarchy. The problem is scaled to match the available memory; the matrix size is 16384^2 on Zoot and 32768^2 on IG (32bits integers). Matrices are distributed by rows across all the available cores. The MPI_Bcast operation is called 16384 times (64 KB message) on Zoot and 32768 times (128 KB message) on IG. The KNEM collective component uses the linear KNEM algorithm on Zoot and the hierarchical pipelined algorithm on IG. All tests use the same mapping between cores and processes.

Table 3.1: ASP application execution time breakdown and speedup from using the KNEM collective.

	Zoot		IG	
	Bcast	Total	Bcast	Total
Open MPI	405.7s	2891.2s	550.2s	6650.9s
MPICH2	152.3	2640.4s	293.9s	6413.8s
KNEM Coll	26.8s	2508.4s	198s	6288.1s
Improvement	82.4%	5.2%	33%	2%

Table 3.1 presents the application execution time of ASP when using different collective components. The improvement is the relative difference between the best performing MPI library (between Open MPI and MPICH2) and our KNEM collective components. By using the KNEM collective components, the application can see a significant improvement in the time it spends doing Broadcast operations, with the improvement of 82% on Zoot and 33% on IG. One can notice that the performance improvement of the Broadcast only on the SMP machine is even more pronounced than for the synthetic benchmark, because unlike the synthetic benchmark, the application does not systematically invalidate the cache before performing the operation. As a consequence of the shorter time spent in the collective operations, the overall application runtime is improved when compared to other optimized collective components, with an improvement of 5% on Zoot and 2% on IG.

CPMD

The CPMD code is a parallel plane wave/pseudo-potential implementation of density functional theory, particularly designed for ab-initio molecular dynamics [32]. Its MPI version was developed around a core routine, involving both point-to-point and collective communication, and can be considered as a communication-intensive application. Using a lightweight MPI profiling software (mpiP) [60], we investigate the communication overhead distribution in this application, including the percentage of MPI runtime in the application runtime, the percentage of each MPI call runtime in the whole communication time, and the message size distribution. Our experimental platform is still the 48 core AMD NUMA machine (IG).

Inside Open MPI, two different communication setups are compared: the SM setup uses the Open MPI *Tuned* collective component [20] and the SM point-to-point Byte Transfer Layer (BTL); the KNEM setup uses the KNEM collective component and the SM/KNEM BTL [42]. Because the KNEM collective component implemented a subset of MPI collective operations: Broadcast, Gather(v), Scatter(v), Allgather(v), and Alltoall(v), operations not implemented in the KNEM collective component such as Reduce, Allreduce, etc. will use Open MPI's Basic collective component. The mapping between physical cores and MPI processes is identical for both setups, regardless of the underlying communication components.

The CPMD software (version 3.13.2) [32] is configured as 'LINUXMPI' with the BLAS/LAPACK libraries (LAPACK 3.3.0). We selected one simulation from CPMD's vibrational analysis tests: methan-fd-nosymm.inp.

	Total Application time(sec)	MPI time(sec)
KNEM-enabled	276	74.4
SM-enabled	423	229

Table 3.2: Total application time and MPI time for CPMD's methan-fd-nosymm test between Open MPI's KNEM mode and SM mode, with 48 MPI processes on IG's 48 cores.

Table 3.2 compares the execution time, broken down into compute time and communication time, of the CPMD application on IG, between the two communication modes (KNEM and SM, differing in their use of kernel assisted memory copies). As expected, the computation execution time remains generally constant when changing the communication mode (201s versus 194s). The major performance difference between the two setups lies in the communication overhead (MPI time), which occupies 26.9% of the overall application runtime for the KNEM-enabled mode, while it rises to 54.1% when using the regular SM communication mode. The CPMD application makes extensive use of many-to-many collective communication, which enjoy around 3.4× speedup when using the KNEM-enabled approach, translating into a 1.5× application speedup in the methan-fd-nosymm test case.

Table 3.3 presents the accumulated time, over all processes, spent in the five most time consuming MPI functions. In both cases, using KNEM or SM, the AlltoAll operation takes more than 90% of the MPI execution time. However, compared with SM-based communication, the KNEM version reduces the Alltoall cost from 10, 000s to about 2, 900s. Based on the statistics gathered using mpiP, the average message size for each AlltoAll operation is 24KBytes, a size in the range where KNEM is beneficial

KNEM-enabled			SM-enabled		
Call	Time(millisec)	MPI%	Call	Time(millisec)	MPI%
Alltoall	2.88e+06	90.32	Alltoall	1.02e+07	97.71
Bcast	1.15e+05	3.59	Bcast	1.08e+05	1.03
Allreduce	1.12e+05	3.52	Allreduce	1.05e+05	1.00
Barrier	7.01e+04	2.20	Allreduce	8.6e+03	0.08
Allreduce	7.49e+03	0.23	Recv	6.17e+03	0.06

Table 3.3: Sum of all processes' execution time for the 5 most used MPI functions in CPMD's methan-fd-nosymm using shared memory and KNEM (48 processes on IG's 48 cores).

to collective operation performance (bigger than 16KB). About 3.4× speedup from adopting the KNEM Alltoall operations is greater than the speedup of a similar operation: Alltoallv shown in synthetic benchmarks (Alltoall and Alltoallv have the same speedup in IMB benchmarks, about 2-2.5× in aggregate bandwidth on IG) like Figure 3.10(b). The extra speedup comes from the cache-friendly design in the KNEM collective operations by eliminating intermediate buffers. Real applications often have more cache reuse cases than cache-off synthetic benchmarks, like CPMD. This is also a direct reason why our KNEM collective operations show more speedup in real applications than in synthetic benchmarks. Here, Allreduce in the KNEM-enabled setup is worse than in the SM-enabled setup (1.12e5 vs 1.05e5), because Allreduce in the KNEM setup actually triggers the Open MPI's Basic collective component, which is a simple and basic implementation of collective operations without any optimization. It is not a surprise for a *Tuned* collective operation to outperform a Basic collective operation.

Figure 3.12 shows a strong scaling performance of CPMD's methan-fd-nosymm tests, using the KNEM-enabled and SM communication modes. In this experiment, the process i is bound to core i, for all modes. The KNEM-enabled MPI communication outperforms the SM MPI communications, regardless of processes in use and the number of NUMA nodes. The CPMD application benefits from a better scalability, when increasing the number of cores, with the KNEM-enabled

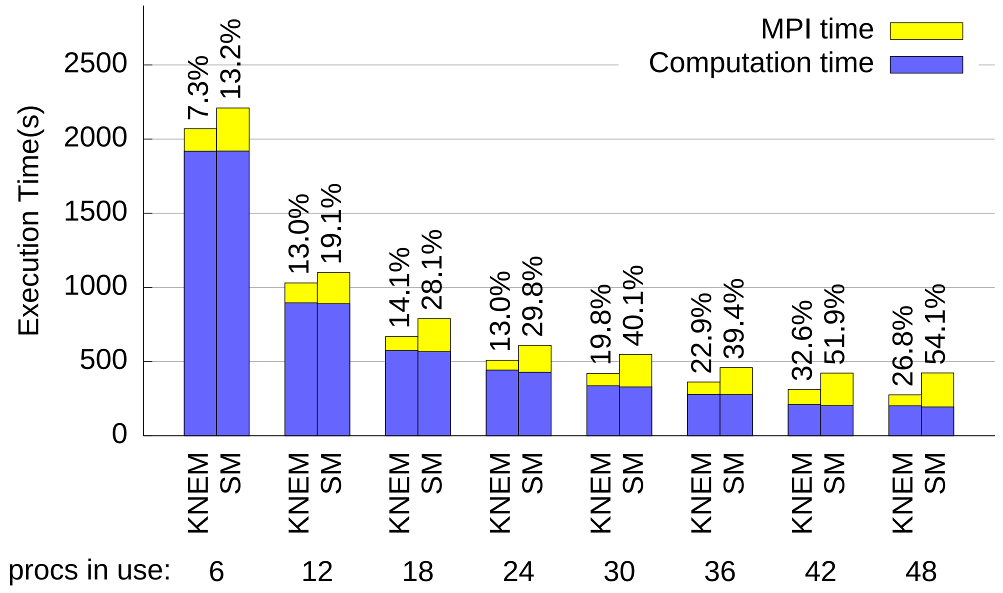


Figure 3.12: Strong Scaling for CPMD's methan-fd-nosymm test over KNEM and shared memory. Processes are bound to IG's cores in a compact way (rank i is bound to core i).

MPI operations. The SM communications do not permit the application to scale to more than 24 processes (the limit where all processors are on the same motherboard, in IG), because the SM communications are oblivious to the underlying hardware topology. On the other hand, the KNEM-enabled operations enable the application to benefit from the expected scalability for the CPMD application, even though the NUMA topology is extremely challenging on this platform.

3.8 Conclusion

Modern HPC platforms are trending towards more complex hardware in an effort to avoid the memory wall. Unfortunately, such platforms are increasing software complexity; software must now be aware of both process and data locality within a single server, and ensure not to create memory subsystem hotspots when communicating between processors (regardless of whether the communication is explicit or implicit).

Indeed, as HPC vendors strive to create new platforms with higher peak (hardware) performance metrics, even minor differences between platforms can translate to major software porting efforts to maintain peak overall performance.

An MPI implementation's goal is to hide as much of this underlying networking and data locality complexity from applications as possible; if performance portability can be preserved in most core MPI message passing functions, application developers can concentrate on their high-level goals rather than the underlying hardware.

In this chapter, we showed that an MPI implementation can successfully take advantage of new features in the kernel, namely kernel assisted memory copies between process spaces, to greatly improve the performance of shared memory collective communication while successfully hiding the complexity of the NUMA characteristics of modern many-core nodes. Three challenges specific to multicore or many-core systems have been addressed to meet that goal: NUMA complexity, cache pollution, and collective algorithm sequentialization.

The main contributions of this Chapter are: 1) extend kernel-assisted memory copy operations to include direction and granularity control, thereby giving more flexibility and data movement control to the collective component implementation, and 2) embed the use of kernel-assisted copies directly into collective algorithms rather than relying on kernel-assisted speedups from MPI point-to-point-based primitives. 3) Our KNEM collective component can also adjust the stream direction of kernel-assisted memory copies according to communication pattern: many-to-one or one-to-many to offload memory copies to non-root processes in order to parallelize operations with multiple receivers. 4) Our KNEM component is also cache-friendly thanks to eliminating intermediate buffers. This feather leads to more speedup for real applications with cache reuse. Finally, the pipelining algorithm strategically overlaps the latency incurred by transferring first to the upper levels of the NUMA hierarchies with data transfers to the leaf nodes in the hierarchical multi-level collective topology.

Experimental results show that our approach outperforms all other types of optimizations on many-core systems, including the Open MPI *Tuned* component (even though *Tuned* benefits from KNEM-enabled point-to-point operations). Our KNEM-enabled MPI communication can increase application performance and expose a better scalability than the classical shared memory approach when integrating more CPU cores and memory into compute nodes.

There are more optimizations that can be applied to collective algorithms inside shared memory compute nodes. For small messages, the cost of switching to kernel mode to use kernel-assisted copies cannot be recovered by the performance advantage of the copies. Clearly, small messages must still rely on legacy collective implementations. In the near future, we plan to integrate our work with the standard shared memory collective component of Open MPI to enable automatic selection of the best implementation and algorithms (legacy or KNEM-assisted), based on the architecture, processes placement, and message size.

Chapter 4

Distance-aware Framework

4.1 Introduction

Chapter 3 has shown huge performance improvement via building collective algorithms directly based on kernel-assisted memory copy. But kernel-assisted memory copy can not tackle all cases by itself, and it must be combined with adaptive topology-aware techniques: dynamically mapping collective topologies onto underlying hardware topologies. The KNEM NUMA-aware Broadcast or Allgather Ad-hoc algorithms have presented examples of correctly and wrongly mapped collective topologies onto large NUMA compute nodes, respectively. Benefiting from a topology-aware design, memory access workloads in the NUMA-aware Broadcast are evenly distributed across all available NUMA memory nodes. On the other hand, the aggregate bandwidth of the Ad-hoc Allgather is restricted by single NUMA memory nodes due to ignoring NUMA architecture in its ad-hoc algorithm. All evidence points to the importance of dynamically and automatically mapping MPI collective topologies onto underlying hardware topologies. This chapter outlines a distance-aware framework that was developed to provide such a mechanism to make collective algorithms be aware of not just underlying hardware architecture, but also process placement and run-time communicator composition.

Another motivation of the distance-aware framework arises from irregular process-core mappings generated by an intelligent process placement module, e.g., MPIPP [16], TreeMatch [33], etc. Multi- or many-core clusters are currently the most popular platform in high performance computing. With the increasing number of computing resources and memory hierarchies integrated into a single compute node, the distribution of MPI processes inside a node become critical in order to fully exploit the node's capabilities. Moreover, even if not yet standardized by the MPI Forum, most of the MPI libraries provide proprietary interfaces to bind MPI processes to specific cores. A lot of research has been done to adjust the process layout based on an application's communication pattern and underlying hardware architecture. MPIPP [16] provided a profile-guided approach to automatically find the optimal mapping between MPI processes and resources to minimize the cost of point-to-point communications for arbitrary message passing applications. Emmanuel Jeannot *et al.* proposed a near-optimal process placement algorithm called "tree match" that maps processes to CPU cores [33]. For example, a profiling file shows point-to-point communication between pairs of processes: (0, 1), (2, 4), (3, 6) and (5, 7) occupies the most percentage in MPI communication time. The process placement module will arrange these pairs of processes as closely as possible with respect to physical distance, as depicted in Figure 4.1, where pairs of processes are bound to two cores on the same socket. As a bridge between MPI libraries and applications, these intelligent process placement libraries help users find an optimal process placement in order to reduce the communication cost of the whole application.

Although these intelligent process placements have the potential to significantly decrease the overall communication time, their methodology is based on a pure point-to-point communication pattern, ignoring the different communication topologies used by MPI collective algorithms. As most collective communication exhibits specific communication patterns which are allowed to adapt to the underlying architectural features (split binary tree, binomial tree, chain, etc.), there is a mismatch between the MPI internal collective topology (which is usually created based on the MPI

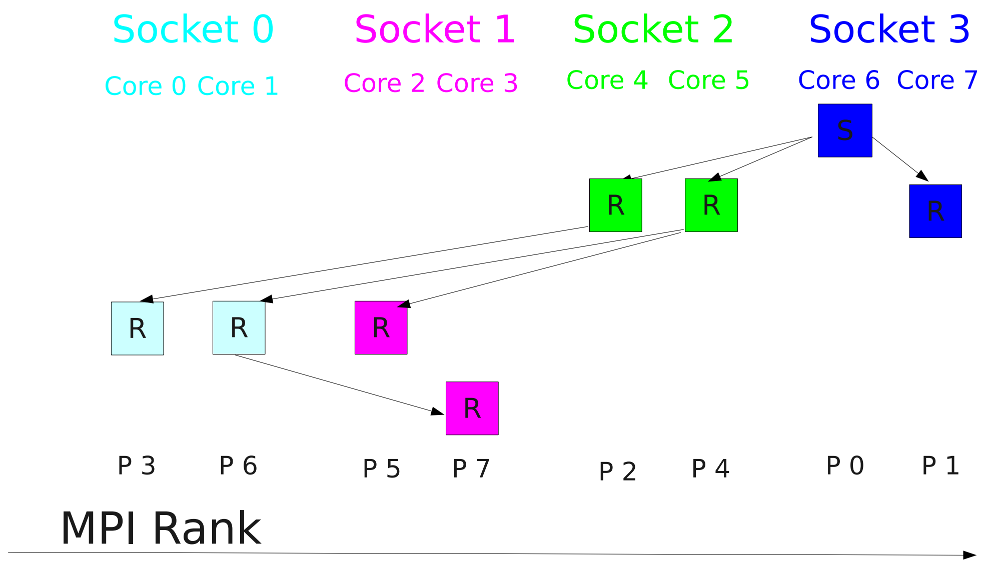


Figure 4.1: An example of an in-order binomial broadcast tree spanning over 8 processes on quad-socket dual-core nodes, which are placed following applications' communication pattern.

process ranks) and the external process placement decision. Under the same process placement in Figure 4.1, let's assume a Broadcast operation with these 8 processes on 8 cores in the application. Based on MPI ranks, a binomial broadcast tree is constructed as in Figure 4.1. Every edge along the critical path ($P0 \rightarrow P4 \rightarrow P6 \rightarrow P7$) in the broadcast tree crosses the longest physical distance, a bus connecting each socket. Obviously this binomial broadcast tree will lose its advantage and efficiency due to a mismatch between broadcast topology, underlying hardware architecture, and process placement.

The topology of MPI collective communication varies during the life cycle of the application, because the communicator in the application can change, and the order of the processes participating in each collective communication is dynamic. As the process placement is fixed for an application, even with the help of the previously described tools, without precise knowledge of the collective communication algorithm to be used for a specific collective in a communicator, globally optimizing the parallel applications become an impossible mission. However, if internally the MPI library takes advantage of the architectural features of the hardware environment, and adapts its own communication algorithms to maximize the hardware capabilities, solving the irregular process placement problem becomes simpler.

