Improving Communication Performance through Topology and Congestion Awareness in HPC Systems

by

Seyed Hessamedin Mirsadeghi

A thesis submitted to the
Department of Electrical and Computer Engineering
in conformity with the requirements for
the degree of Doctor of Philosophy

Queen's University
Kingston, Ontario, Canada
July 2017

Abstract

High-Performance Computing (HPC) represents the flagship domain in providing high-end computing capabilities that play a critical role in helping humanity solve its hardest problems. Ranging from answering profound questions about the universe to finding a cure for cancer, HPC applications span nearly every aspect of our life. The impressive power of HPC systems comes mainly from the massive number of processors—in the order of millions—that they provide. The efficiency of communications among these processors is the main bottleneck in the overall performance of HPC systems.

This dissertation presents new algorithms for improving the communication performance in HPC systems by exploiting the topology information. We propose a parallel topology- and routing-aware mapping heuristic and a refinement algorithm that improves the communication performance by achieving a lower congestion across the network links. Our experimental results with 4,096 processors show that the proposed approach can provide more than 60% improvement in various mapping metrics compared to an initial in-order mapping of processes. Communication time is also improved by up to 50%. We also propose four topology-aware mapping heuristics designed specifically for collective communications in the Message Passing Interface (MPI). The heuristics provide a better match between the collective communication algorithm and the physical topology of the system, and decrease the communication latency by up to 78%.

Furthermore, we expand topology-aware communications into the scope of accelerated computing. Using accelerators—especially Graphics Processing Units (GPUs)—to speed up

certain types of computations plays an increasingly important role in HPC. We present a unified framework for topology-aware process mapping and GPU assignment in multi-GPU systems. Our experimental results on two clusters with 64 GPUs show that the proposed approach improves communication performance by up to 91%.

Finally, we present a novel distributed algorithm that uses the process topology information to design optimized communication schedules for MPI neighborhood collectives. The proposed algorithm finds the common neighborhoods in a distributed graph topology and exploits them as an opportunity to improve the communication performance through message combining. The optimized schedules reduce the communication latency of MPI neighborhood collectives by more than 50%.

Acknowledgments

To my father, Hossein, whose wisdom has been the light of my life

To my mother, Masoumeh, whose love makes this life worth living

Fist of all, I would like to thank my parents for their unconditional love and support. No matter how hard I try, I will not be able to find the right words to express how deeply I feel indebted to them. I hope this dissertation can be a means to bring a smile on their face. I would also like to thank my beloved sister, Mahsa, who has always been an example for me to look up to and follow.

Next, I would like to thank my supervisor, Prof. Ahmad Afsahi, for the valuable support, feedback, and motivation that he granted me in conducting the research for this dissertation. Prof. Afsahi's dedication to high-quality research and his hard-working attitude will always be an example for me to look up to. I would also like to express my gratitude to the members of my dissertation examining committee for their valuable feedback regarding the research conducted in this dissertation. In addition, I would like to thank Dr. Pavan Balaji of the U.S. Argonne National Laboratory and Dr. Jesper Träff of the Vienna University of Technology for their insightful comments and their openness to share their world-class knowledge with me through the multiple discussions that we had.

I also thank the Natural Science and Engineering Research Council of Canada (NSERC), the Electrical and Computer Engineering (ECE) Department of Queens University as well as the school of graduate studies for the financial support of this research. A very special gratitude also goes to Ms. Debra Fraser who I think is the best graduate assistant that one

could ever wish for. Thank you for always being so welcoming, supportive and nice.

I am grateful to Compute Canada and the HPC Advisory Council for providing me with the supercomputing resources required to conduct the research in this dissertation. Special thanks to Dr. Scott Northrup, Dr. Daniel Gruner, and the rest of the team at the SciNet supercomputing center in Toronto for their valuable support with the GPC supercomputer. My gratitude also goes to Dr. Maxime Boissonneault from Calcul Québec for his support with the Helios supercomputer. I am also grateful to Mr. Pak Lui for his support with the computing resources at the HPC Advisory Council.

Finally, I would like to thank all my colleagues at the Parallel Processing Research Laboratory, Dr. Judicael Zounmevo, Dr. Ryan Grant, Dr. Reza Zamani, Grigory Inozemtsev, Iman Faraji, Kashual Kumar, Mahdieh Ghazimirsaeed, Ramapriya Balasubramaniam, and Mac Fregeau. Additional thanks to Judicael from whom I learned a lot; thanks for not getting tired of answering all the questions that I asked you. Special thanks to Iman for all those constructive discussions, the team work, and your unconditional help whenever I needed it.

Statement of Collaboration

The work in Chapter 5 was performed collaboratively with Mr. Iman Faraji. The theoretical design of the unified three-phase framework for the topology-aware process mapping and GPU assignment was developed by me in collaboration with Mr. Iman Faraji regarding the GPU communications technologies. The development of the intra-node topology-aware GPU assignment in Phase 3 of the framework was done by Mr. Iman Faraji in collaboration with myself regarding the mapping algorithms/libraries. The congestion-based heuristic was developed by me. Microbenchmarks were developed in a joint effort. I developed the core structure of the microbenchmarks, and for the general-purpose processor communications only; Mr. Faraji extended them to include the GPU devices too. Microbenchmark experiments and results analysis were conducted collaboratively, and Mr. Faraji was the main person who launched and monitored the jobs on the GPU clusters. Also, the experiments with the HOOMD-Blue application were performed by Mr. Faraji; the analysis of the corresponding results were performed jointly with myself.

Table of Contents

List of Tables

List of Figures

Glossary

AMD Advanced Micro Devices 3
API Application Programming Interface 11

BBMH Binomial Broadcast Mapping Heuristic 86
BFT Breadth First Traversal 85
BGMH Binomial Gather Mapping Heuristic 87

CA Channel Adapter 22
CPU Central Processing Unit 103

DCMF Deep Computing Message Framework 161
DFT Depth First Traversal 85

FLOPS FLoating Operations Per Second 2

GPC General Purpose Cluster 51
GPU Graphics Processing Unit 5

HB Hop-Bytes 44
HCA Host Channel Adapter 23
HPC High Performance Computing 1
HT HyperTransport 3

IB InfiniBand 21
IBA InfiniBand Architecture 21
IBTA InfiniBand Trade Association 22
iWARP Internet Wide Area RDMA Protocol 25

LID Local Identifier 22

MC Maximum Congestion 44
MPI Message Passing Interface 2
MPP Massively Parallel Processors 8

MWM Maximum Weighted Matching 139

NUMA Non-Uniform Memory Access 3
NZCA Non-Zero Congestion Average 44
NZCV Non-Zero Congestion Variance 44

OFED OpenFabrics Enterprise Distribution 34
OS Operating System 22

PE Processing Element 103
PGAS Partitioned Global Address Space 9
PTRAM Parallel Topology- and Routing-Aware Mapping 27

QAP Quadratic Assignment Problem 35
QPI QuickPath Interconnect 3

rCUDA remote CUDA 109
RDMA Remote Direct Memory Access 23
RDMH Recursive Doubling Mapping Heuristic 83
RMA Remote Memory Access 18
RMH Ring Mapping Heuristic 84

SMP Symmetric Multi-Processor 8

TCA Target Channel Adapter 23

Chapter 1

Introduction

Mathematical models and numerical simulations play a pivotal role in today's science and engineering by enabling the study of complex phenomena that would be too expensive or dangerous to study by direct experimentation. However, these techniques require such an enormous computational capability that significantly exceeds what is provided by ordinary computers. In this regard, High-Performance Computing (HPC) or supercomputing is the broad domain in which novel computing architectures and techniques are investigated to provide high-end computing power. Thanks to HPC, simulation is now being used as an integral part of manufacturing, design, and decision-making processes, and as a fundamental tool for scientific research. Weather prediction, energy research, design of vehicles and aircraft, finding oil, computational chemistry, computational medicine, understanding the evolution of the universe, and online fraud detection are just a few representative examples of domains for the application of HPC.

It is well established that parallel processing is the key approach to satisfy the ever increasing demands for more computational power in HPC. For this purpose, an application is decomposed into a number of tasks that can be executed simultaneously. Each task is assigned to a process/thread which is in turn executed by one of many processing elements. Accordingly, HPC systems are designed and engineered to provide substantial computational power by supplying an ever-increasing number of processors that enables the parallel

execution of an application's tasks. Such increases in computational power have successfully brought HPC to the Petascale Era with systems capable of performing operations at the scale of 10^{15} FLoating-point Operations Per Second (FLOPS). Presently, the top publicly known supercomputers in the world such as Sunway TaihuLight, Tianhe-2, and Sequoia [117] provide users with millions of processors. However, the size and complexity of the problems in HPC demand such a high computational capability that not only saturates the impressive power of the top supercomputers in the world, but also goes beyond it. Thus, the HPC community is striving to expand the computational boundaries even further by moving toward the Exascale Era (10^{18} FLOPS).

The efficient utilization of the abundant hardware resources in HPC systems hinges upon having software layers that can provide an easy-to-use abstract model of the parallel system. This issue becomes particularly important when we consider the complexity and the large scale at which HPC systems are deployed. In this regard, a key challenge is to deliver as much performance to the upper-layer users as the system has to offer. A well-known bottleneck to achieving this goal is the performance of communications within a parallel system. In parallel applications, a significant amount of communication is required among the processing nodes to coordinate the concurrent execution of the job. Therefore, the efficiency of inter-process communications is a key factor in the overall performance of HPC systems. In addition, the rapid increase in the number of processing elements and the speed gap between communication and computation have made communication the performance bottleneck. As a result, decreasing communication overheads continues to be an important paradigm of performance enhancement in HPC systems.

In this dissertation, we discuss how topology information can be utilized to improve the performance of communications in HPC. Topology information includes the physical topology of the target system, as well as the virtual process topology of the parallel application. Our main focus will be on the Message Passing Interface (MPI) [85] as it represents the most successful and widespread parallel programming paradigm in HPC.

1.1 Motivation

As we move toward the Exascale Era, both the number of nodes in a system and the number of cores within each node are increasing. HPC systems are becoming more and more complex, introducing increasing levels of heterogeneity in communication channels. Inter-node communications that traverse the dedicated network for interconnection of a large number of nodes are generally slower than the intra-node communications that use conventional read and write operations on memory that is shared by a modest number of processors within the same node. We also see various levels of communication performance within the network itself or inside a single node. Messages that pass across a larger number of links and switches suffer more in terms of latency. Similar issues exist within modern multicore nodes due to Non-Uniform Memory Access (NUMA) effects and multiple levels of cache hierarchies. For instance, communications among the cores within the same chip in a single physical socket (intra-socket communications) are considerably faster than communications among those that belong to different sockets (inter-socket communications). The former can take advantage of high-speed shared L2/L3 caches, whereas the latter has to pass through (relatively) slower links such as Intel QuickPath Interconnect (QPI) [4] or AMD HyperTransport (HT) [49]. This challenges the assumptions made by commonly used HPC programming paradigms, threatening the performance efficiency of new systems. Most parallel programming paradigms—in particular MPI—have been designed in a topology-agnostic fashion that assumes a uniform communication performance among the system processors.