As an extension of our work published in IEEE Cluster [43], this chapter introduces a general framework for MPI implementations to detect, express, and take advantage of the run-time process distance. Based on run-time process distance information in the context of each compute node, the MPI library constructs an adaptive communication topology for each collective operation, and this topology reflects the underlying hardware architecture. This distance-aware collective communication always provides optimal performance regardless of process placement of the members participating in the communicator. This automatic approach at the MPI level provides further optimization and complements the intelligent process placement approach without user intervention.

The rest of this chapter is organized as follows: Section 4.2 provides some background about collective logical topology and topology-aware MPI communication. Section 4.3 formulates and outlines the extent of the problem between process placement, underlying architecture and the run-time communicator. Then, Section 4.4 describes our framework designed to combine process distance information with collective communication topologies, and then two distance-aware adaptive collective operations (Broadcast and Allgather) are implemented in Open MPI's KNEM collective communication component as examples. A performance study is presented in the Section 4.5, substantiating the benefits of our approach when compared to the Open MPI's default *Tuned* collective component. Finally, Section 4.6 concludes the chapter with a discussion of the results.

4.2 Related Work

As Chapter 2 describes, most MPI libraries, like MPICH2 [29] and Open MPI [23], select *Tuned*-style collective as default collective components. In the *Tuned* collective component, a run-time selection framework is used to determine the optimal collective algorithms based on message and communicator size. These algorithms actually use "fixed" topologies decided by pre-defined fan-out and communicator size. Additionally, most of these fixed topologies are built based on MPI's logical ranks, and are totally agnostic of process placement. It is impossible for these algorithms to provide stable and optimal performance under all circumstances, any process placement, any underlying hardware architecture, and dynamic communicators.

Jesper Larsson Traff first describes the remapping to MPI libraries according to underneath hierarchical architecture such as SMP clusters [58]. The proposed methodology, based on graph partitioning, generates a multi-level hierarchical view of the hardware environment.

Similar approaches found their way into MPI implementations a few years later when the first platform-specific topology-aware vendor MPI libraries appeared on the market, e.g. MPI/SX [58] and HP-MPI [55].

Several research teams have investigated topology-aware collective algorithms on specific platforms: cluster of clusters, Grids and InfiniBand clusters. Thilo Kielmann *et al.* proposed MagPIe [38] which is a hierarchical MPI collective communication operation for a cluster of clusters. A similar topology-aware multi-level approach is introduced in [37] to tackle collective operations in Grids. D.K. Panda *et al.* proposed SMP-aware [44] or topology-aware [35] collective operations over InfiniBand clusters. These topology-aware algorithms achieved a significant performance increase using knowledge about the underlying networks or hardware architecture. However, most of these algorithms only tackle special cases, and some of them are difficult to port in general MPI libraries due to a lack of a generalized MPI framework to express and detect distances between processes. With more hierarchies introduced into nodes or clusters, there is an urgent demand for a general framework for MPI to detect and express process layouts based on process distance.

We also proposed a framework to select the optimal communication parameters according to run-time process distance between peers for MPI intra-node point-to-point communication in [41]. Compared with point-to-point communication, collective communication is more demanding about the physical topology information, and more sensitive about memory hierarchies, and distance between processes. With the knowledge of such hardware information, efficient scheduling can be implemented in specific collective algorithms to balance memory accesses across memory controllers and maximize the overall collective communication bandwidth.

The Portable Hardware Locality (hwloc) [12] is a project providing a much needed portable abstraction of hierarchical topology of modern CPUs, including NUMA nodes, sockets, caches, cores, and simultaneous multithreading. The hwloc software package has been widely adopted in state-of-the-art MPI libraries such as Open MPI, MPICH2, and MVAPICH2. Our run-time process distance detection framework is also based on the information collected by hwloc.

4.3 Mismatch Problem

Nowadays most of the MPI implementations recognize the process placement problem. In order to alleviate the issue, they propose proprietary interfaces to bind processes to resources. For example, Hydra, the process manager in MPICH2, provides users with options like ‘user’, ‘rr’, ‘cpu’, and ‘cache’. “-binding user” is a user-defined binding. “-binding rr” is based on a round-robin mechanism using the operating system (OS) processor IDs. MPICH2 can also provide some cpu-aware or cache-aware allocation strategies such as “-binding cpu” that packs processes as closely as possible to each other with respect to CPU cores, and “-binding cache” that does it respectfully to the cache layout.

Figure 4.2 shows a bandwidth comparison of Broadcast operation (MPICH2 1.4), on ‘Zoot’, an SMP compute node depicted in Section 3.7.1, between four different binding strategies: round-robin(rr), user-defined binding(user), CPU, and cache. As

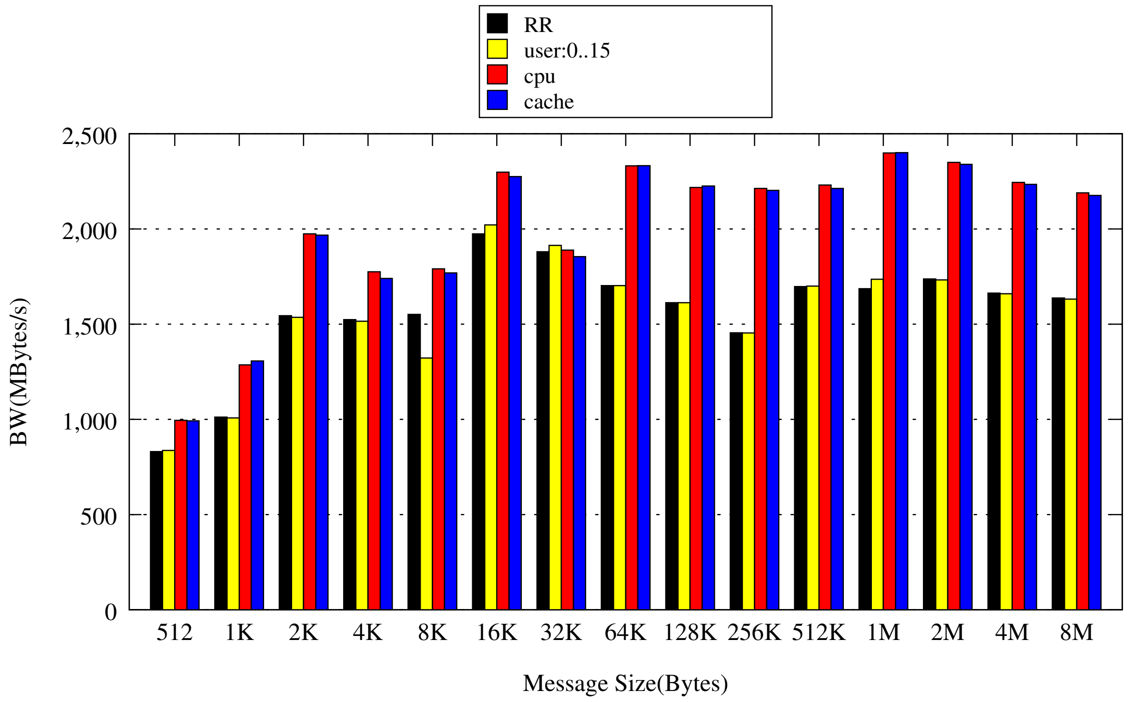


Figure 4.2: Bandwidth Comparison of MPICH2-1.4 Broadcast Operation on Zoot between 4 binding cases: round-robin, user:0..15, CPU, and cache.

depicted in Section 3.7.1, the 'Zoot' compute node consists of a quad-socket quad-core Intel Tigerton processor with a single memory controller on the front side bus (FSB), where logical consecutive core IDs belong to different sockets. Round-robin binding uses the logical order provided by the OS. MPI processes (ranks from 0 to 15) are bound to processor units (PU) in order (P#[0..15]), making neighbor processes always connected directly over the single memory FSB. 'user:0..15' binding strategy has the same binding map with round-robin binding on Zoot. Numbers behind 'user' are just PU's physical identities provided by the OS. Binding by 'cpu' and 'cache' gives out a different map with the round-robin strategy on Zoot. Neighbor processes (close in MPI ranks) are also close from each other, with respect to the number of memory buses to traverse.

In Figure 4.2, the same MPICH2's Broadcast algorithm provides different performance under different process placement on Zoot. Compared with the 'cpu' and 'cache' cases, the aggregate bandwidth is reduced by up to 35% in the round-robin ('rr') and user-defined ('user') cases. As MPICH's broadcast algorithm constructs the broadcast topology according to MPI's logical ranks, which does not include information about the physical topology, 'rr' and 'user:0..15' place the processes in a way that forces the broadcast algorithm to transfer data several times over the FSB, and therefore reducing the possibilities of cache reuse. On the opposite way, in the 'cpu' and 'cache' cases, the MPI library places the MPI ranks in a compact way: neighbor processes run on physically close cores. It turns out that the way the MPICH2 broadcast algorithm builds its internal tree matches this distribution scheme, which leads to a decreased communication time. While this worked for the MPI_COMM_WORLD communicator used in the tests, it is obvious that a communicator with the rank rearranged would have shown significantly different performance in the same process placement scenario. Moreover, the same algorithm building the internal communication tree differently, would not have reached the same performance level. Such issues are not particular to the MPICH2 library, and they can be found in other MPI implementations.

The fundamental problem in this mismatch phenomenon lies in MPI libraries' ignorance of inter-process distance; the collective communication topology is constructed according to logical MPI ranks not based on physical distances. In this context, even the most highly balanced and tuned algorithms will suffer from varied communication patterns, underlying hardware architecture, and process placement. Therefore, MPI libraries must be aware of run-time process placement, and directly reflect the physical topology on the communication pattern of the collective algorithms. Although MPI libraries can't modify run-time process placement, they can re-construct the internal communication topology according to the distance between cores on which processes run, instead of simply the MPI ranks, to match the underlying physical topologies.

The approach presented in this chapter will automatically build the best fitting collective topology for each communicator, depending on the processes involved in the communicator, the collective algorithms in use, the hardware capabilities, and the process placement.

4.4 Framework

4.4.1 Distance Between Processes

The collective framework described in this chapter bases its decisions on detected process distance. Process distance reflects how many functional units, buses, caches, memory controllers, etc. messages travel through from source memory to destination memory. In fact, this distance is computed under the assumption that each process is bound to a particular computing unit (core), and therefore the distance is relative to the core placement at the hardware level. As a result, the distances can be measured using the memory and physical hierarchies. We use four factors to compute the distances between two processes (or cores as explained above): 1) sharing any caches; 2) on the same physical socket; 3) sharing any memory controllers; and 4) on the same physical board. To simplify the algorithm design, processes sharing any caches (L1, L2, or L3 caches) are considered as being at a distance of '1', regardless of the shared cache hierarchy. For processes without common caches, we check whether or not they can satisfy conditions 2) and 3). If 2) and 3) are both satisfied, processes are at distance '2'. If 2) is unsatisfied and 3) is satisfied, processes are at distance '3'. If 2) is satisfied and 3) is unsatisfied, processes are at distance '4'. If (2) and (3) are both unsatisfied, the distance is solely determined based on condition 4). Processes on the same board are at a distance of '5', otherwise the distance is considered as being '6'. Distance range can be further extended if network-style routers or switches (Intel QPI or AMD HT) are interconnecting NUMA nodes or boards. At the inter-node level, the distance can take into account network adapters, links, and even switches and routers, by a simple and natural extension. However, in the context of this section we limit the distance to '6'.

If we look at two particular hardware generations, we can exemplify the distances as follows. Zoot, described in detail in Section 3.7.1, is a 16 core machine with 32GB of memory. The system has four sockets with a quad-core 2.40 GHz Intel Xeon Tigerton E7340 CPU featuring 4 MB L2 caches shared between pairs of cores. A single SMP memory controller in the north-bridge chipset connects all the sockets with the global shared memory. There are several scenarios for distances between processes on Zoot. MPI processes can be bound to different cores on the same die, sharing a L2 cache (distance '1'), different dies on the same socket (distance '2'), or on different sockets (distance '3').

With more cores, memory/cache hierarchies, and network-style interconnects integrated into modern multi-core processors, distance between processes becomes more complex. IG is a 48-core machine with 128GB of memory depicted as Figure 3.2. The system is composed of 8 sockets with a six-core 2.8 GHz AMD Opteron 8439 SE CPU, 5 MB L3 caches and 16 GB memory per NUMA node. The sockets are further divided as two sets of 4 sockets on two separate boards connected by an interlink. Each core on IG has a 64KB L1 cache and a 512KB L2 cache not shared with other cores. Six cores inside one socket share L3 cache and one memory controller. Distances between processes bound to the 6 cores of the same socket are equally distance '1'. Processes on different NUMA nodes/sockets but on the same board, e.g., between core#0 and core#12, are assigned the distance '5'. Processes bound to cores on different boards, e.g., between core#0 and core#24 are at distance '6'.

Based on the run-time process distance information, we construct our distance-aware collective communication framework. As a proof of concept, we implemented two distance-aware collective operations, Broadcast and Allgather, as an extension to the KNEM collective component.

4.4.2 Distance-Aware KNEM Broadcast

Algorithm 1 shows how the broadcast tree is constructed based on the distance between processes. Vertexes in the graph stand for processes participating in the communicator. Edge weight is the distance. This algorithm is similar to the Kruskal minimum spanning tree [18], with a single major change: the order of edges in the queue Q. The edges in the queue are sorted in an increasing order by weight. For the edges with the same weight, we check whether or not one of the edges includes the vertex of the root process. Edges including the root process will be moved to the front of the queue. For all edges with identical weight covering the root process, we order them by their non-root vertex's MPI rank in a non-decreasing order. For edges with the same weight without root vertex, we order them firstly by small MPI rank in one vertex and then by big rank in another vertex. The MAKE-SET(v) function constructs a set for vertex v. The FIND-SET(v) function returns the head node of the set including vertex v, which is the root process if it includes it, or a process (vertex) with the smallest MPI rank in each set if not. After sorting, the root process, or the process with the smallest MPI rank in the set, will be selected as a leader of each set. The union of all the sets build a tree with the minimum depth among all minimum weight spanning trees based on this sorting.

```
Define a forest $T \leftarrow \emptyset$
**for** each vertex $v \in V[G]$ **do**
  MAKE-SET(v)
**end for**
First store the edges of E in queue Q by non-decreasing orders of weight, whether
including root and then MPI ranks.
**for** each edge $(u, v) \in E$ from Q and T has fewer than n-1 edges **do**
  **if** FIND-SET(u) $\neq$ FIND-SET(v) **then**
    T $\leftarrow$ T $\cup$ (u,v)
    UNION(u,v)
  **end if**
**end for**
return T
```

Algorithm 1: Distance-Aware Broadcast Tree Construction

Figure 4.3 shows an example of distance-aware broadcast with 12 processes across 12 cores distributed among 4 NUMA nodes under a random binding case. NUMA node 0 and 1 are on the same board connected to another board containing NUMA node 2 and 3. There are three kinds of process distances. Processes on the same NUMA node are at the shortest distance '2'. Processes on different NUMA nodes but the same board are at distance '5'. The longest distance '6' is between processes residing on different boards connected by a slow network. The distance-aware broadcast constructs a spanning tree with root process P5 as depicted in Figure 4.3. The steps from (1) to (11) in Figure 4.3 show a sequence of union between sets. The distance-aware broadcast algorithm minimizes the number of messages crossing the slowest links. In the case of Figure 4.3, only one chunk of message crosses the links that interconnect two boards. Between processes on the same distance set, the 'leader' of each set is the process with the smallest MPI rank or the root process (any other heuristic results in an equivalent outcome). These leader processes are used to communicate with processes on the upper levels with the shortest distance. The processes at the same distance as the leader process are connected directly to the leader process of each set. This guarantees a tree with a minimum depth among all minimum weight spanning trees attributable to the order of the edges in the queue.

In the case of large messages, a pipeline can be applied along the paths of a tree containing intermediate nodes, to reduce waiting time between processes on the tree's intermediate nodes or leaf nodes. For example, along the path of $P5 \rightarrow P1 \rightarrow P7 \rightarrow P10$ in Figure 4.3, the message is split into multiple chunks, and once such a piece of data is received, a process will notify its children in the tree, creating a pipeline effect.

The distance-aware broadcast topology is different from 'fixed' topologies, which are decided statically based on the number of nodes and the expected fan-out. Our topology is dynamically adapting, varying with run-time process distribution, underlying architecture, and processes in the communicator used by the collective

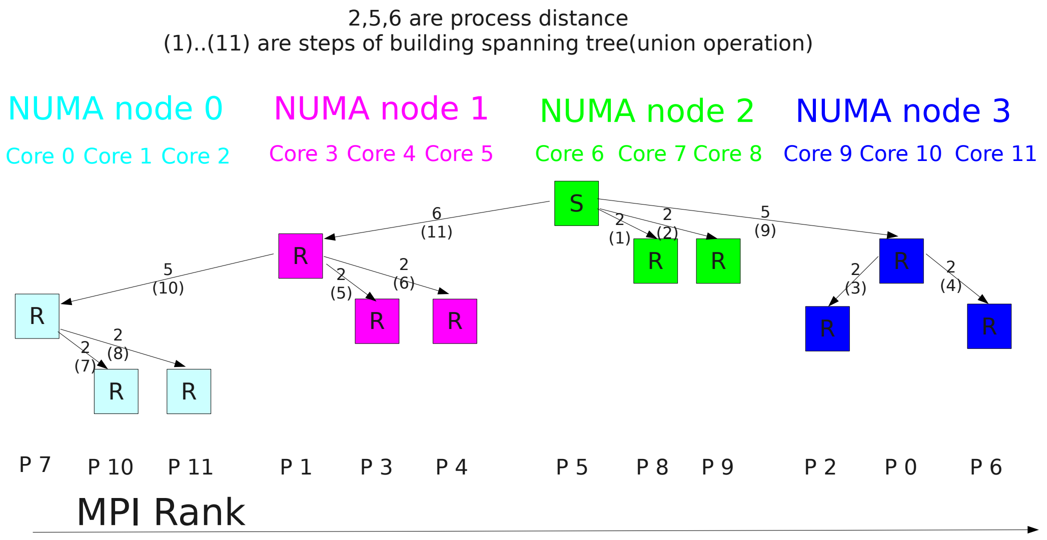


Figure 4.3: An example of a distance-aware broadcast with 12 processes on 4 NUMA nodes with random binding between processes and cores.

communication. It always provides the optimal topology regardless of process placement.

4.4.3 Distance-Aware KNEM Allgather

Algorithm 2 shows how an Allgather ring topology is constructed based on the process distance. Similar to the Broadcast algorithm, the distance-aware Allgather uses a greedy algorithm to construct the ring. The edges in the queue Q are in increasing order, first by weight, and then by MPI ranks. Physical neighbor processes are clustered together in this greedy algorithm. Only processes on edges between sets will communicate with each other via the slower links.

Figure 4.4 shows an example of distance-aware KNEM Allgather operations with 8 processes bound to a quad-socket dual-core node under a random binding case. Processes are organized in a ring structure and physical neighbor processes are arranged together along the ring. A local memory copy is executed at step (1) from sender buffer to receiver buffer with an offset of $rank \times type\_size \times count$. This is

```
Define a forest $R \leftarrow \emptyset$
**for** each vertex $v \in V[G]$ **do**
  MAKE-SET(v)
**end for**
First store the edges of E in queue Q by non-decreasing orders of weight, and
then MPI ranks.
**for** each edge $(u, v) \in E$ from Q and R has fewer than n-1 edges **do**
  **if** FIND-SET(u) $\neq$ FIND-SET(v) and fan-out of vertex u and v in each set is
  less than 2 **then**
    R $\leftarrow$ R $\cup$ (u,v)
    UNION(u,v)
  **end if**
**end for**
R $\leftarrow$ R $\cup$ (u',v'), in which u' or v' are head or tail nodes in R.
return R
```

Algorithm 2: Distance-Aware Allgather Ring Construction

the same with the step (1) in Figure 4.4. After finishing step (1), each process sends its current working index (rank in step (1)) by out-of-bound transfer to its right neighbor to notify the neighbor that a buffer is available to be retrieved. KNEM kernel copy will handle this one-sided RDMA-style pull operation. Step (2) will be repeated $N - 1$ times until all buffers are copied, where N is the communicator size. The whole operation works like an out-of-order pipeline.

Let us take an example of Allgather operation with 48 processes bound to IG's 48 cores 3.2. No matter what process placement, KNEM Allgather always constructs a ring and organizes physical neighbor MPI processes together along the ring. Processes on the same socket/NUMA node (distance '1'), are clustered as a set; 8 sets are formed and processes in each set are arranged with a non-decreasing order of MPI ranks. These 8 sets are connected in a ring following the order [2, 0, 1, 3, 4, 6, 7, 5] of NUMA node identities. Physical neighbor processes are organized as closely as possible. Left and right neighbor process ranks are calculated in the way described above.

A theoretical analysis of the memory accesses is shown for a ring-style Allgather operation. Let's assume an Allgather operation with $N \times P$ processes unfolded across

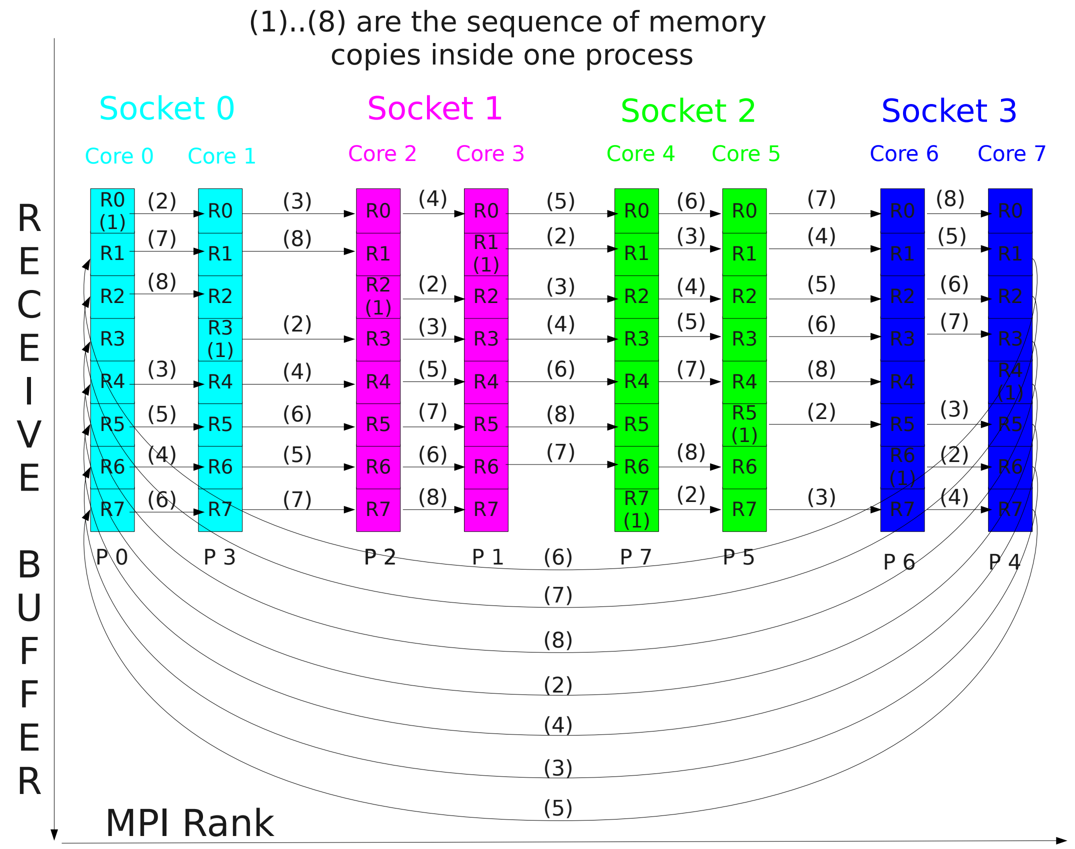


Figure 4.4: An example of distance-aware Allgather with 8 processes on a quad-socket dual-core node with random binding between processes and cores.

N NUMA nodes with P cores in each NUMA node. Each read/write access touches an amount of $type\_size \times count$ memory. Each NUMA node has $P \times P \times N$ memory reads and $P \times P \times N$ memory writes. The overall remote memory accesses along the slow links are as small as possible in this Allgather, which is the product of the number of links between NUMA nodes, and $P \times N - 1$ ($links \times (P \times N - 1)$). Only processes on the edge of each set have remote memory accesses. This distance-aware KNEM Allgather is perfectly balanced in terms of workloads to each process and memory accesses to each NUMA node. Each process has $P \times N$ times of memory copies. There is no hot-spot for any memory controller and memory accesses are distributed evenly across memory controllers. The overhead of this algorithm lies in the $N \times P$ times synchronization for each process to notify its right neighbor when its buffer is ready for transfer. Compared with the large data transfer time, this synchronization overhead is negligible in the shared memory intra-node case.

4.5 Experiments

4.5.1 Broadcast and Allgather

We use IG (see Figure 3.2) as our main experimental platform (described in Section 3.7.1). It represents a wide variety of multi-core and many-core designs, with several levels of memory hierarchies and physical distances, and can greatly benefit from a distance-aware collective communication framework. As distance-aware algorithms are implemented in the KNEM collective component, our experiment focuses on intra-node communication. The software setup includes KNEM version 0.9.5 [6]. The Intel MPI benchmark suite IMB-3.2 [5] was used to assess the difference between the collective components. Because the KNEM collective has a smaller memory footprint than other components, cache reuse in the benchmark was disabled with the *off-cache* option for a fair comparison. We used the Open MPI's default collective component, *Tuned*, to compare against. This collective component is not distance-aware, but it is considered a state-of-the-art implementation for intra-node collective communication. In this particular context (on a single shared memory node), the *Tuned* collective is configured based on SM/KNEM BTL (Byte Transfer Layer) as underneath point-to-point communication. SM/KNEM BTL uses KNEM memory copy to speed up point-to-point communication for messages larger than or equal to 4KB. A shared memory based copy-in/copy-out algorithm is still used in SM/KNEM BTL to deliver messages smaller than 4KB. In terms of point-to-point performance, the underlying technology used by the Open MPI *Tuned* collective and our distance-aware collective component are similar, and therefore any difference in performance is expected to come from the factors non-related to the point-to-point protocol.

In order to minimize the number of results, and to present a clear picture, we decided to compare two possible process placement cases: a contiguous case and a cross socket case; they represent what is expected to deliver the best, and respectively the worst, performance for a non distance-aware collective component (such as *Tuned*). The contiguous case is packing processes as closely as possible, similar to MPICH2's 'cpu' or 'cache' binding methods. E.g., in IG's contiguous case, 48 MPI processes are bound in the same order with core identities, with process i bound to core i in Figure 3.2. In the cross socket case, MPI processes from rank 0 to 47 are bound to cores in an order that maximizes the inter-socket exchanges. More precisely in the IG case the core c holds the MPI rank r iff $c = (r \mod 8) \times 6 + \lfloor r/8 \rfloor$.

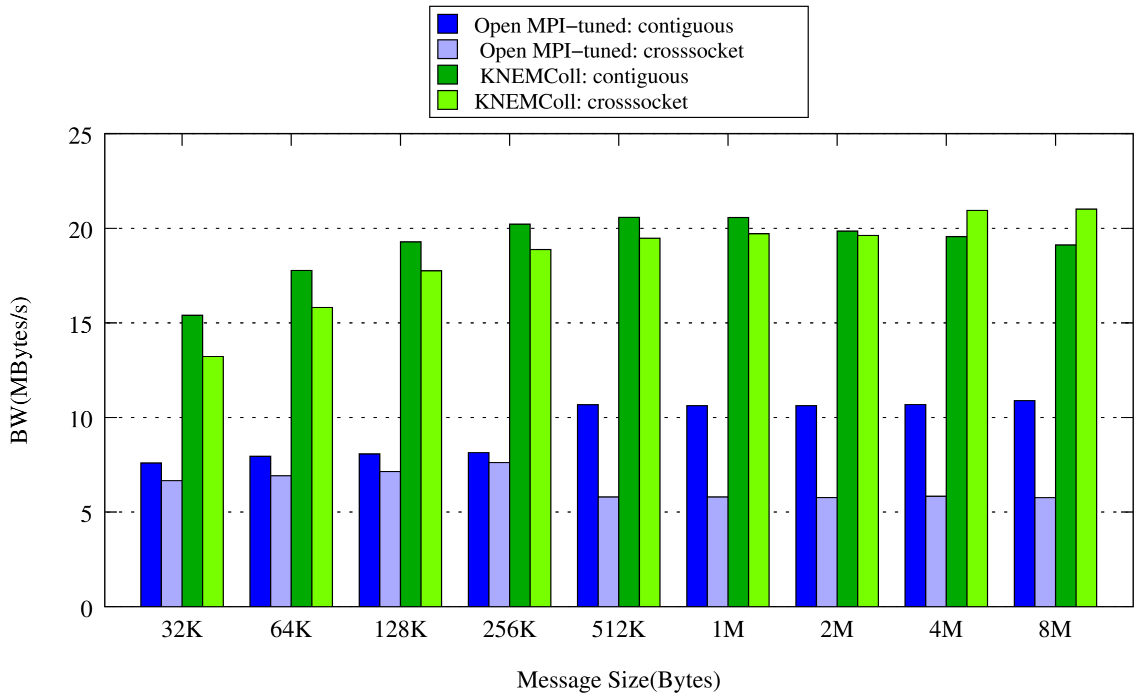


Figure 4.5: Bandwidth Comparison for Broadcast for Open MPI's *Tuned* and KNEM collective on IG between 2 binding cases: contiguous case and cross socket case.

Figure 4.5 shows the aggregate bandwidth comparison for the Broadcast collective from Open MPI's *Tuned* and KNEM collective on IG between the contiguous case and cross socket case. When the message size is larger than 256KB, the difference from different binding cases becomes obvious for the *Tuned* collective. The bandwidth loss for Open MPI's *Tuned* collective in a cross socket case reaches more than 45%, when compared with in a contiguous case. On the opposite spectrum, the KNEM collective provides stable bandwidth regardless of process placement. The variance between contiguous and cross socket cases is less than 14% in the KNEM collective, and can be attributed to the increasing cost of these synchronizations in the cross-socket case. Once the messages are large enough, the cost of these synchronizations become insignificant, and the performance of the cross-socket case is slightly better, even as the tree constructed in both cases is similar.

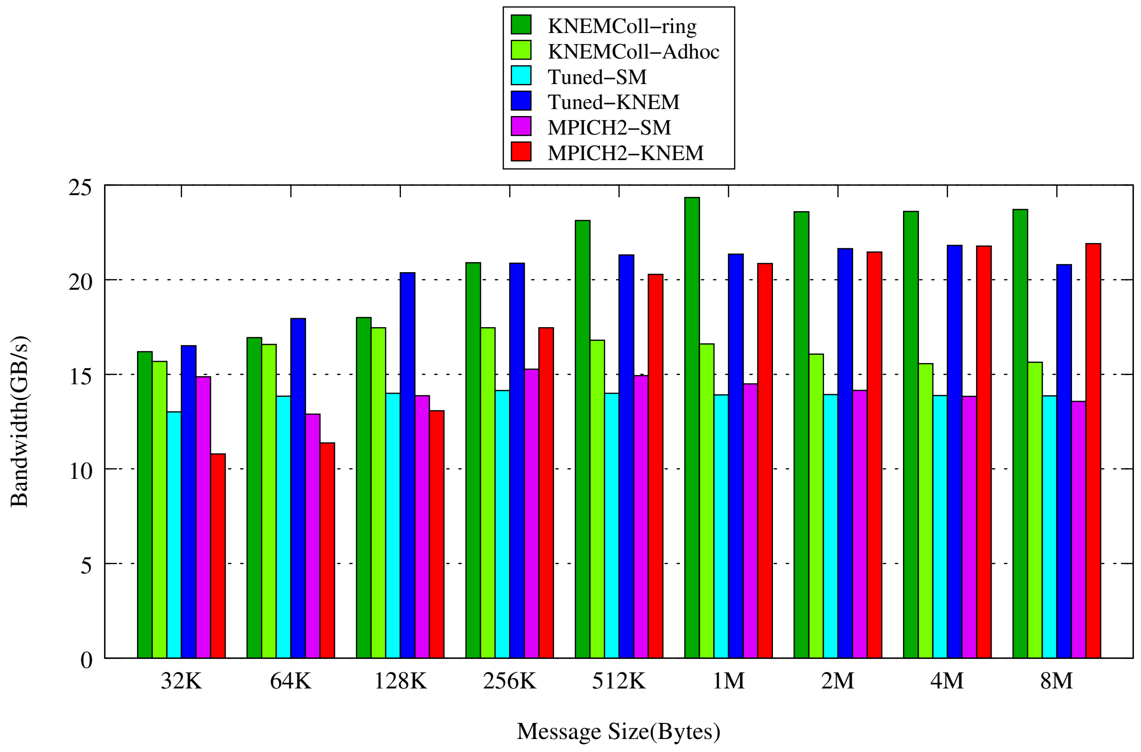


Figure 4.6: Aggregate Bandwidth Comparison of AllGather operations between Open MPI Tuned components, MPICH2, and KNEM's ring-style and Ad-hoc style Allgather.