The variation in performance across different communication channels within modern HPC systems makes it necessary to consider topology information at higher levels such as in MPI. Topology awareness can help to achieve more efficient utilization of the underlying communication channels, leading to higher-performance communications at the application level. In particular, topology-aware process mapping has been shown to be a promising and necessary approach to improve communication performance in modern large-scale systems

[7, 101, 45, 11]. Mapping processes based on the communication pattern of an application and the physical topology of the underlying system can prevent communication over slower channels and message transmission over long paths.

Topology-aware mapping can also reduce the amount of congestion at various levels of the system hierarchy. Congestion is known to be an important factor that can adversely affect communication performance. It has been shown that communications can suffer up to 6.5 times lower in bandwidth, and 5 times higher in latency in existing InfiniBand [51] installations because of congestion [44]. Current trends suggest that network congestion will worsen in future systems. The increase in the size and scale of modern systems increases the amount of traffic that traverses the network. Although network devices are scaled up to meet the requirements of large scale systems, they still lag behind the rate at which the number of processing nodes are increased [114]. Therefore, we will see under-provisioned networks in exascale systems, leading to higher amounts of congestion. In addition, the advent of many-core processor chips exacerbates the situation by increasing the contention at each interface to the network as well as within the network [75]. Also, as systems scale up, network diameter typically increases, which causes messages to traverse more links to reach their destinations. Consequently, each link will be shared among a higher number of messages, provoking more congestion [14].

Although various topology-aware mapping techniques have been proposed by different researchers, the NP-hard nature of the problem on one hand, and the rapid increase in the size and complexity of modern systems on the other hand have kept the issue as an open problem which demands further study. In particular, designing heuristics that can provide high-quality mapping results and yet show good scalability in computational complexity is of utmost importance for large-scale systems. Topology awareness can also be used to provide a better match between the communication algorithms used internally by the MPI libraries and the physical topology of the target systems.

In addition, high-performance computing is evolving towards hybrid accelerator-based

architectures. Particularly, using Graphics Processing Unit (GPU) to accelerate certain computations of an application is becoming the mainstream in HPC. This is particularly true in the context of recent deep learning applications that have brought a new widespread attention to HPC. Such hybrid systems introduce new challenges and complexities with respect to topology awareness and communications performance.

The ever-increasing scale of supercomputers also demands new parallel algorithms at the application level that can scale with increases in the number of cores. In this regard, new algorithms attempt to reduce or avoid global communications among the parallel processes of an application. Instead, they attempt to limit the communications to a sparse neighborhood of each process. In order to provide support for such inherently more scalable algorithms, MPI-3.0 [85] added neighborhood collective communications into the process topology interface of the standard. Neighborhood collectives provide another opportunity to optimize communications by exploiting topology information.

1.2 Problem Statement

This dissertation addresses the following list of questions.

1. How can we optimize process placement in large-scale HPC systems with respect to the congestion that is incurred across the communication channels? How can we do this without adding excessive overhead? What are the missing components of current topology-aware mapping frameworks and how can they be resolved?

2. How can topology-aware mapping be used to fine-tune collective communications in MPI? More specifically, how can we design fast mapping algorithms that are specifically tailored for certain collective communication patterns?

3. How can topology-aware mapping help to improve communication performance in hybrid GPU clusters? In particular, how can we design a framework so as to optimize

process-to-node mapping and GPU-to-process assignment across a system consisting of multicore multi-GPU nodes?

4. How can we improve the performance of the sparse neighborhood communications that have recently been added to MPI? How can we extract useful information from a distributed process topology and exploit it to design nontrivial communication schedules for neighborhood communications in MPI?

1.3 Contributions

In Chapter 3, we propose a parallel mapping heuristic and a parallel refinement algorithm that exploit topology and routing information to decrease congestion within the network. For InfiniBand clusters in particular, we take into account the static assignment of routes across the links. The heuristic attempts to minimize a hybrid metric that is used to evaluate candidate mappings from multiple aspects. The refinement algorithm attempts to directly reduce maximum congestion by refining the mapping output from the greedy heuristic. We take advantage of parallelism in the design and implementation of our proposed algorithms. We believe parallelism is the key for having a truly scalable topology-aware mapping approach in current and future HPC systems.

In Chapter 4, we propose four fine-tuned mapping heuristics for various communication patterns and algorithms commonly used in MPI allgather collective communication. The heuristics provide a better match between the collective communication pattern and the topology of the target system, and outperform generic approaches in terms of solution quality and overhead.

In Chapter 5, we take topology awareness into the domain of heterogeneous GPU clusters and propose a mapping approach for GPU clusters. Our proposed approach provides a unified methodology for topology-aware process-to-core mapping and GPU-to-process assignment in multicore multi-GPU clusters.

In Chapter 6, we propose a distributed algorithm to design optimized communication patterns and schedules for MPI neighborhood collectives. More specifically, we show how to find the common neighborhoods in a generic distributed process topology graph, and exploit them to optimize communication performance through message combining.