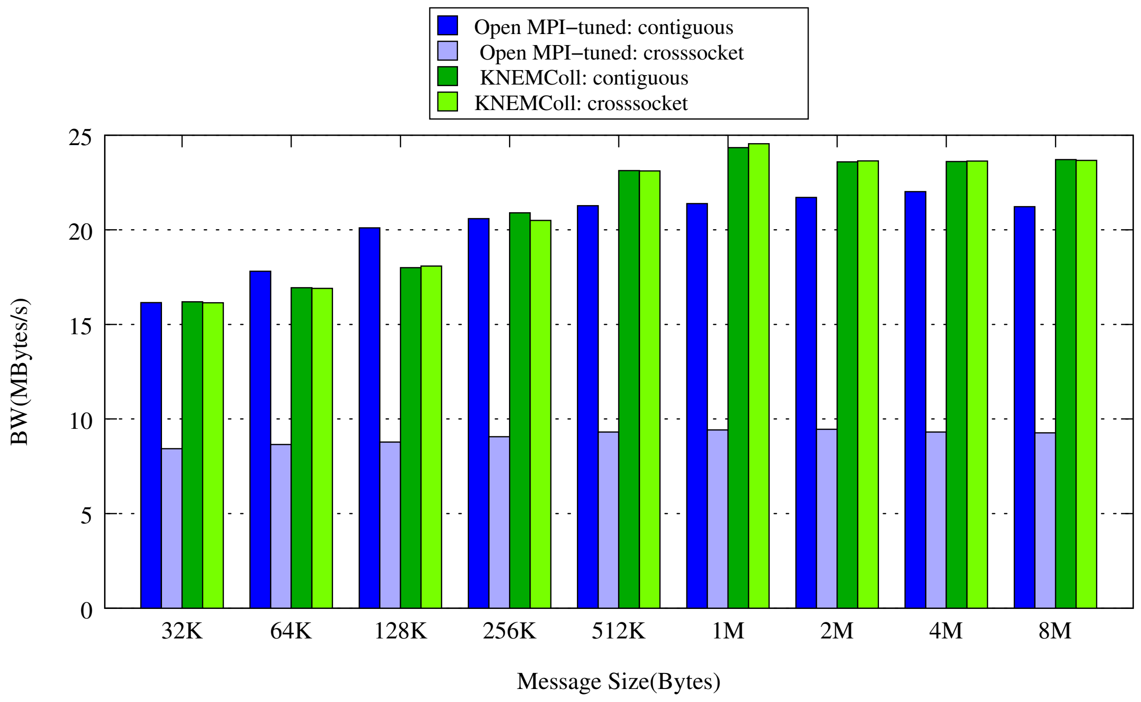


Figure 4.7: Bandwidth Comparison for Allgather for Open MPI's *Tuned* and KNEM collective with 48 processes on IG between 2 binding cases: contiguous case and cross socket case.

Figure 4.6 shows adding our distance-aware KNEM ring-style Allgather (KNEMColl-ring) into Figure 3.11(b) from Chapter 3. Even while spending extra time on process distance calculating during the communicator initialization, our distance-aware Allgather operation provides the best performance for any message size bigger than 128KBytes. The ring-style Allgather algorithm generated by the distance-aware framework outperforms the ad-hoc algorithm because of evenly distributing memory accesses among all available NUMA memory nodes on IG. The distance-aware calculation overhead in our implementation is very trivial, and only calculated once and cached during the whole life of communicators. Figure 4.7 shows the bandwidth comparison of Allgather operations of Open MPI's *Tuned* and KNEM collective on IG between the contiguous case and cross socket case. The bandwidth variance of the *Tuned* Allgather between different binding cases can reach up to 58%, significantly more than in Broadcast communication. This is because an Allgather is more communication-intensive than a Broadcast. In the cross socket case, all memory accesses between neighbors in the *Tuned* Allgather are remote memory accesses, which are slower than local memory accesses. Our distance-aware KNEM Allgather always provides stable bandwidth, regardless of the process binding.

Using the process distance information, KNEM Broadcast and Allgather construct topologies which reflect memory hierarchies and physical layout. No matter what binding map between cores and processes is decided at the MPI application level or in the process placement module, the distance-aware algorithms always provide stable and optimal bandwidth.

4.5.2 Discussion

The overhead of our distance-aware framework comes mostly from sorting the edges between processes on the topology information. Performance from Figure 4.5 and Figure 4.7 includes the overhead of collecting process placement information and constructing the collective topologies. This overhead of sorting up to thousands of edges is minimal in intra-node cases. However, on a larger scale system, it is difficult for these greedy algorithms to scale well with fully-connected graphs. Actually, only directly connected processes are helpful to construct topologies, e.g., father, children, siblings, etc. In future work, we will explore how much process placement information is necessary for each process to construct an optimal or near-optimal topology. A distributed algorithm will be a feasible approach for a large scale system.

There is another question about whether any “distance” information is equally important as another, for a distance-aware collective component. If some of the hardware information can be excluded when the internal topology is built, without losing accuracy on the decision, the decision will be greatly simplified on the next generation architectures, which are expected to have deeper memory hierarchies. Figure 4.8 shows a bandwidth comparison of a KNEM Broadcast on Zoot with 16 processes over two topologies: a two-level hierarchical tree with 4 sets and the linear

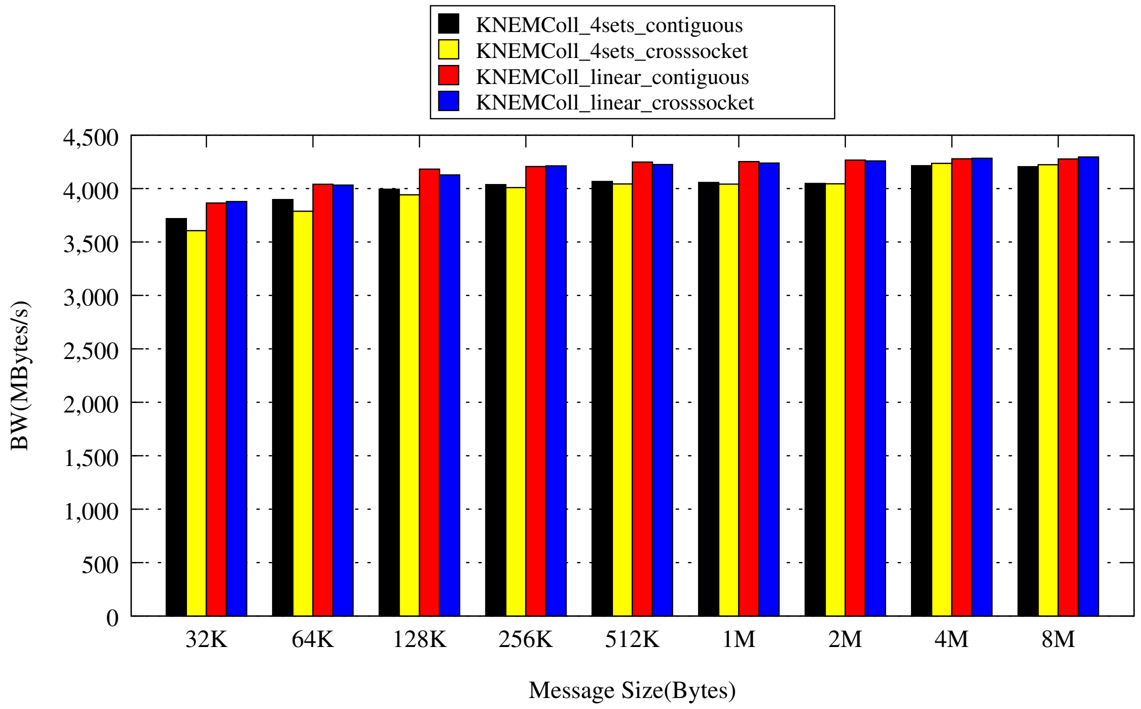


Figure 4.8: Bandwidth Comparison of KNEM Broadcast on Zoot with 16 processes over two different topologies: 4 sets and linear, between 2 binding cases: the contiguous case and the cross socket case.

topology. When considering the distance between sockets, which is distance '3' in Zoot's case, 16 processes on Zoot's 16 cores are split into 4 sets. Correspondingly, a two-level hierarchical tree is built based on these four sets following the approach described in Section 4.4.2. Ignoring the distance '3' set, a linear topology will be generated making all non-root processes access the root's buffer simultaneously. The pipeline is unnecessary in a linear algorithm because all non-root processes are leaf nodes directly connected to the root node.

From Figure 4.8, KNEM linear topology outperforms KNEM hierarchical topology. Further splitting into 4 sets according to distance '3' does not help KNEM hierarchical Broadcast on this architecture. The most plausible reason is that, although these 4 sets of processes reside on different physical sockets, they share a single memory controller. If we look at the outcome of a Broadcast communication, we can describe it as a certain number of reads from the single input buffer (the buffer at the root process), and a number of writes (flushing back into memory the buffers on each process). As a result, the single memory controller will be overloaded with write requests, and the potential benefit we can get on the read side by taking advantage of memory hierarchies, is totally annihilated. Therefore, splitting the broadcast tree on Zoot does not achieve extra bandwidth, but increases the execution path by increasing the depth of the hierarchical tree. So, distance '3' is of little importance for large message (bigger than 16KB) broadcast operations on this SMP node. However, small messages are still sensitive to physical distance between sockets, and distance '3' becomes useful again. This clearly shows that the message size is not only important for the selection of the collective algorithm, but also for the selection of how to map this topology on a particular hardware architecture.

More important information can be observed by comparing the Figure 4.2 presented in Section 4.3 with the above mentioned Figure 4.8. On the same environment, Zoot, the performance of our distance-aware broadcast communication outperforms both Open MPI and MPICH2 implementations, and is independent of the process placement.

4.6 Conclusion and Future Work

The current trend in HPC is toward a large increase in the non-uniformity of a single compute node, both from the number of cores and the number of memory/cache hierarchies. This leads to more complex architectures, and more challenging scenarios for harnessing the full potential of such environments. As the most widely used HPC software, MPI libraries must have a complete view of process distance to construct optimal topologies to allow collective algorithms to work in an efficient way. This remains true even in the case where the process distribution is optimally decided to match the most common communication pattern in the application.

In this chapter, we have presented a collective communication framework combining process distance, underlying hardware architecture, and runtime communicator composition. We supplemented this framework with two distance-aware collective operations, Broadcast and Allgather, as examples. Moreover, we have demonstrated that our distance-aware collective component provides stable and optimal performance, regardless of process placement, significantly outperforming the state-of-the-art collective components in both Open MPI and MPICH2 libraries.

Chapter 5

Extension to Distributed Memory Machines

Chapter 3 and Chapter 4 have already demonstrated that the collective performance issues incurred by multicore memory hierarchies can be solved on shared memory multicore computer nodes. The careful mapping between the collective topology and the core distance in Chapter 4 [43], and the use of kernel assisted direct memory copy mechanisms deep inside the collective algorithms [42] have been proven to greatly increase the shared memory collective communication efficiency in Chapter 3. However, on distributed memory machines, like multicore or many-core clusters, a single approach cannot encompass the extreme variations not only in the bandwidth and latency capabilities, but also in features such as the aptitude to operate multiple concurrent copies. Efficient multicore shared memory approaches are so specific, including kernel assisted direct memory copy, that it can't be applied onto network communications; on the other hand, regular network approaches fail to extract performance from shared memory links. This calls for a tight collaboration between multiple layers of collective algorithms, dedicated to managing intra- or inter-node communication.

This chapter, as an extension of our work published in IPDPS [40], presents how HierKNEM, a kernel-assisted topology-aware collective framework, orchestrates the collaboration between multiple layers of collective algorithms. Leaders are selected among the core-centric collective algorithm, to participate in the inter-node collective topology. Intra-node communications are managed by offloading memory copies to non-leader processes, taking advantage of the kernel assisted direct memory copy approach to balance the memory copy workloads among all available CPU cores. The resulting scheme enables perfect overlap between intra-node communication and inter-node communication, thanks to our innovative hierarchical algorithms. We demonstrate experimentally, by considering three distinct collective patterns (one-to-many, many-to-many and many-to-one), that 1) this approach can provide the highest performance under any process-core bindings, any hardware architecture, and any run-time communicators; 2) it outperforms state-of-the-art MPI libraries (Open MPI, MPICH2 and MVAPICH2), demonstrating up to a 30x speedup for messages between 8KB and 256KB in synthetic benchmarks, and up to 3x speedup for a parallel graph application (ASP [51]); 3) it demonstrates a linear speedup with the increase of the number of cores per computer node, a paramount requirement for scalability on future many-core hardware.

The rest of this chapter is organized as follows: Section 5.1 describes the framework for kernel-assisted hierarchical collective communication on multicore clusters and details three collective algorithms: one-to-many (Broadcast), many-to-one (Reduce), many-to-many (Allgather), and their corresponding implementations in a new Open MPI collective component: HierKNEM. These algorithms are experimentally compared with state-of-the-art MPI implementations to assess the benefits of our innovative approach in Section 5.2. Finally, Section 5.3 concludes the chapter with a discussion of the results.

5.1 Collective Algorithm Composition

5.1.1 Framework

As hinted previously, most existing approaches to develop hierarchical collective communication are based on a multi-level approach where the top level represents the largest area network, and each subsequent level is for a smaller area network. While they provide interesting performance compared with single-level approaches, they do not benefit from the entire overlapping potential of collective algorithms, as the transition processes (i.e., processes that are leafs in one level and become root on the next), are step by step blocked in a collective for a particular level. What has been missing in these attempts at providing hierarchical collective operations on clusters of multicore system was the ability to express a multi-level algorithm with a very tight level of interoperability between the levels. In the present effort, we want to enable an unprecedented level of integration between different algorithms, by dissolving the boundaries between the levels, and allowing the transition processes to overlap collectives between the inter- and intra-node levels.

From a technical point of view, in most hierarchical approaches including ours, collective communication is divided between inter- and intra-node communication. Each process has an intra-node communicator encompassing all processes hosted on the same physical compute node. Among these local processes, a leader process is selected to represent the compute node in the inter-node layer. All non-leader processes only communicate with the local leader process and then messages are forwarded by the leader process to remote leader processes on remote compute nodes. The advantage is that the messages carried through expensive inter-node links are explicit, giving leverage for the algorithm composition to minimize cross-traffic volume. From a technical standpoint, what differentiates our approach compared to previous attempts is the level of integration between the layers of the hierarchy, allowing multiple algorithms to coordinate their pipelining strategies at a very low level.

One major challenge for multi-level algorithms is to coordinate around the usage of common resources. In this particular instance, one should pay attention to the load imposed on the memory bus. This load is two-folds: on one side, sending/receiving data over the network translates into moving data across the PCI/PCIe bus from the memory bus; on the other side, moving data inside the compute node generates memory bus traffic, and therefore collides with the network transfer (the data in Figure 5.4 highlight this fact). Therefore, special care has been taken to minimize the number of memory transfers at the inter-node level. The approach chosen in this framework is to base all intra-node memory transfers on the KNEM collective components, described in Chapter 3. KNEM's offloading capability is naturally matched up with leader-based hierarchical collectives: workloads of memory copies can be off-loaded onto non-leader processes. Non-leader processes can simultaneously read or write leader processes' memory through KNEM primitives; meanwhile, leader processes can dedicate themselves to inter-node forwarding, without sequentialization experienced by less integrated hierarchical approaches. For communication strictly within large NUMA compute nodes, different approaches yield varying performance. Our new hierarchical algorithms leverage the knowledge accumulated on a shared memory multicore compute node to design sound algorithm compositions that can cope with a large number of cores within compute nodes.

In this new context, we provide three improved versions of the most used collective operations: a one-to-many (Broadcast), a many-to-many (Allgather) and a many-to-one (Reduce).

5.1.2 Broadcast

Let's assume the intra-node communicator for each compute node is named by 'lcomm', the inter-node communicator for leader processes is 'llcomm', and process

```
Input: MPI_Bcast(void *rubf, int count, MPI_Datatype dtype, int root, MPI_Comm comm)
if P is leader process then
    Register rbuf into KNEM device and get a cookie;
    Broadcast this cookie to all non-leader processes on the same compute node;
    if P is root process then
        for i ← 1 to seg_num do
            Isend segment i to its children in spanning tree;
            Wait for all Isend;
        end
    else if P is a leader process in an intermediate compute node then
        Post Irecv for 1st segment from its father;
        for i ← 1 to seg_num-1 do
            Post Irecv for next segment(segment i+1) from its father;
            Wait for previous Irecv(segment i);
            Isend received segment(segment i) to its children;
            Barrier in the lcomm communicator;
            Wait for all Isend;
        end
        if i ≡ seg_num then
            Wait for previous Irecv(last segment);
            Isend last segment to its children;
            Barrier in the lcomm communicator;
            Wait for Isend;
        end
    else
        for i ← 1 to seg_num do
            Recv segment i from its father;
            Barrier in the lcomm communicator;
        end
    end
    Barrier in the lcomm communicator;
    Deregister buffer from KNEM device;
else
    Get KNEM cookie from the leader process;
    if P is on the same compute node with root process then
        Fetch the whole data from root process by KNEM;
    else
        for i ← 1 to seg_num do
            Barrier in the lcomm communicator;
            Fetch segment i from leader process by KNEM;
        end
        Barrier in the lcomm communicator;
    end
end
```

Algorithm 3: The HierKNEM Broadcast Algorithm.

rank is ‘P’. Suppose a two-level Broadcast algorithm, using a spanning tree-based approach for the inter-node level and a linear approach for the intra-node level. Our HierKNEM Broadcast algorithm is adaptive enough to handle special cases, e.g., when all processes are allocated on a single compute node, our broadcast is transformed into a linear algorithm identical to the KNEM linear Broadcast; when each compute node has a single process in the communicator, our HierKNEM broadcast is automatically morphed into a spanning tree broadcast identical to the inter-node level.

Algorithm 3 presents the pseudo-code of the HierKNEM Broadcast. In order to save space, we trimmed the pseudo-code handling the special cases mentioned above and presented the algorithm processing a general case: each compute node has more than one process bound to different cores and all leader processes are organized into a spanning tree with more than two levels: a root node, intermediate nodes, and leaf nodes. At first, each leader process registers ‘rbuf’ into the KNEM device and gets a ‘cookie’ back at step 2. This cookie is a unique identifier to point to an entry recording rbuf’s physical memory address, and any other process in the compute node having this identifier can access (based on the granted right) this registered buffer via the KNEM memory copy module. This cookie will then be broadcasted to all non-leader processes on the same compute node (step 3 and 33). Afterward the message is divided into equal-sized fragments and forwarded in a pipelining fashion along the spanning tree composed of all leader processes (between step 4 and 29). In this particular context, father and children mentioned in Algorithm 3 refer to the process up and down the spanning tree from the current process P. For intermediate and leaf nodes in the spanning tree, once the leader processes receive a segment from its father node, they will notify all non-leader processes on the same compute node to fetch the segment (step 15 and 21). Upon receiving this notification at step 38, each non-leader process will fetch the segment by a KNEM *get* operation (receiver-reading) at step 39. This *get* operation is one-sided and will be offloaded to the non-leader processes. Therefore, the overhead of intra-node data movement can be overlapped at the leader process with the forwarding between leader processes on the upper level (step 12, 14, or 20).

This is the fundamental reason why our HierKNEM collective can outperform other collective components: intra-node communication is offloaded to non-leader processes and leader processes can dedicate themselves to inter-node message forwarding. In an ideal situation, the intra-node communication overhead can be completely hidden from the overall execution time and the entire collective communication execution time is made close to the inter-node collective execution time (the collective on the leader processes communicator). In the event of a perfect overlap, a core-centric Broadcast operation can be made *number-of-compute-nodes* dependent instead of *number-of-cores* dependent.

5.1.3 Reduce

```
Input: MPI_Reduce(void *sbuf, void *rbuf, int count,MPI_Datatype dtype,
                  MPI_Op op, int root, MPI_Comm comm)
1  if P is the 1st leader process then
2      for i ← 1 to seg_num do
3          Wait notification from 2nd leader;
4          Reduction in the llcomm for segment i;
5      end
6  else if P is the 2nd leader process then
7      for i ← 1 to seg_num do
8          Fetch segment i from 1st leader;
9          Reduction between two leaders' segment i;
10         Reduction in the new_comm for segment i;
11         Push reduction result of segment i to 1st leader's tmpbuf;
12         Notify 1st leader that pushing operation is done;
13     end
14 else if non-leader processes exist inside the compute node then
15     for i ← 1 to seg_num do
16         Reduction in the new_comm for segment i;
17     end
18 end
```

Algorithm 4: The HierKNEM Reduce Algorithm.

Similarly to the Broadcast algorithm, the HierKNEM Reduce uses an inter-node communicator (llcomm) and intra-node communicator (lcomm). In addition to these two communicators, the HierKNEM Reduce creates another local communicator, a subset of the lcomm, to organize all non-leader processes on the same compute node (new_comm). This new_comm is used to isolate leader processes from the intra-node reduction. The HierKNEM Reduce is actually a double-leader algorithm: the 1^{st} leader process participates in the upper level (inter-node) reduction while the 2^{nd} leader process will be the root for an intra-node reduction on each of the new_comm communicators, and is responsible for updating the 1^{st} leader with the local contribution to the upper level reduction. Algorithm 4 describes the HierKNEM Reduce for a general case, where each compute node has more than two processes participating in a reduction operation: one leader for the inter-node reduction and another leader for the intra-node reduction. In order to save space, we trimmed the algorithm of the handling of special cases, the internal management, and distribution of KNEM registrations.

The 2^{nd} leader fetches segment i from the 1^{st} leader's sbuf by a KNEM get operation (step 8), and applies the reduction operation between their sbuf's segment i (step 9). As a root process, the 2^{nd} leader calls an intra-node reduction for segment i in the new_comm with the result from step 9. After finishing this reduction, the 2^{nd} leader will push the reduction result of segment i to the 1^{st} leader by a KNEM writing. After getting the notification from the 2^{nd} leader (step 3), the 1^{st} leader will trigger an inter-node reduction between leader processes with pushed results for segment i. The intra-node reduction for segment i+1 can be overlapped with inter-node reduction for segment i, thanks to KNEM's one-sided operation and the pipelining reduction algorithm between hierarchical communicators.

5.1.4 Allgather

The HierKNEM provides two algorithms for Allgather: a leader-based algorithm for clusters of small compute nodes (2-6 cores per compute node) and a ring algorithm for large compute nodes. The leader-based algorithm has three steps: 1) gathering messages into leader processes; 2) exchanging data between leader processes; and 3) broadcasting data from leader processes to non-leader processes. Step 1 and 3 happen inside a compute node while step 2 exchanges data using inter-node communications.

At the inter-node level (step 2), the leader processes are organized into a logical ring and each leader process communicates only with the left and right neighbors in this ring. Once leader processes get a message from step 1 or step 2, they will notify non-leader processes to fetch data by KNEM copy. Because KNEM copy in step 1 or 3 is one-sided and always offloaded onto non-leader processes, leader processes only synchronize with non-leader processes before or after non-leader processes write or read data into or from leaders. This synchronization overhead is minimal compared with the cost of intra-node data movement. As a result, the leader processes can dedicate themselves to inter-node data exchanging, and steps 1-3 can be totally overlapped. The critical path of our algorithm depends on the overhead of inter-node exchanging or intra-node gather (step 1) and broadcast (step 3). When intra-node communication cost exceeds inter-node exchanging time (more cores per compute node or faster network), the leaders' memory bandwidth is overloaded by this ad-hoc memory access pattern. Thus, the overall throughput is seriously restricted by such a simple combination of Gather and Broadcast operations. So in clusters of large NUMA compute nodes, the HierKNEM Allgather adopts a ring-based algorithm to distribute data both at the inter-node and intra-node levels in order to avoid such hot-spots on leader processes. The HierKNEM Allgather ring algorithm is similar to the MPICH Allgather ring algorithm [57]: all processes are organized into a logical ring, and each process receives messages only from its left neighbor and sends messages only to its right neighbor. This send and receive will be executed number of comm_size-1 times and a local memory copy will be executed at the beginning. A notable improvement over the ordinary ring algorithm, the construction of the HierKNEM's logical ring is not based on the order of MPI ranks, but adheres to the physical process distance in terms of sockets and NUMA compute nodes. Thus, processes physically close are clustered together into a set. Only processes on edges between sets communicate through slow links: inter-node links or inter-socket links.

5.2 Experimental Evaluation

We used two clusters of the Grid5000 experimental platform: Stremi and Parapluie. The Stremi cluster features 32 compute nodes, each with two AMD Opteron 6164 HE twelve-core CPUs (24 cores per compute node). Each socket has 10 MB L3 caches and two NUMA memory nodes, and 6 cores in each socket share one NUMA memory node with 12 GB of memory (48 GB of memory per compute node). These 32 compute nodes are interconnected by Gigabit Ethernet. The Parapluie cluster is identical to Stremi, except that the 32 compute nodes are interconnected through a 20G InfiniBand network.

Our HierKNEM collective is based on Open MPI version 1.5.3. We compared the HierKNEM collective with the Open MPI's *Tuned*, and Hierarch collective, MPICH2 version 1.4.1 on the Ethernet cluster (Stremi) and MVAPICH2 version 1.7 on the InfiniBand cluster (Parapluie). All implementations that support kernel assisted memory copy use KNEM version 0.9.5 [13] except MVAPICH2. MVAPICH2 uses LiMIC2 0.5.5 [34] as the kernel assisted memory copy module.

For intra-node communications, HierKNEM, *Tuned*, and Hierarch collective components are configured to use the SM/KNEM BTL (byte transfer layer) as an underneath point-to-point communication helper. SM/KNEM BTL uses KNEM copy to speed up point-to-point communication; for performance reasons, the copy-in/copy-out approach is still used for messages smaller than 4KB. The same configuration is applied to MPICH2 or MVAPICH2: KNEM/LIMIC copy is enabled for large message transfer (LMT). For inter-node communications, the appropriate low level point-to-point transport module is used, depending on the underlying hardware (Open IB, TCP). For all MPI libraries, the process/core binding is the default uniform "by-core" strategy, except when explicitly mentioned. In this default strategy, sequential MPI ranks are bound into adjacent processor cores until all slots of a compute node have been used, then the same process is applied for the next compute node in the list. To summarize, the underlying technology used by our

HierKNEM algorithm and all other collective components to perform point-to-point operations is similar and uses kernel assisted direct memory copy (KNEM or LiMIC); similarly process placement is comparable, therefore any performance difference roots solely in the proposed collective operation innovations.

The Intel MPI benchmark suite IMB-3.2 [5] is used to assess the difference between the collective components on a variety of collective operations. The ASP [51] problem is a typical example of a parallel graph shortest path search algorithm. It is used to illustrate how performance differences in micro-benchmarks translate into application improvement.

5.2.1 Leader Selection

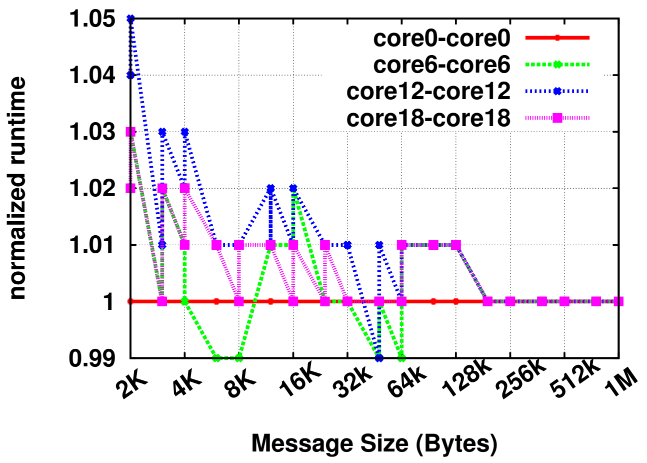


Figure 5.1: Non Uniform I/O Effect on point-to-point communication (Pingpong Test) Execution Time on Parapluie Cluster; Runtime is normalized to the result for Pingpong test between two compute nodes' core 0 (the smaller, the better).

Similar to Non-Uniform memory access (NUMA), Non-Uniform Input/Output access (NUIO) phenomenon was found in some platforms because some I/O devices are closer to some processors and memory banks than to the others [25]. The latency or bandwidth of inter-node communication between leader processes may be affected by the selection of leader processes due to this Non-Uniform Input/Output access.

We did four pingpong tests between the Parapluie's two compute nodes to inspect the NUIO impact on our experiment platforms. To simplify the results in the graph, core 0, 6, 12, and 18 are selected in the experiments to represent other cores on four NUMA memory nodes. Pingpong tests are only executed between two cores with the same id on two compute nodes. Figure 5.1 shows the execution time of pingpong tests between pairs of cores normalized to the runtime between core 0s. In most cases, the selection of processes on core 0 or core 6 as leaders will give out the best inter-node communication performance. But the difference is trivial, about 1% to 2% for message sizes selected as pipeline sizes, so even a wrong leader selection here will not lead to a performance disaster of HierKNEM collectives. In the following experiments, HierKNEM keeps selecting processes with the minimum core id as leader processes in collective communication. In the future release, we will add a module to help HierKNEM select leaders according to the hardware locality [12].

Another issue related with the leader selection is how many leader processes were selected in the collectives. HierKNEM can be configured to select one leader process from each compute node, each board, each NUMA memory node, or each socket. The run-time configuration is decided by the ratio between inter-node and intra-node communication. For example, in a scenario of large NUMA compute nodes connected by fast interconnections, e.g., a 20G or 40G InfiniBand network, the runtime of intra-node collective communication (an intra-node Broadcast) possibly greatly exceeds the runtime of inter-node forwarding (sending/receiving). HierKNEM will adjust the whole hierarchical topology by selecting leaders from each NUMA memory node instead of from each compute node. The HierKNEM Broadcast on the Parapluie Cluster follows this selection. This flexible selection will lead to better overlapping between intra- and inter-node communication, and meanwhile intra-node memory accesses will be spread evenly over NUMA memory nodes. Except for this special case, other leader selections still follow one leader process per compute node in HierKNEM collectives.

5.2.2 Pipeline Size

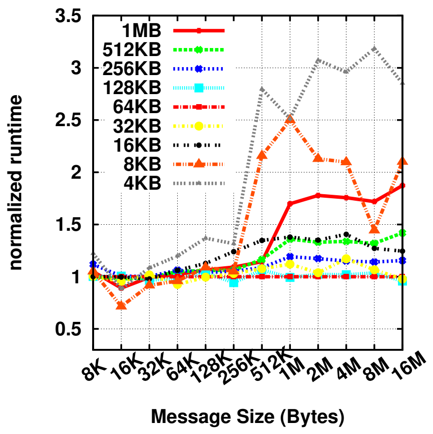


Figure 5.2: Effect of Pipeline Size on HierKNEM Broadcast Execution Time (Parapluie cluster: 768 Processes, 32 compute nodes, InfiniBand 20G); Runtime is normalized to the result for 64KB pipeline size (the smaller, the better).

In the HierKNEM collective component, both the Broadcast and the Reduce operations are pipelining algorithms, in which messages are split into several smaller chunks. Tuning an optimal size of a chunk is a key criterion of every pipeline algorithm. Figure 5.2 presents the effect of the pipeline size on the HierKNEM Broadcast execution time. In this Broadcast test, 768 processes are spawned on the Parapluie cluster. To ease figure clarity, the execution time for all pipeline sizes is normalized to the runtime obtained with a pipeline size of 64KB (t_z/t_{64}). One can see that the pipeline size is indeed critical to the HierKNEM collective performance, and a wrong selection of pipeline sizes leads to significant penalty. On one hand, a too small pipeline size results in inefficient inter-node communication, as the small message latency comes to dominate, preventing the full point-to-point bandwidth from being leveraged; as an example, the Broadcast with a pipeline size of 4KB is more than 3 times slower than with 64KB. On the other hand, a too large pipeline size results in long pipeline fan-in and fan-out phases, where the pipeline algorithm is not at steady-state efficiency. Experimentally, 8KB or 64KB is the ideal pipeline size for the Broadcast operation on the Parapluie cluster for messages smaller or larger than 64KB. We did similar experiments for HierKNEM's Broadcast and Reduce on both the Parapluie and Stremi clusters. Table 5.1 shows the best pipeline size for each operation on each type of cluster. Both HierKNEM's Broadcast and Reduce algorithms use the pipeline size in Table 5.1 in the following tests.

Table 5.1: Best pipeline size for Broadcast and Reduce for Differing Network Capacities

Operation	Parapluie (IB20G)		Stremi (Ethernet)	
	message size	pipeline size	message size	pipeline size
Broadcast	[8KB,64KB)	8KB	[8KB,512KB)	16KB
	[64KB,∞)	64KB	[512KB,∞)	32KB
Reduce	[2K, 16MB]	64KB	[2K, 16MB)	64KB
	(16MB,∞)	1MB	[16MB,∞)	1MB

5.2.3 Impact of Kernel-assisted approaches

Intra-node point-to-point operations can be implemented using shared memory copy-in/copy-out or kernel-assisted direct memory copy approaches. Kernel-assisted approaches have the benefit of decreasing the number of memory copies, a factor critical in memory intensive code sections such as collective communication. Figure 5.3 shows a comparison of the Broadcast collective bandwidth on the Stremi cluster between collective components over the kernel-assisted or shared memory point-to-point communication. The Kernel-assisted approach shows a huge performance boost in Open MPI's collective components: *Tuned* and Hierarch collectives, up to 3$\times$ for some message sizes. On the opposite side, the performance difference regarding MPICH2 is less evident. Other collective operations, such as Reduce and Allgather, exhibit similar speedup when the kernel-assisted approaches are replacing shared memory point-to-point communication. Moreover, this performance improvement on collective communication is directly inherited by the applications using them, as highlighted in Section 5.2.8. Thus, in the remainder of this chapter we focus on kernel-assisted approaches, discarding the sub-optimal approaches built on top of shared memory point-to-point communication.