1.4 Organization of Thesis

The remainder of this dissertation is organized as follows. In Chapter 2, we provide a brief overview of the background material related to this dissertation. In Chapter 3, we discuss our proposed algorithms for parallel topology- and routing-aware process mapping. Chapter 4 presents our proposed mapping heuristics for various communication patterns used for allgather collective communication. Chapter 5 is devoted to our proposed framework for topology-aware process mapping and GPU assignment in GPU clusters. In Chapter 6, we explain our proposed approach for designing optimized communication schedules for MPI neighborhood collective communications. Finally, in Chapter 7, we present concluding remarks and potential directions for future work.

Chapter 2

Background

2.1 Parallel Computers

There are various architectures based on which parallel computers are built. Among them, clusters have gained a higher level of acceptance due to multiple benefits they provide. These include a high performance-cost ratio, flexibility in configuration, wide range of available software, etc. At the time of writing this document, 86.4% of the top 500 supercomputers in the world are clusters [117]. An HPC cluster consists of a number of independent commodity compute nodes connected to each other by a high-performance interconnect. Each node itself can have multiple processors in a Symmetric Multiprocessor (SMP) or NUMA configuration. In an SMP configuration, access performance to all parts of the system memory is the same for all the processors, whereas in a NUMA configuration the access performance is non-uniform. The reason is that in a NUMA architecture the system memory is partitioned into multiple portions each of which is closer to a certain group of processors (see Fig. 3.1(b) for an example). Thus, each processor can access its own local memory faster than the remote memory that is local to other processors. Massively Parallel Processors (MPP) are another important class of parallel computers. MPP systems consist of a large number of computing elements connected to each other by a custom-designed interconnect. MPPs are characterized by their use of proprietary components such as proprietary operating

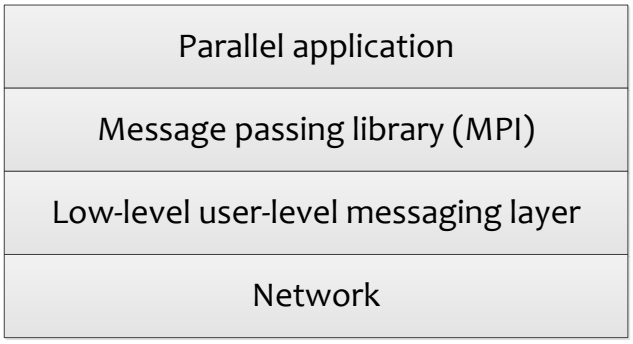


Figure 2.1: A schematic diagram showing where MPI stands in a parallel system

systems, special hardware, dedicated network for certain communications, etc. Therefore, MPPs most often provide a higher level of computational power compared to commodity clusters, but are at the same time much more costly to acquire.

2.2 Message Passing Interface

The Message Passing Interface (MPI), is the de facto standard for parallel programming in HPC, and is maintained by the MPI Forum [85]. Like every other programming model, MPI provides an interface between the high-level application and the lower-level system architecture. As shown in Fig. 2.1, MPI decouples parallel applications from the underlying communication layers. Originally, MPI was designed for distributed-memory programming, with the first version of the standard released in 1994. Since then, MPI has gradually (and successfully) evolved into a comprehensive parallel programming *system* that provides support for various models including Partitioned Global Address Space (PGAS) [96] and shared memory. The current version 3.1 of the standard was released in June 2015.

MPI provides a message passing library interface specification that can be used by both developers and users of message passing libraries. It also acts as the lower-level mechanism for implementing other higher-level parallel programming models [110, 24] such as PGAS languages [123]. Note that MPI itself is not a message passing library implementation, but

rather the specification of what such a library should be to provide portability, scalability, and efficiency. There exist multiple implementations for MPI, including both commercial and non-commercial versions. MPICH [86], MVAPICH [87] and Open MPI [92] represent three of the most popular, non-commercial, and open-source MPI libraries. In particular, MPICH deserves a specific recognition as it forms the basis for many other implementations of MPI. For instance, MVAPICH2 is an MPICH derivative tailored to exploit the features provided by well-known high-performance interconnects that are commonly used in HPC clusters.

At its core, MPI provides a means to achieve high-performance communication among the parallel processes of an application. Communication is realized through explicit movement of data from the address space of one process to the address space of another[1]. In the following sections, we review the key concepts related to communications in MPI.

2.2.1 Groups and Communicators

MPI uses the concept of *groups* and *communicators* to define the scope and context of all communications. A group defines an ordered collection of processes based on which each process is assigned a *rank* between zero and $N-1$, where N denotes the number of processes. Process ranks provide a mechanism to uniquely identify each process within a group. MPI groups are used within communicators to designate the participants in each communication. In addition to a group instance, each MPI communicator contains a communication context which provides the ability to have separate communication universes. A message sent in one context cannot be received in another context. Consequently, all MPI communication operations use a communicator to describe the participants and the context of communications. The standard defines a predefined communicator `MPI_COMM_WORLD` that includes all the processes of an application. The standard provides various Application

[1]It is worth noting that the standard does not mandate an MPI process to be an operating system process. MPI processes are implementation-dependent objects. Thus, as far as the standard is concerned, two MPI processes can share the same operating system process.

Programming Interfaces (APIs) that can be used to duplicate a communicator or create sub-communicators. It is worth noting that a process can be a member of more than one communicators, and will have a separate rank with respect to each.

2.2.2 Point-to-Point Communications

MPI provides support for different types of communications. *Point-to-point* or *two-sided* communications represent the basic communication mechanism of MPI that allows for sending and receiving messages between two individual processes. These communications use send/receive semantics which require explicit involvement of both source and destination. The source process issues a send operation for which a matching receive must be posted by the destination process.