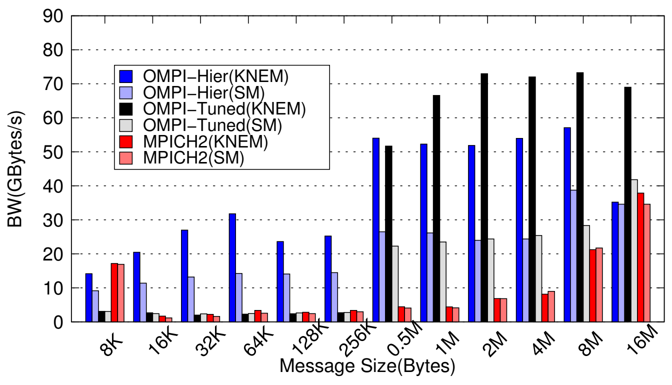


Figure 5.3: Comparison between Kernel-assisted and Shared Memory Approach: Broadcast Bandwidth on Stremi (Ethernet, 768 processes, 24 cores/ compute node)

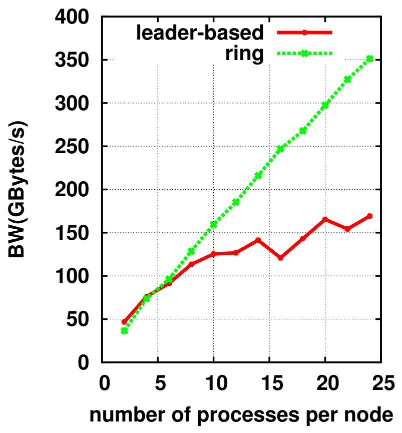


Figure 5.4: Bandwidth Comparison between Leader-based and Ring Allgather Algorithms, when increasing the number of processes per compute node (from 2 to 24), on Parapluie's 32 compute nodes.

5.2.4 Allgather Algorithm Selection

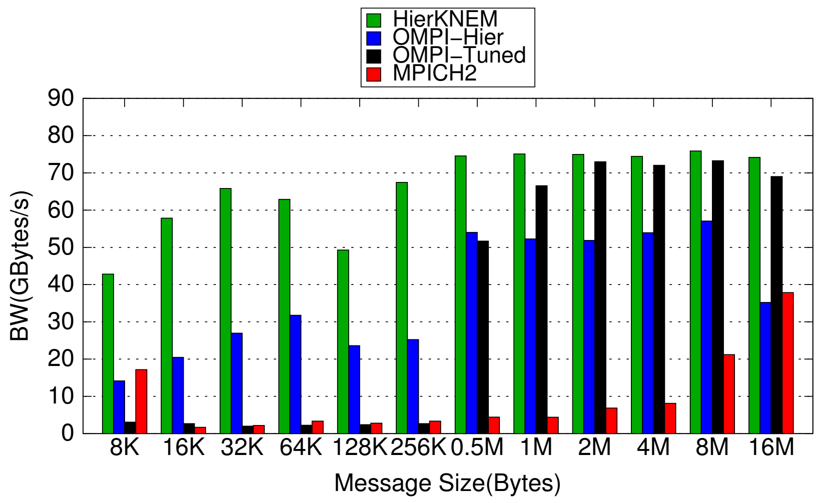
Although the two levels of algorithms are tightly integrated, there are still a variety of combinations that are possible, whose performance greatly varies depending on hardware features and properties. In the case of the Allgather algorithm, we identified two combinations of interest: both use the pipelined *Tuned* collective component between compute nodes, but the internal operation differs depending on the number of cores between compute nodes. Between cores, the algorithm can rely on the leader originating all messages simultaneously (referred to as "leader-based" algorithm), but for large core counts, this approach has the potential to result in heavy traffic contention on the memory bus of the core hosting the leader. For a larger number of cores per compute node, the ring algorithm has more potential to even out the load on all cores. Figure 5.4 shows the aggregate bandwidth for the two algorithms combinations for a 512KB message's Allgather operation on Parapluie's 32 compute nodes when increasing the number of processes per compute node from 2 to 24. The leader-based algorithm has a slight performance advantage in dual-core or quad-core compute nodes, as the parallel KNEM accesses overlap one another. For larger setups, the bandwidth contention on the leader core prevents aggregate bandwidth to scale, while the ring algorithm, which proves more scalable thanks to evenly distributing data access load across all memory links, dominates. Results (not presented here) are similar for other message sizes, and when using different inter-node networks on Stremi and Parapluie clusters. In the following tests, we use the ring algorithm as we mainly target large multicore compute nodes.

5.2.5 Collective Communication Performance

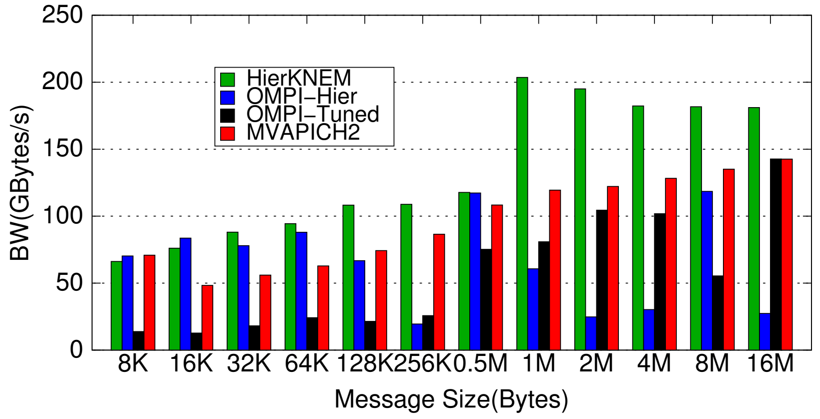
In this Section we will analyze the performance of three of the most used collective communication operations, namely Broadcast, Reduce, and Allgather. They cover all of the regular collective patterns available in MPI, one-to-many, many-to-one and many-to-many, providing a quite extensive view of all the potential performance improvement in collective communication.

Broadcast Performance

Figure 5.5 presents the aggregate Broadcast bandwidth for HierKNEM, Open MPI's Hierarch and *Tuned* components, and MPICH2 or MVAPICH2 on, respectively, the Ethernet Stremi cluster or the InfiniBand Parapluie cluster. On Stremi (Figure 5.5(a)), for message size between 8KB and 256KB, HierKNEM Broadcast



(a) Stremi (Ethernet)



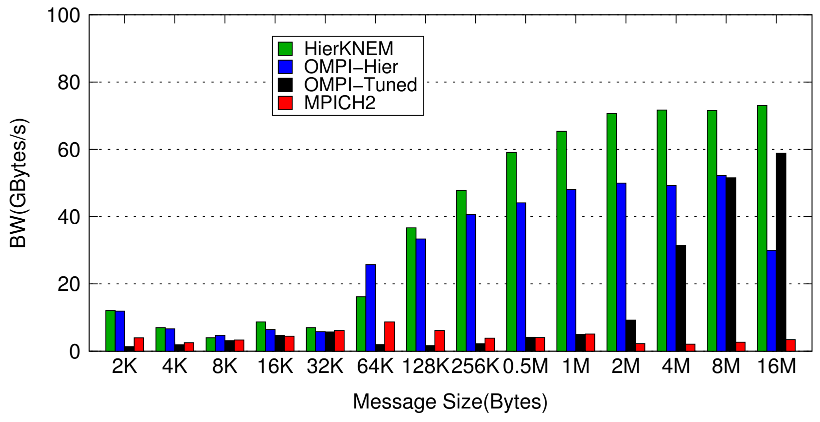
(b) Parapluie (IB20G)

Figure 5.5: Aggregate Broadcast bandwidth of collective components on multicore clusters (768 processes, 24 cores/ compute node).

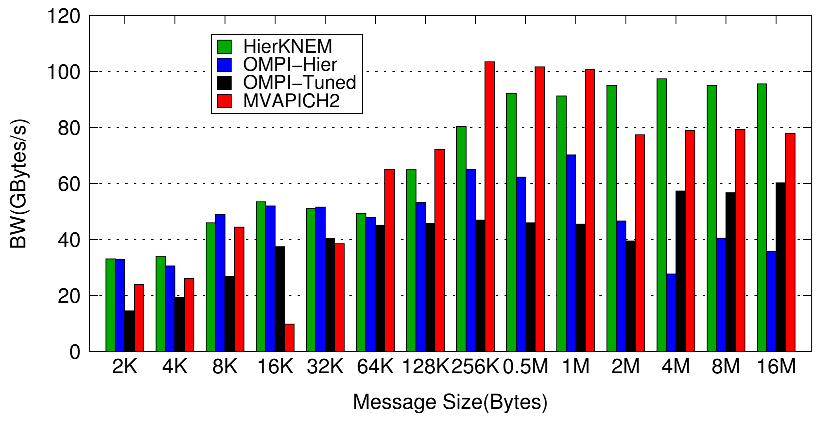
provides a significant speedup, sometimes up to 30x, when compared with MPICH2 and Open MPI. Compared with OpenMPI's Hierarch, our HierKNEM version provides more than twice the aggregate bandwidth in this message size range.

For larger message sizes (larger than 512KB), the most important tuning factors are process mapping and proper pipelining to evenly spread the workload across cores and links. In the *Tuned* collective component, the "by core" binding luckily happens, in this experiment, to match the hardware topology; and the pipeline size selected by the *Tuned* collective component to optimize the network communications is suitable for core communications. The Hierarch collective component of Open MPI is not as successful for large messages, because the intra-node and inter-node layers do not cooperate to evenly spread the load of the pipelining algorithm. The leader processes are unavailable for long periods of time when they take part in the shared memory local operation, resulting in effectively sequentializing the local and remote collective operations without opportunity for overlap. With such a large core count, the large intra-node overhead offsets the benefits of the standard hierarchical algorithm. In contrast, the HierKNEM algorithm obtains the best performance in all cases, thanks to explicitly taking into account process mapping and using directional KNEM control to offload parts of the operations onto the non-leader processes, hence enabling intra and inter-node communication to overlap.

Similarly, on the InfiniBand cluster (Figure 5.5(b)), in most cases, the HierKNEM Broadcast still outperforms other collective components. One major difference in the results, when compared with the Ethernet case, is that the performance of the classical hierarchical algorithm is much better for small message sizes. On the InfiniBand network, the tuning parameters selected by the two non-cooperating algorithms forming the hierarchical collective are matching better. However, as one can see, the tuning parameters for larger message size are not as lucky; the performance for large messages drops, with the notable exception of 512KB messages, for which the pipeline length matches the balance for 32 processes and 24 cores. This discrepancy illustrates the difficulty of tuning the behavior of separate collective algorithms cooperating in a hierarchical manner. Even with expert knowledge, it is unrealistic to tune Open MPI's *Tuned* collectives on such a complex system with so many hierarchies and diverse networks, when a small variation in message size results in unexpected and dramatic performance consequences. Although the HierKNEM collective is not immune to the challenges of unpredictable and unstable performance on varying hardware, the fact that both algorithms select compatible tuning parameters, that the outer collective operation can overlap imperfection on the inner operation, and that the collective topology is constructed to match core hierarchies greatly alleviates this difficulty, as illustrated by more stable results across the message size range.



(a) Stremi (Ethernet)



(b) Parapluie (IB20G)

Figure 5.6: Aggregate Reduce bandwidth of collective components on multicore clusters (768 processes, 24 cores/compute node).

Reduction Performance

Figure 5.6 presents the aggregate Reduce bandwidth on the Ethernet cluster (Figure 5.6(a)). For message sizes between 2KB and 32KB, the HierKNEM Reduce competes closely with Open MPI's Hierarch Reduce. After 64KB, the HierKNEM Reduce dominates other collective components, thanks to a good overlapping between inter-node Reduce and intra-node Reduce. Similarly with the Broadcast, the Hierarch

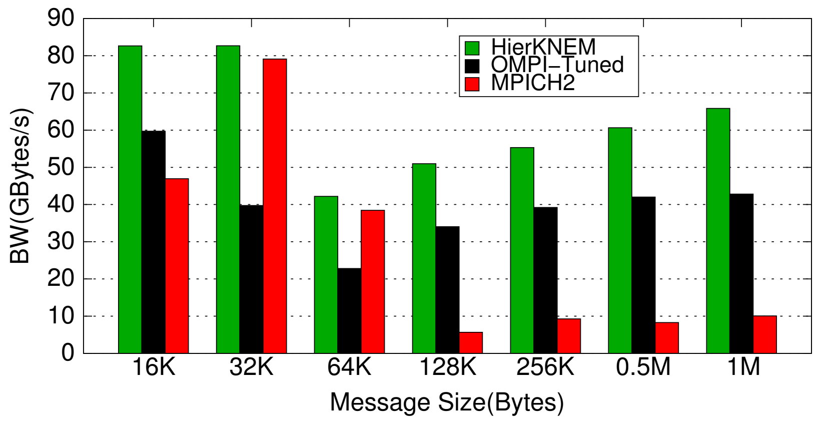
Reduce worsens for large messages due to the increased intra-node Reduce overhead which can not be dodged by overlap. Again, the performance of the *Tuned* Reduce improves for messages larger than 4MB, but is still 19%-28% slower than the HierKNEM Reduce.

On the InfiniBand cluster (Figure 5.6(b)), the HierKNEM Reduce clearly dominates for message size in the range of [2KB, 32KB] and (1MB, 16MB]. When message size is between 64KB and 1MB, although HierKNEM Reduce still achieves significant speedup when compared with Open MPI's Hierarch and *Tuned* Reduce, it is a little behind MVAPICH2 performance, about 20% in the worst case. By profiling a 64KB message's Reduction operation with 32 processes on Parapluie's 32 compute nodes (no multicore or hierarchies), we discovered that the Open MPI *Tuned* Reduction suffers from a serious performance limitation on the InfiniBand network; meanwhile MVAPICH2 enjoys very good performance (366μs for Open MPI compared to 281μs for MVAPICH2). As our HierKNEM composite algorithm reuses the original *Tuned* collective component for inter-node communication, it suffers from the same defect and cannot compete with MVAPICH2, until the Open MPI community addresses this issue.

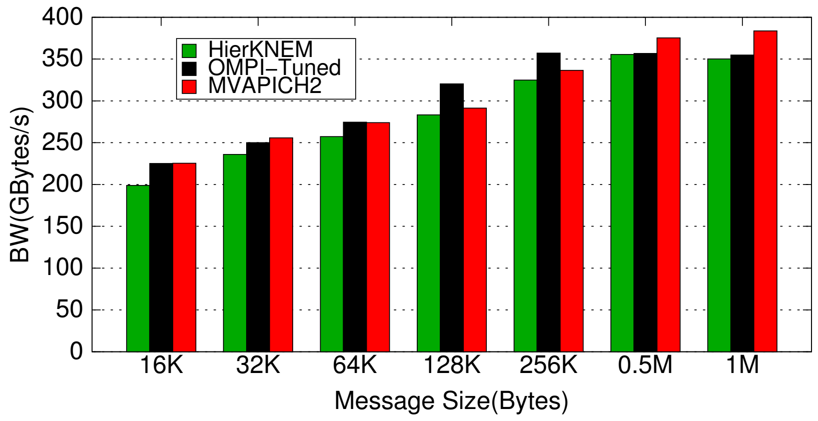
Allgather Performance

Figure 5.7 presents the aggregate Allgather bandwidth. The HierKNEM Allgather is enabled only when the message size is larger than 8KB. The biggest message size is 1MB, because of the large amount of memory required for this all-to-all operation between 768 processes exhausting available system memory for larger sizes. On both clusters, the HierKNEM Allgather adopts a ring algorithm, as described in section 5.1.4. The Open MPI Hierarch collective component is not presented for this collective operation, as it has not been implemented.