The standard defines blocking and nonblocking versions of send/receive operations. A blocking send operation returns only when it is safe to reuse (modify) the send buffer. Safety here means that any changes to the buffer will not impact the content of the data that was passed to the send operation. However, it does not necessarily mean that the data has been received by the receiving side, or even started to be sent to it. The implementation might simply use an intermediate system buffer to copy the data passed to the send operation and thus safely return from a blocking send operation. A blocking receive operation returns only when the data is received at the application receive buffer.

On the other hand, a nonblocking send/receive operation returns immediately and leaves the responsibility for the safety of buffer modification to the programmer. These type of operations provide opportunity for communication/computation overlap where the process issues only a request to send/receive a message and continues its remaining computations. With appropriate support (e.g., network offloading mechanisms [37]), the communication request can proceed transparently without interrupting the ongoing computations. The standard defines a family of operations (`test` and `wait`) to test/block for/until the completion of nonblocking operations.

2.2.3 Collective Communications

Collective communications provide an abstraction for communications among a group of processes rather than just two. MPI defines multiple collective communication operations such as barrier, broadcast, allgather, alltoall, allreduce, etc. The barrier operation provides support for explicit synchronization across the processes, whereas the other operations enable a certain type of collective data exchange among the processes. Collective communications are widely used in parallel applications [100] because they provide a convenient, portable, and yet highly optimized way to conduct one-to-many and many-to-many communications. Consequently, MPI collective communications performance has a substantial impact on the overall performance of parallel applications. Accordingly, an extensive body of research has been devoted to optimize the performance of collective communications in MPI [2, 29, 66, 70, 116].

MPI collectives are often implemented as a series of point-to-point communications. Such implementations are also known as unicast-based algorithms which distinguishes them from hardware-based implementations that exploit special hardware supports such as hardware multicast. A common feature among unicast-based collective algorithms is that the communication is scheduled over a sequence of stages the union of which provides the desired collective data movement. In each stage, communication happens among a particular permutation of source-destination processes. The permutations used in each stage by the collective algorithm make up the internal communication pattern of the corresponding collective operation. Different collective communication algorithms might very well use different permutations, resulting in different internal communication patterns for a single collective.

Various algorithms have been proposed in the literature to decompose collectives into a corresponding set of point-to-point communications. In practice, MPI libraries make use of a combination of such algorithms and choose one of them based on different parameters

such as message and communicator size. A comprehensive set of such algorithms for various MPI collective operations (such as broadcast, allgather, alltoall, etc.) can be found in the work done by Thakur et al. [116]. We refer to these algorithms as *generic* algorithms. These generic algorithms provide the basis for many other tuned collective communication algorithms proposed in the literature [59, 70, 78]. In the remainder of this section, we briefly describe the commonly used algorithms in an `MPI_Allgather` operation as it is necessary for Chapter 4 of this dissertation.

Commonly used algorithms in `MPI_Allgather`

MPI allgather is a many-to-many collective communication where each process gathers data from every other process in the communicator. An allgather communication can be equivalently considered as an all-broadcast communication where each process has a message that has to be transmitted to every other process. The allgather operation is a data-intensive collective communication that can contribute significantly to the communication time of an application. It can also used as a building block for other collective operations such as broadcast and allreduce.

Non-hierarchical algorithms Previously, collective algorithms have been designed in a non-hierarchical approach for flat systems and executed across all processes in the system. For allgather in particular, *recursive doubling* and *ring* [116] are two of the most commonly used algorithms. Recursive doubling consists of $\log_2 N$ stages, where N denotes the total number of processes in the communicator. At each stage s, where $s = 0, 1, \ldots, \log_2 N - 1$, rank i exchanges data with rank $i \oplus 2^s$, where $\oplus$ represents the binary XOR operator. Consequently, each rank such as i exchanges messages with ranks $i \oplus 1, i \oplus 2, i \oplus 4, \ldots, i \oplus \frac{n}{2}$ throughout the stages of the recursive doubling algorithm.

Fig. 2.2 shows an example for the recursive doubling pattern with 8 processes. There exist three communication stages distinguished from each other by three different colors.

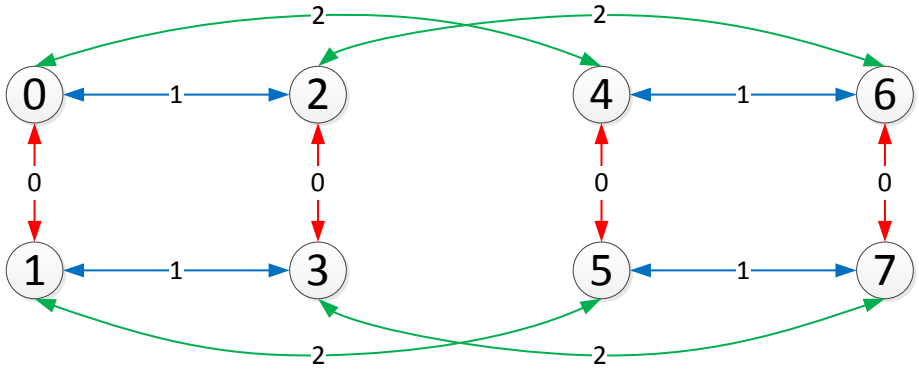
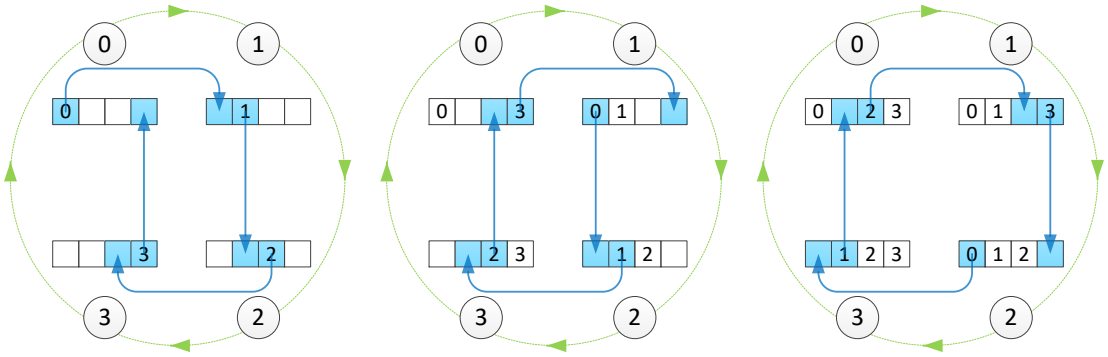