On the InfiniBand cluster (Figure 5.7(b)), both MVAPICH2 and *Tuned* Allgather operations slightly outperform HierKNEM's Allgather. In this message range, Open MPI's *Tuned* Allgather adopts a similar ring-style but more aggressive



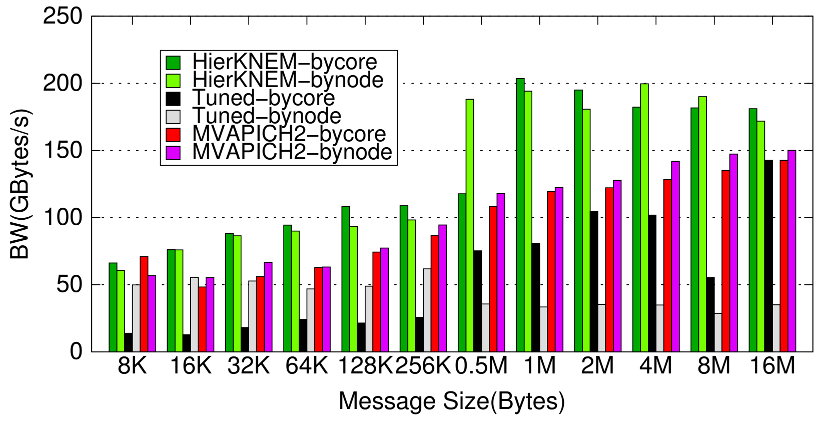
(a) Stremi (Ethernet)



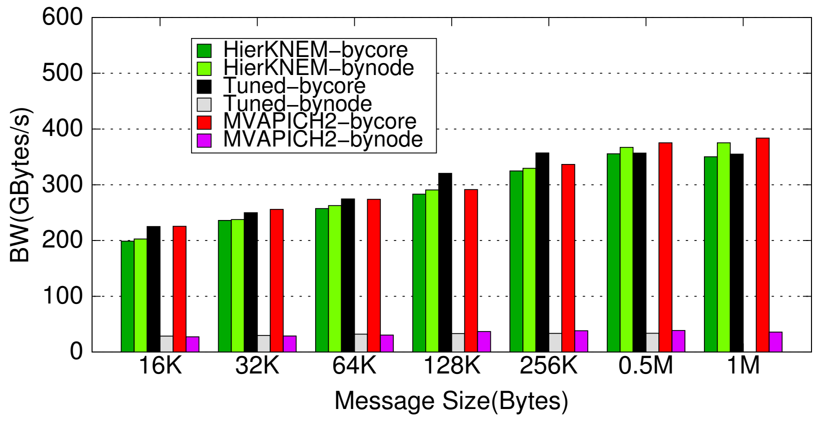
(b) Parapluie (IB20G)

Figure 5.7: Aggregate Allgather bandwidth of collective components on multicore clusters (768 processes, 24 cores/ compute node).

algorithm: neighbor exchange. Different than the ordinary ring algorithm which exchanges rcount data in each round, 2× rcount data are exchanged in each round in the neighbor exchange algorithm (except the first round). The neighbor exchange algorithm arranges processes continuously in MPI ranks together in a ring, and the "by core" binding strategy used in this test coincidentally maps the logical ring of the *Tuned* Allgather correctly to the underlying hardware topology. As a consequence, *Tuned* and HierKNEM are actually running on the same underlying hardware topology, but *Tuned* does not have to pay for the extra cost of detecting the physical distance between processes, and fast networks like 20G InfiniBand show better performance in a more aggressive policy due to more available bandwidth capacity. On the Ethernet cluster (Figure 5.7(a)), the HierKNEM Allgather outperforms all other collective components for all message sizes. While adopting a similar ring topology for large messages, the *Tuned* Allgather on this Ethernet cluster suffers up to 50% in performance loss because the aggressive policy in the neighbor exchange algorithm overloads slow networks, thereby decreasing the overall network throughputs. Clearly, adopting an aggressive strategy, like the neighbor exchange algorithm, should depend on the capacity of the networks.



(a) Broadcast



(b) Allgather

Figure 5.8: Impact of process mapping: aggregate Broadcast and Allgather bandwidth of the collective components for two different process-core bindings: by core and by node (Parapluie cluster, IB20G, 768 processes, 24 cores/ compute node).

5.2.6 Impact of Process Placement

It is well known that process placement can have a major impact on collective operations performance. Approaches such as MPIPP [16] have been designed to detect communication patterns during a “tuning run”, whose result is used to hint process placement to decrease long distance communication volume during subsequent production runs. However, this approach is not practical in many cases, as it requires being able to run smaller problems that exhibit similar communication patterns; and the collective algorithm underlying communication topology might depend on the message and communicator size. Another difficulty is that, one might want to optimize for the pattern of point-to-point operations explicitly realized at the application level (such as the typical hypercube topology found in many CG implementations), which means that the process placement may or may not fit the expectations of the collective components. As a consequence, the default deployment approach is usually less elaborate and simply allocates ranks sequentially on the available resources.

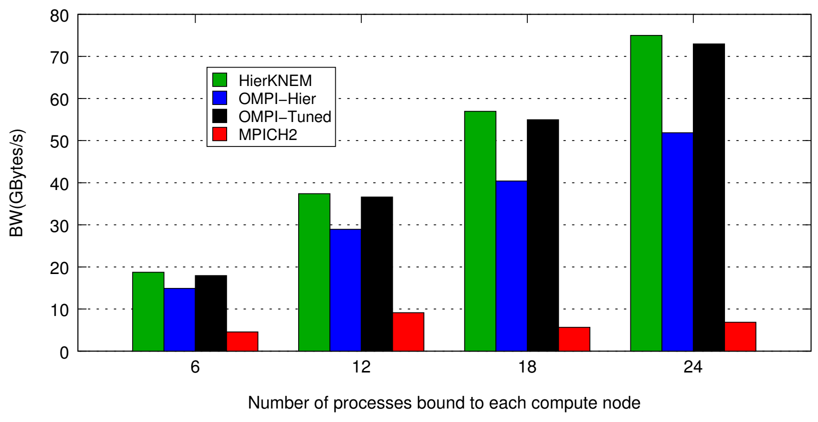
Figure 5.8 shows the impact of two typical process placements on the performance of the Broadcast and Allgather operations. The goal of this experiment set is to investigate the sensitivity of the hierarchical approaches to variations in the process placement. As such, more than raw performance, it is the difference between the same algorithm on different mappings that is of interest here. The Hierarch collective has been trimmed from the figure because it does not feature an Allgather operation, and uses a similar topology as HierKNEM for the Broadcast (hence similar performance trends). Considering the Broadcast (Figure 5.8(a)), one can see that hierarchical approaches (HierKNEM and MVAPICH2 both feature a hierarchical algorithm) achieve more stable performance. The *Tuned* algorithm exhibits very unstable performance trends, for some message sizes the bynode binding reaches better performance, while it is the contrary for larger messages.

Figure 5.8(b) further displays the importance of considering hierarchical features to enable portability of performance across varied process mappings. In this algorithm, the HierKNEM algorithm demonstrates very stable performance when changing from bycore to bynode process mappings. The performance variation between two bindings is less than 10%, which is very small when compared to the tremendous performance penalty suffered by non hierarchical algorithms, commonly more than 6× and sometimes up to 14× increased communication time. In the "by node" binding, the *Tuned* Allgather uses a ring-style neighbor exchange algorithm for large messages; every edge of the logical ring (768 edges in this case) passes through inter-node links (InfiniBand), causing serious traffic congestion on the InfiniBand network. This clearly illustrates the penalty suffered by topology-unaware algorithms when considering irregular process-core bindings. Although our HierKNEM collective pays an overhead due to constructing the internal topology, it provides stable performance independently of process placement. Such a flexible process placement is a desirable feature to enable deeper optimization of the hard-coded point-to-point communication patterns and ensures maximum performance with default settings on complex architectures.

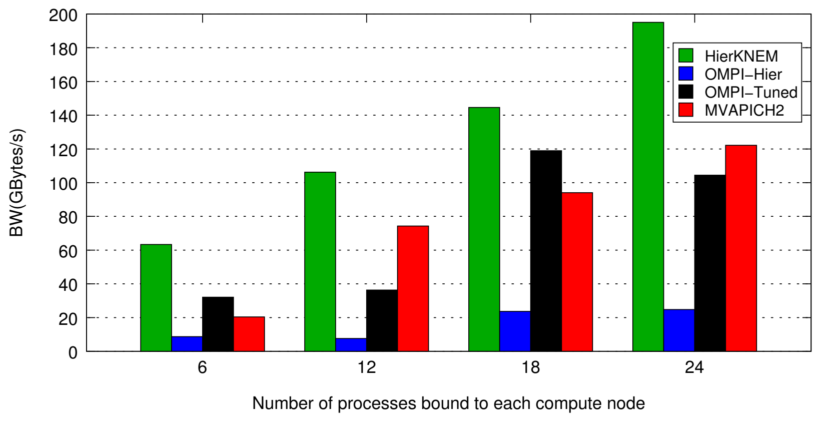
5.2.7 Core per Node Scalability

In the next experiment, we investigate the trend of aggregate bandwidth when varying the number of cores per compute node. The total number of compute nodes is left unchanged (32 compute nodes), but the number of processes per compute node is increasing for each experiment, reaching up to the maximum of 24 processes per compute node. The message size is kept constant at 2MB. Processes on each compute node are bound to cores sequentially.

On both clusters (Figures 5.9(a) and 5.9(b)), the aggregate bandwidth of HierKNEM Broadcast achieves a linear speedup when more cores per compute node are involved, because our HierKNEM Broadcast dodges the intra-node communication



(a) Stremi (Ethernet)



(b) Parapluie (IB20G)

Figure 5.9: Core per compute node scalability: aggregate bandwidth of Broadcast for 2MB messages on multicore clusters (32 compute nodes).

overhead by overlapping it with the inter-node message forwarding. Increasing processes (cores) per compute node does not increase the overall Broadcast completion time on these two platforms. This linear speedup can be maintained until the time necessary to perform the entire intra-node communication (a KNEM Broadcast) exceeds the inter-node forwarding time of the network.

5.2.8 Application Performance

We have asserted the maximum possible performance improvement by solely executing synthetic benchmarks over the modified operations. The next step is to evaluate how much of this improvement results in improved performance for applications. To evaluate the impact of the HierKNEM collective algorithms on real application performance, we consider a typical parallel graph application: ASP [51], used in Chapter 3. This application unfolds the parallel Floyd-Warshall algorithm to solve the all pairs shortest path problem. At the beginning of each iteration, the master process broadcasts a row of the square matrix representing edges' weight to all peers in the communicator, in order to distribute the workload. The outer loop of the algorithm iterates on rows, until the entire matrix is treated. Overall, for a matrix of size N, the algorithm performs N broadcasts, with a message size of column_num $\times$ type_size. As a consequence, MPI_Bcast contributes to the majority of the runtime of the ASP's MPI usage.

Table 5.2: Kernel-Assisted Approach Comparison: ASP Application Execution Runtime Execution Breakdown on Stremi (Ethernet, 768 processes, 24 cores/ compute node). Using KNEM and SM.

Problem	HierKNEM		Tuned		Hierarch		MPICH2	
Size	Bcast	Total	Bcast	Total	Bcast	Total	Bcast	Total
16K	20.3s	97.4s	229s	308s	31.7s	109s	128s	204s
32K	79s	711s	929s	1560s	173s	806s	417s	1020s

			Tuned-SM		Hierarch-SM		MPICH2-SM	
			Bcast	Total	Bcast	Total	Bcast	Total
16K			229s	309s	66s	145s	124s	201s
32K			929s	1574s	245s	899s	429s	1040s

Table 5.2 compares the overall execution time and communication time (mostly MPI_Bcast) of the ASP application on the Stremi cluster when using different collective components. By subtracting the communication time from the overall execution time, one can assert that ASP's computational part remains generally constant for a given problem size, independently of the communication setup. The major performance difference between these four setups comes from the communication overhead (MPI_Bcast). The cost of communications occupies 21% of the overall application runtime for the HierKNEM collective, while it rises to 74% when using Open MPI's default *Tuned* collective. Even considering the hierarchical

broadcast, the HierKNEM's ability to overlap between inter and intra communications shows a significant improvement in this application.

The second part of Table 5.2 shows the same application (ASP) on the same platform using shared memory point-to-point communication instead of kernel-assisted approaches. Similarly to the prior results in Section 5.2.3, most collective components over shared memory point-to-point communication lose performance compared with kernel-assisted point-to-point communication due to double memory copies and more memory usage, sometimes up to 30%. The impact on collective performance is directly translated to wasting time for the applications.

5.2.9 Experiments Accuracy

In this experimental evaluation, we preferred using widely accepted benchmarks, such as the Intel Message Passing (IMB) benchmark, for they are specifically designed to eliminate noise and artifacts. For small and intermediate messages, experiments are executed thousands of times with different roots in order to eliminate any hot caching and pipelining effect, while the experiments are executed hundreds of times for large messages. To further emphasize reproducibility, for each experiment, we reserved the whole clusters to avoid sharing network switches with other users. In addition to these precautions, every experiment was realized multiple times, and we took into account not only the minimum, maximum, and average values, but the standard deviation as well. For the message size of 512KBytes, the IMB tests resulted in a bandwidth between 25.6 and 26.2GByte/s, with an average of 25.9GByte/s, and a standard deviation of 0.17. Similarly, particular care was taken regarding the standard deviation of the ASP application performance. Among 18 times of repeated experiments, the maximum ASP execution time was 146s, the minimum time 144s, the average execution time is 145.1, and the standard deviation is 0.6. These small standard deviation values are indicative that the design of the IMB benchmark, ASP application, and the precautions we have taken guarantee the accuracy and stability of our experimental results.

5.3 Conclusion

In this chapter, we described a kernel-assisted topology-aware collective framework, HierKNEM, which enables efficient combinations of multiple layers of collective algorithms, to tackle collective communication on clusters of many-core compute nodes. The algorithms are built reusing modular combinations of existing collective algorithms (such as the *Tuned* and the KNEM components in Open MPI). The main contributions of this chapter are: 1) propose an adaptive hierarchical collective framework to enable tight collaboration between the collective algorithms pertaining to different layers of the hierarchy, 2) combine offloading and pipelining techniques into the hierarchical framework to release leader processes from intra-node data movement, hence maximizing the overlap between inter- and intra-node communications, and 3) build internal collective topologies to form a mapping between the runtime process-core binding and the hardware features, which means stable collective performance independent of process placements.

We demonstrated the benefits of this approach by devising three hierarchical integrated collective algorithms, one of the most useful for each major type of collective communication (one-to-many: Broadcast, many-to-one: Reduce, and many-to-many: Allgather). Experimental results demonstrate that, our approach outperforms not only hierarchy-unaware state-of-the-art MPI implementations (MPICH2 and Open MPI's *Tuned* collectives), even these setups benefit from kernel assisted memory copies as well (KNEM point-to-point transfer), but also significantly outperforms approaches that account for the hierarchy (MVAPICH2 Broadcast, Hierarch component in Open MPI). A simple leader based algorithm that does not enable pipeline coordination, and intra-node copies offloading, under-performs compared to our HierKNEM approach that introduces these features. The performance improvement is visible not only in synthetic benchmarks, but also results in up to a ten-fold performance improvement when compared to the default hierarchy-unaware strategy, and still features two-fold improvements when compared to other hierarchical strategies.

Chapter 6

Conclusion and Future work

6.1 Conclusion

In this dissertation, we have identified three critical issues specific to MPI collective communication on multi- or many-core platforms: 1) the gap between the logical collective topology and the underneath hardware topology due to topology-unaware designs in most MPI libraries; 2) lacking efficient shared memory message delivering approaches; and 3) lacking a framework to tightly coordinate multi-layered collective algorithms together on distributed memory machines. To address these existing issues, we developed a kernel-assisted and topology-aware MPI collective communication framework.

Our kernel assisted collective component can successfully take advantage of new features in the operating system (OS) kernel, namely kernel assisted direct memory copy between process spaces, to greatly improve the performance of shared memory collective communication. These collective algorithms, directly based on kernel assisted direct memory copy, can maximize their bandwidth via eliminating intermediate memory copies and off-loading memory copy work-loads onto non-root processes. The cache-friendly design characteristics in our kernel-assisted MPI collective algorithms lead to more throughputs for real applications.

In addition, our distance-aware framework can provide an automatic mechanism to develop topology-aware MPI collective communication. Under this framework, process distance is calculated by underneath physical hardware distance. Automatically generated collective topologies can be dynamically mapped onto underlying hardware topologies, making distance-aware operations always deliver the highest performance no matter with any process placement, any underneath hardware architecture, and any run-time communicators. Compared with huge performance gains from our distance-aware framework, the distance calculation overhead is trivial: it is only calculated once during the communicator initialization and cached during the whole life of a communicator.

These techniques can be further extended onto multi-layered hierarchical collective communication on distributed memory machines, like multicore clusters. We developed a kernel-assisted topology-aware collective framework: HierKNEM, which enables efficient combinations of multiple layers of collective algorithms, to tackle collective communication on clusters of many-core compute nodes. The algorithms are built reusing modular combinations of existing collective algorithms (such as the *Tuned* and the KNEM collective components in Open MPI). The main contributions are: 1) propose an adaptive hierarchical collective framework to enable tight collaboration between the collective algorithms pertaining to different layers of the hierarchy, 2) combine offloading and pipelining techniques into the hierarchical framework to release leader processes from intra-node data movement, hence maximizing the overlap between inter- and intra-node communications, and 3) build internal collective topologies to form a mapping between the runtime process-core binding and the hardware features, which means stable collective performance independent of process placements. Experimental results demonstrate that, 1) our approach is immune to modifications of the underlying process-core binding; 2) it outperforms state-of-the-art MPI libraries (Open MPI, MPICH2, and MVAPICH2), demonstrating up to a 30x speedup in synthetic benchmarks, translating into up to a 3x acceleration for a parallel graph application (ASP); 3) it furthermore demonstrates a linear speedup with the increase of the number of cores per compute node, a paramount requirement for scalability on future many-core clusters.

6.2 Future Work

The technique of coordinating multi-layered collective algorithms together has shown more advantages than a simple concatenation of layered collective algorithms one by one. For example, our HierKNEM Broadcast greatly outperforms Open MPI's Hierarch and MVAPICH2 Broadcast. Even manually composing collective algorithms delivers great performance, which leaves a huge space for research about automatic collective algorithms' composition.

Another direction to extend this dissertation is to use hardware-based or NIC-based inter-node collectives to replace our software- and host-based inter-node data movement on the network layer. Off-loading inter-node data movement onto smart NICs can further decrease memory access pressure on memory banks and buses, speed up progressing intra-node collective, and reduce the interference between multi-level data movement. The kernel assisted direct memory copy can be tightly cooperated with other data movement techniques like Mellanox's fabric collective accelerate (FCA) engine or IBM's RDMA-based collectives. The flexibility of the plug-in/plug-out design in our framework allows for easy embedding of a faster and more efficient inter-node data movement module into our framework. Undoubtedly, a faster and more powerful inter-node data movement module will further boost the throughputs of our kernel-assisted and topology-aware collective algorithms.

Bibliography

Bibliography

[1] (1994). MPI: A Message Passing Interface Standard. http://www.mpi-forum.org/.

[2] (2003). Elan Programming Manual 5.1. http://staff.psc.edu/oneal/compaq/Elan_Programming_Manual_5.1.pdf.

[3] (2009). NetPIPE 3.7.2. http://bitspjoule.org/NetPIPE/.

[4] (2012). INFINIBAND TRADE ASSOCIATION. http://www.infinibandta.org/.

[5] (2012). Intel MPI benchmarks 3.2. http://software.intel.com/en-us/articles/intel-mpi-benchmarks/.

[6] (2012). KNEM: High-Performance Intra-Node MPI Communication. http://runtime.bordeaux.inria.fr/knem/.

[7] (2012). MPI: A Message-Passing Interface Standard Version 3.0. http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf.

[8] Alam, S., Barrett, R., Bast, M., Fahey, M. R., Kuehn, J., McCurdy, C., Rogers, J., Roth, P., Sankaran, R., Vetter, J. S., Worley, P., and Yu, W. (2008). Early evaluation of IBM BlueGene/P. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 23:1–23:12, Piscataway, NJ, USA. IEEE Press.

[9] Beecroft, J., Addison, D., Hewson, D., McLaren, M., Roweth, D., Petrini, F., and Nieplocha, J. (2005). QsNetII: Defining High-Performance Network Design. *IEEE Micro*, 25(4):34–47.

[10] Brightwell, R., Pedretti, K., and Hudson, T. (2008). SMARTMAP: operating system support for efficient data sharing among processes on a multi-core processor. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 25:1–25:12, Piscataway, NJ, USA. IEEE Press.

[11] Brightwell, R., Pedretti, K. T., Underwood, K. D., and Hudson, T. (2006). SeaStar Interconnect: Balanced Bandwidth for Scalable Performance. *IEEE Micro*, 26(3):41–57.

[12] Broquedis, F., Clet-Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S., and Namyst, R. (2010). hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications. In *Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, PDP '10, pages 180–186, Washington, DC, USA. IEEE Computer Society.

[13] Buntinas, D., Goglin, B., Goodell, D., Mercier, G., and Moreaud, S. (2009). Cache-Efficient, Intranode Large-Message MPI Communication with MPICH2-Nemesis. In *Proceedings of the 38th International Conference on Parallel Processing (ICPP-2009)*, pages 462 – 469, Vienna, Austria. IEEE Computer Society Press.

[14] Buntinas, D., Mercier, G., and Gropp, W. (2006). Design and Evaluation of Nemesis, a Scalable, Low-Latency, Message-Passing Communication Subsystem. In *Proceedings of the 6th IEEE International Symposium on Cluster Computing and the Grid*, CCGRID '06, pages 521–530, Washington, DC, USA. IEEE Computer Society.

[15] Chen, D., Eisley, N., Heidelberger, P., Senger, R., Sugawara, Y., Kumar, S., Salapura, V., Satterfield, D., Steinmacher-Burow, B., and Parker, J. (2012). The IBM Blue Gene/Q Interconnection Fabric. *IEEE Micro*, 32(1):32–43.

[16] Chen, H., Chen, W., Huang, J., Robert, B., and Kuhn, H. (2006). MPIPP: an automatic profile-guided parallel process placement toolset for SMP clusters and multiclusters. In *Proceedings of the 20th annual international conference on Supercomputing*, ICS '06, pages 353–360, New York, NY, USA. ACM.

[17] Coll, S., Duato, D., Petrini, F., and Mora, F. J. (2003). Scalable Hardware-Based Multicast Trees. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing (SC '03)*, pages 54–74, New York, NY, USA. ACM.

[18] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2001). *Introduction to Algorithms*. MIT Press and McGraw-Hill, second edition. Section 23.2: The algorithms of Kruskal and Prim.

[19] Davis, K., Hoisie, A., Johnson, G., Kerbyson, D. J., Lang, M., Pakin, S., and Petrini, F. (2004). A Performance and Scalability Analysis of the BlueGene/L Architecture. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, SC '04, pages 41–, Washington, DC, USA. IEEE Computer Society.

[20] Fagg, G. E., Bosilca, G., Pješivac-Grbović, J., Angskun, T., and Dongarra, J. (2006). Tuned: A flexible high performance collective communication component developed for Open MPI. In *Proccedings of DAPSYS'06*, pages 65–72, Innsbruck, Austria. Springer-Verlag.

[21] Faraj, A., Kumar, S., Smith, B., Mamidala, A., Gunnels, J., and Heidelberger, P. (2009). MPI collective communications on the blue gene/p supercomputer: algorithms and optimizations. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 489–490, New York, NY, USA. ACM.

[22] Ferreira, K. B., Bridges, P., and Brightwell, R. (2008). Characterizing application sensitivity to OS interference using kernel-level noise injection. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 19:1–19:12, Piscataway, NJ, USA. IEEE Press.

[23] Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R. H., Daniel, D. J., Graham, R. L., and Woodall, T. S. (2004). Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings of the 11th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 97–104, Budapest, Hungary.

[24] Geoffray, P., Prylli, L., and Tourancheau, B. (1999). BIP-SMP: high performance message passing over a cluster of commodity SMPs. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, Portland, OR.

[25] Goglin, B. and Moreaud, S. (2011). Dodging Non-uniform I/O Access in Hierarchical Collective Operations for Multicore Clusters. In *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '11, pages 788–794, Washington, DC, USA. IEEE Computer Society.

[26] Goglin, B. and Moreaud, S. (2012). KNEM: a Generic and Scalable Kernel-Assisted Intra-node MPI Communication Framework. *Journal of Parallel and Distributed Computing*. http://hal.inria.fr/hal-00731714.

[27] Graham, R. and Shipman, G. (2008). MPI Support for Multi-core Architectures: Optimized Shared Memory Collectives. In Lastovetsky, A., Kechadi, T., and Dongarra, J., editors, *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing*

Interface, volume 5205 of *Lecture Notes in Computer Science*, pages 130–140. Springer Berlin / Heidelberg.

[28] Graham, R., Venkata, M., Ladd, J., Shamis, P., Rabinovitz, I., Filipov, V., and Shainer, G. (2011). Cheetah: A Framework for Scalable Hierarchical Collective Operations. In *Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2011)*, pages 73–83.

[29] Gropp, W. (2002). MPICH2: A New Start for MPI Implementations. In *Proceedings of the 9th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 7–7, London, UK, UK. Springer-Verlag.

[30] Gropp, W., Lusk, E., Doss, N., and Skjellum, A. (1996). A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828.

[31] Huse, L. P. (1999). Collective Communication on Dedicated Clusters of Workstations. In *Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 469–476, London, UK, UK. Springer-Verlag.

[32] Hutter, J. and Iannuzzi, M. (2012). CPMD: CarParrinello Molecular Dynamics. http://www.cpmd.org/.

[33] Jeannot, E. and Mercier, G. (2010). Near-optimal placement of MPI processes on hierarchical NUMA architectures. In *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part II*, Euro-Par'10, pages 199–210, Berlin, Heidelberg. Springer-Verlag.

[34] Jin, H.-W. and Panda, D. K. (2005). Limic: Support for high-performance mpi intra-node communication on linux cluster. In *Proceedings of the 2005 International*

Conference on Parallel Processing, ICPP '05, pages 184–191, Washington, DC, USA. IEEE Computer Society.

[35] Kandalla, K., Subramoni, H., Vishnu, A., and Panda, D. (2010). Designing topology-aware collective communication algorithms for large scale InfiniBand clusters: Case studies with Scatter and Gather. In *Proceedings of the 2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW '10)*, pages 1–8.

[36] Karonis, N. T., de Supinski, B. R., Foster, I., Gropp, W., Lusk, E., and Bresnahan, J. (2000). Exploiting Hierarchy in Parallel Computer Networks to Optimize Collective Operation Performance. In *The 14th International Parallel and Distributed Processing Symposium*, pages 377–384.

[37] Karonis, N. T., Supinski, B. R. D., Foster, I. T., Gropp, W., and Lusk, E. L. (2002). A Multilevel Approach to Topology-Aware Collective Operations in Computational Grids. *Computing Research Repository*.

[38] Kielmann, T., Hofman, R. F. H., Bal, H. E., Plaat, A., and Bhoedjang, R. A. F. (1999). MagPIe: MPI's collective communication operations for clustered wide area systems. *Proceedings of the ACM SIGPLAN symposium on Principles and Practice of Parallel Programming(PPoPP'99)*, 34(8):131 – 140.

[39] Kumar, S., Sabharwal, Y., Garg, R., and Heidelberger, P. (2008). Optimization of All-to-All Communication on the Blue Gene/L Supercomputer. In *Proceedings of the 2008 37th International Conference on Parallel Processing*, ICPP '08, pages 320–329, Washington, DC, USA. IEEE Computer Society.

[40] Ma, T., Bosilca, G., Bouteiller, A., and Dongarra, J. (2012). HierKNEM: An Adaptive Framework for Kernel-Assisted and Topology-Aware Collective Communications on Many-core Clusters. In *Proceedings of the 2012 IEEE 26th International Parallel Distributed Processing Symposium (IPDPS)*, pages 970–982.

[41] Ma, T., Bosilca, G., Bouteiller, A., and Dongarra, J. J. (2010). Locality and topology aware intra-node communication among multicore CPUs. In *Proceedings of the 17th European MPI users' group meeting conference on Recent advances in the message passing interface*, EuroMPI'10, pages 265–274, Berlin, Heidelberg. Springer-Verlag.

[42] Ma, T., Bosilca, G., Bouteiller, A., Goglin, B., Squyres, J. M., and Dongarra, J. J. (2011a). Kernel Assisted Collective Intra-node MPI Communication Among Multi-core and Many-core CPUs. In *Proceedings of 2011 International Conference on Parallel Processing*, pages 532–541, Taipei, Taiwan.

[43] Ma, T., Herault, T., Bosilca, G., and Dongarra, J. J. (2011b). Process Distance-aware Adaptive MPI Collective Communications. In *Proceedings of 2011 IEEE International Conference on Cluster Computing*, pages 196–204, Austin, Texas. IEEE.

[44] Mamidala, A., Chai, L., Jin, H.-W., and Panda, D. (2006a). Efficient SMP-aware MPI-level broadcast over InfiniBand's hardware multicast. In *6th Workshop on Communication Architecture for Clusters (CAC) held in conjunction with the 20th International Parallel and Distributed Processing Symposium*.

[45] Mamidala, A., Vishnu, A., and Panda, D. (2006b). Efficient Shared Memory and RDMA Based Design for MPI_Allgather over InfiniBand. In *Proceedings of the 13th European PVM/MPI Users Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 4192 of *Lecture Notes in Computer Science*, pages 66–75. Springer Berlin /Heidelberg.

[46] Meuer, H. W., Strohmaier, E., Dongarra, J. J., and Simon, H. D. (2012). TOP500 Supercomputing Sites (2012). http://top500.org.

[47] Moody, A., Fernandez, J., Petrini, F., and Panda, D. K. (2003). Scalable NIC-based Reduction on Large-scale Clusters. SC '03, pages 59–, New York, NY, USA. ACM.

[48] Moreaud, S., Goglin, B., Goodell, D., and Namyst, R. (2010). Optimizing MPI Communication within large Multicore nodes with Kernel assistance. In *CAC 2010: The 10th Workshop on Communication Architecture for Clusters, held in conjunction with IPDPS 2010*, pages 1–7, Atlanta, GA. IEEE Computer Society Press.

[49] Pedretti, K. (2011). Cross-Process Memory Mapping. http://code.google.com/p/xpmem/.

[50] Petrini, F., Feng, W.-c., Hoisie, A., Coll, S., and Frachtenberg, E. (2001). The Quadrics Network (QsNet): High-Performance Clustering Technology. HOTI '01, pages 125–, Washington, DC, USA. IEEE Computer Society.

[51] Plaat, A., Bal, H. E., Hofman, R. F. H., and Kielmann, T. (2001). Sensitivity of parallel applications to large differences in bandwidth and latency in two-layer interconnects. *Future Generation Computer Systems*, 17(6):769 – 782.

[52] Qian, Y. and Afsahi, A. (2007). RDMA-based and SMP-aware Multi-port All-Gather on Multi-rail QsNet II SMP Clusters. In *Proceedings of the 2007 International Conference on Parallel Processing (ICPP 2007)*, page 48.

[53] Rajamony, R., Arimilli, L. B., and Gildea, K. (2011). PERCS: the IBM power7-IH high-performance computing system. *IBM Journal Research and Development*, 55(3):233–244.

[54] Sancho, J. C., Jokanovic, A., and Labarta, J. (2012). Reducing the impact of soft errors on fabric-based collective communications. In *Proceedings of the 2011 international conference on Parallel Processing - Volume 2*, Euro-Par 11, pages 262–271, Berlin, Heidelberg. Springer-Verlag.

[55] Solt, D. (2007). A profile based approach for topology aware MPI rank placement. http://www.tlc2.uh.edu/hpcc07/Schedule/speakers/hpcc_hp-mpi_solt.ppt.

[56] Squyres, J. M. and Lumsdaine, A. (2004). The Component Architecture of Open MPI: Enabling Third-Party Collective Algorithms. In *Proceedings of 18th ACM International Conference on Supercomputing, Workshop on Component Models and Systems for Grid Applications*, pages 167–185. Springer.

[57] Thakur, R. and Gropp, W. (2003). Improving the Performance of Collective Operations in MPICH. In *Proceedings of the 10th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 2840 of *Lecture Notes in Computer Science*, pages 257–267. Springer Berlin / Heidelberg.

[58] Träff, J. L. (2002). Implementing the MPI process topology mechanism. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, Supercomputing '02, pages 1–14, Los Alamitos, CA, USA. IEEE Computer Society Press.

[59] Vaidyanathan, K., Chai, L., Huang, W., and Panda, D. (2007). Efficient asynchronous memory copy operations on multi-core systems and I/OAT. In *Proceedings of 2007 IEEE International Conference on Cluster Computing*, pages 159 –168. IEEE.

[60] Vetter, J. S. (2012). mpiP: Lightweight, Scalable MPI Profiling. http://mpip.sourceforge.net/.

[61] Yeoh, C. (2010). Cross Memory Attach. http://lwn.net/Articles/405284/.

[62] Yu, W., Buntinas, D., and Panda, D. (2003). High performance and reliable NIC-based multicast over Myrinet/GM-2. In *Proceedings of the 2003 International Conference on Parallel Processing (ICPP 2003)*, pages 197 –204.

[63] Yu, W., Panda, D., and Buntinas, D. (2004). Scalable, high-performance NIC-based all-to-all broadcast over Myrinet/GM. In *Proceedings of 2004 IEEE International Conference on Cluster Computing*, pages 125 – 134.

[64] Zhu, H., Goodell, D., Gropp, W., and Thakur, R. (2009). Hierarchical Collectives in MPICH2. In Ropo, M., Westerholm, J., and Dongarra, J., editors, *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 5759 of *Lecture Notes in Computer Science*, pages 325–326. Springer Berlin / Heidelberg.

Vita

Teng Ma was born in Nanjing, China. He received his high school education from the Nanjing No.12 High School. He obtained a Bachelor's degree and a Master's degree in Computer Science and Engineering from Southeast University in Nanjing, China, in 2003 and 2006, respectively.

In 2006, he began his PH.D. study at the Computer Science Department at the University of Tennessee, Knoxville. Meanwhile, he worked as a Research Assistant at the Innovative Computing Laboratory (ICL) under the guidance of Dr. Jack Dongarra and Dr. George Bosilca. His research interests focused on communication libraries, e.g., Message Passing Interface (MPI). While at ICL, he was an active developer in the Open MPI project. During his PH.D. study, he published 7 papers in conferences and journals. His paper "HierKNEM: An Adaptive Framework for Kernel-Assisted and Topology-Aware Collective Communications on Many-core Clusters" was awarded the best paper award in the 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS '12). He also actively participated in collaborative research projects. He completed a summer internship at Los Alamos National Laboratory in 2009 and another at NetApp Inc. in 2012. Teng Ma was also very active at serving in the high performance computing communities: he served as a student volunteer in ACM/IEEE Supercomputing conferences from 2008 to 2011. He will work as a member of the technical staff at NetApp Inc. after graduation.