Figure 2.2: The recursive doubling communication pattern with 8 processes. There exist 3 communication stages: stage 0 (red), stage 1 (blue), and stage 2 (green).

The numbers on each edge denote the specific stage number. For instance, rank 0 and rank 1 exchange their input buffers in stage 0. In stage 1, rank 0 and rank 2 exchange their input buffers as well as the data they received in the previous stage, and so on. Note that the volume of the exchanged messages is doubled at each stage of the algorithm. It is also worth mentioning that recursive doubling is mainly used for a power-of-two number of processes.

In the ring algorithm, all processes are organized into a logical ring in order of their ranks. At each stage, rank i receives a message from rank $i-1$ and sends a message to rank $i+1$. In the first stage, rank i sends its own data to rank $i+1$. In the following stages, rank i forwards to rank $i+1$ the data it received from rank $i-1$ in the previous stage. With N processes, the algorithm runs for $N-1$ stages. Figure 2.3 shows an example with 4 processes. It shows the state of the output buffer of each process at each communication stage. The data corresponding to each process has been represented by its rank, and the blue arrows designate the source-destination buffer offsets that are used at each communication stage.

The main benefit of the recursive doubling algorithm comes from its logarithmic $\log_2 N$ number of stages. This makes recursive doubling a great choice when communication performance is dominated by the startup latencies that are associated with preparing the message and injecting it into the network. Thus, MPI libraries use the recursive doubling algorithm



(a) Stage 0 (b) Stage 1 (c) Stage 2

Figure 2.3: The ring communication pattern with 4 processes. There exist 3 communication stages with the same source-destination processes at each stage (the green ring).

for allgather communications that involve smaller messages. For larger messages on the other hand, MPI libraries mainly use the ring algorithm. The reason is that the ring algorithm tends to cause a lower congestion due to its more localized communication pattern. Thus, despite its larger number of stages ($N-1$), it can deliver a better performance than recursive doubling for larger messages because larger messages are more sensitive to bandwidth and congestion rather than startup latencies. However, as we will discuss in Chapter 4, it is important to take into account the system topology and the process placements in order to achieve the desired performance from a collective communication algorithm.

Hierarchical algorithms Hierarchical algorithms are executed across the nodes/sockets rather than the processes to take advantage of hierarchical communication costs. Typically, for each node, a communicator is created to contain all the processes residing on that node. One process on each node is selected as the leader of that node. The allgather algorithm will then proceed in three phases:

1. gathering intra-node messages into node leaders
2. exchanging the gathered data between all the node leaders, using a recursive doubling/ring-based allgather algorithm

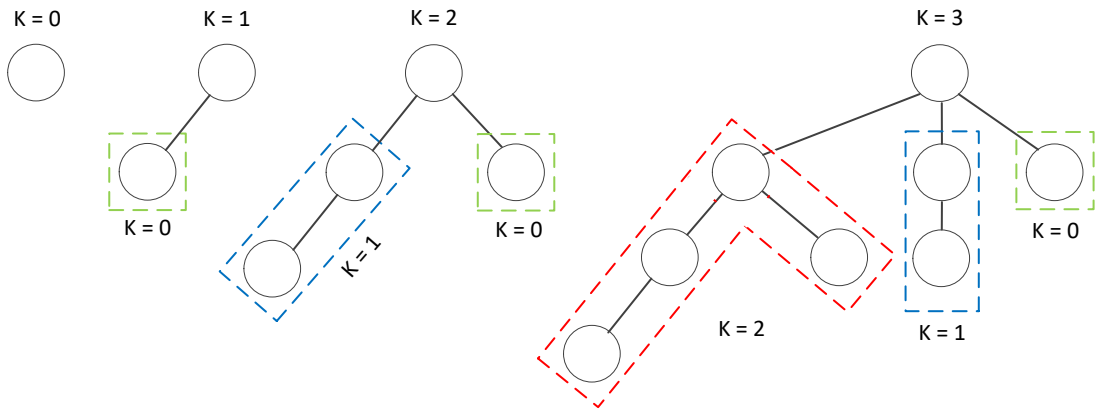


Figure 2.4: Binomial trees of order $k = 0, 1, 2, 3$. The dashed boxes show how lower order trees are recursively used to build the higher order ones.

3. broadcasting the data from each leader process to its local intra-node ranks

The gather and broadcast in phase 1 and 3 might use a direct *linear* pattern or an indirect *non-linear* algorithmic design. In the linear design, all ranks use shared memory to directly send (receive) data to (from) the root process, whereas in the non-linear design, message transmissions follow the particular communication pattern used by the gather (broadcast) algorithm. In this regard, binomial tree is one of the well-known communication patterns used for the gather and broadcast operations [116].

A binomial tree is defined with respect to an order k. The height of the tree is equal to the order k, and the total number of nodes in the tree is equal to 2^k. A binomial tree of order k can be defined recursively as follows:

- The binomial tree of order 0 consists of a single node.
- The binomial tree of order $k > 0$ consists of a root node with k children that are binomial subtrees of orders $k-1, k-2, \ldots, 1, 0$, respectively.

Figure 2.4 shows the binomial trees of order 0, 1, 2, and 3. The trees have been colored to show how lower order trees are used as the child nodes to build the trees of higher orders.

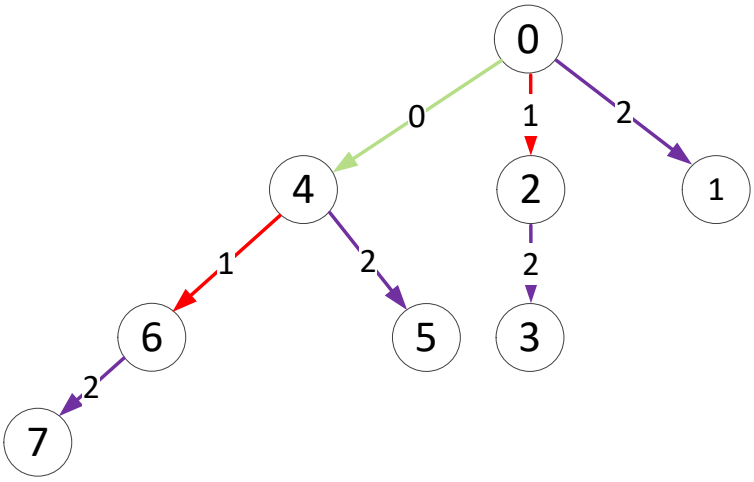


Figure 2.5: Binomial broadcast with 8 processes. Different edge colors have been used to distinguish among different communication stages. The label on each edge designates the specific stage of each communication.

Using a binomial tree for broadcast maximizes the number of processes that contribute/help in sending the message to all destinations. This results in a lower number of communication stages which will in turn decrease the total latency of broadcast. With N processes, the total number of communication stages will be $\log_2 N$. Figure 2.5 shows an example of a binomial broadcast with 8 processes and rank 0 as the root. The edge labels represent the corresponding stage of each communication. Once a process receives the broadcast message, it acts as the root of a broadcast in its own subtree and keeps on sending the message to all its children until the last stage of communications. This is in contrast to a binary (k-ary) tree in which each root process will send only two (k) messages and then remain idle, regardless of the total number of processes. Binomial tree is also used for a gather operation. The pattern will be same as the binomial broadcast used in reverse (bottom-up). Each process gathers the messages from all of its child processes and sends the resulting message to its parent process.

2.2.4 One-Sided Communications

In one-sided communication, a process (called origin) can send/receive a message to/from another process (called target) without involving it in the communication process. The origin process provides all required communication parameters for both the sending and receiving sides. This type of communication is also referred to as Remote Memory Access (RMA) since it allows one process to remotely access the memory of another process. We do not use one-sided communications in this dissertation, hence we refrain from discussing any more details about MPI RMA.

2.2.5 MPI Topology Interface

MPI provides a mechanism to define a logical topology for the set processes of an application. The process topology is attached as an additional and optional attribute to a corresponding communicator. In MPI terminology, process topology is also referred to as the *virtual topology* to avoid confusion with the system physical topology. Virtual topology provides a convenient naming mechanism for the set of processes in a communicator, but more importantly, it can be used to express a certain communication pattern among the processes. Information about the communication pattern of an application can later be exploited to improve the overall performance in different ways. For instance, the runtime system might make use of the virtual topology information to conduct topology-aware mapping optimization.

The topology interface of MPI provides two main interfaces for describing process topologies: (1) the *graph* interface, and (2) the *Cartesian* interface. The graph interface provides the most generic way of defining process topologies in MPI. Each vertex of the graph represents a process and the edges describe the communication relationships among the processes. The MPI-1.0 standard (released in 1994) defines a general graph constructor (`MPI_Graph_create()`) that can be used to build a directed and unweighted topology graph.

Unfortunately, this interface has a number of substantial shortcomings that make it almost useless [6, 9, 118]. Accordingly, the MPI-2.2 standard (released in 2009) defines the *distributed graph topology* functions which provide a more scalable and informative topology interface. In the new interface, the graph topology is defined in a fully distributed fashion, where each process describes only a fraction of the whole communication graph. It is also possible to assign relative *weights* to the communication edges. Moreover, an additional `info` argument can be passed to the interface which provides the user with more control over further optimizations related to process topologies. For instance, the `info` argument can be used to designate various optimization criteria for process mapping, as well as to influence the interpretation of edge weights.

In particular, MPI-2.2 defines two distributed graph constructors: `MPI_Dist_graph_create_adjacent()` and `MPI_Dist_graph_create()`. In the former, each process only specifies its own outgoing and incoming neighbors, whereas in the latter each process can specify an arbitrary set of edges that may or may not include its own neighbors. The adjacent specification has the advantage that the neighborhood information is already available locally at each process, whereas in the non-adjacent specification, extraction of neighborhood information might need communication among the processes. A minor disadvantage of the adjacent specification is that each edge is supplied twice; once by each of its endpoints.

Although the graph topology functions can be used to describe any virtual process topology, it is easier and more efficient to use the Cartesian interface to describe certain process topologies. These include n-dimensional grid-based mesh/torus topologies that can be entirely defined by the number of dimensions and the number of processes along each dimension. Accordingly, MPI provides a number of functions that allow for creation and manipulation of Cartesian topologies. In particular, the function `MPI_Cart_create()` creates a new communicator with a Cartesian topology of a desired number of dimensions. The user can also specify whether each dimension is periodic or not, i.e., whether there is a

wraparound link that connects the edge processes to each other. In a Cartesian communicator, each process can be addressed by its coordinates along each dimension of the Cartesian topology. Note that a Cartesian topology can only specify communications among immediate neighbor processes. For instance, it cannot be used to describe communications between diagonal neighbors. Moreover, it does not support weighted edges.

2.2.6 Neighborhood Collective Communications

The topology interface does not add any communication functions to MPI; it mainly helps to describe virtual process topologies. However, the MPI-3.0 standard (released in 2012) introduced *neighborhood collectives* which add communication functions to the process topologies. Similar to the conventional collective communications, neighborhood collectives provide an abstraction for communications among a group pf processes. However, there are fundamental differences between the conventional collective communications and the more-recently introduced neighborhood collectives.

Despite their importance and extensive use in parallel applications, the conventional collectives suffer from certain restrictions. They have inherent scalability limitations due to the global nature of their communications that encompasses all the processes in a given communicator. This concern is especially important at the exascale level where certain collective patterns (e.g., all-to-all) tend to become too costly to be practical. In addition, MPI collectives model only a fixed set of predefined communication patterns such as broadcast, allgather, etc. Any other desired patterns must be manually implemented by the programmer using individual point-to-point communications.

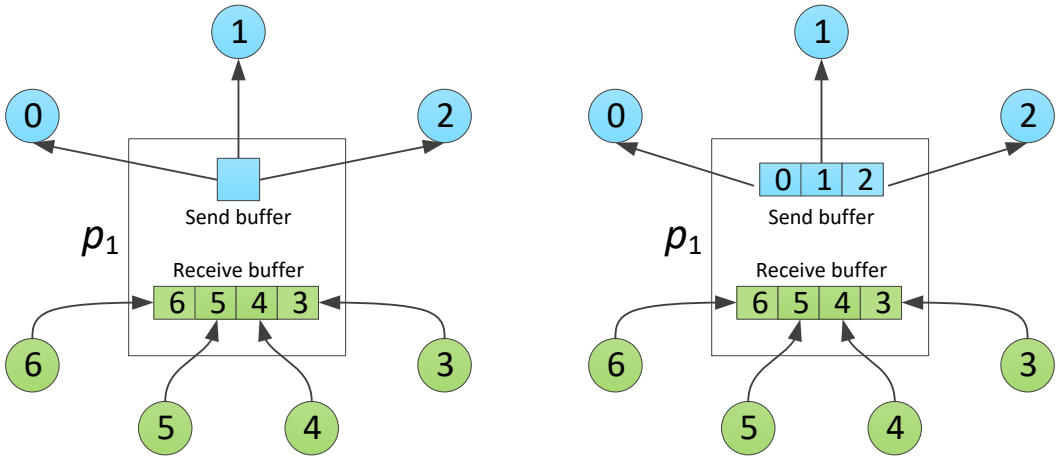
Neighborhood collectives attempt to address these shortcomings. The specific communication pattern of a neighborhood collective is defined by the topology graph of the processes. More specifically, each process will communicate with any other process that is defined as one of its outgoing/incoming neighbors. Thus, unlike conventional collectives, neighborhood collectives allow users to define their own communication patterns through

the process topology interface of MPI. In this sense, neighborhood collectives vastly extend the concept of collective communications in MPI and represent one of the most advanced features of the standard. In addition, neighborhood collectives tend to be more scalable, by restricting communications to a local neighborhood of each process. They also provide support for *sparse* communication patterns [46] found in many applications such as Nek5000 [88], Qbox [39] and octopus [18]. Although such sparse collective communications can be implemented using point-to-point operations, the neighborhood knowledge provided by the virtual topology can be exploited to implement such communication patterns more efficiently. This approach will also lead to performance portability and higher levels of readability and maintainability of the application code.

Currently, the standard defines two main neighborhood collective operations: `MPI_Neighbor_allgather()` and `MPI_Neighbor_alltoall()`. In a neighbor allgather operation, each process sends its data to each of its outgoing neighbors designated by the process topology graph, and receives the data from each of its incoming neighbors. The neighbor alltoall operation has the same communication pattern. The difference is that in neighbor allgather the same message is sent to all outgoing neighbors of a process, whereas in neighbor alltoall, a different message is sent to each outgoing neighbor of a process. Figure 2.6 shows a partial process topology graph and the input/output buffers corresponding to a neighbor allgather and neighbor alltoall operation. Note that the figure only shows the communications from the viewpoint of a single process p_1. Similar communications are conducted at every other process in the topology graph with respect to their own outgoing/incoming neighbors.

2.3 High-Performance Interconnects

InfiniBand (IB) [51] is the most popular switched interconnect in HPC systems. At the time of writing this document it is being used in 37.4% of the top 500 supercomputers in the world [117]. The InfiniBand Architecture (IBA) was first standardized in October 2000 [51] by the



(a) Neighbor Allgather (b) Neighbor Alltoall

Figure 2.6: A sample process topology with the buffer states corresponding to the neighbor allgather and neighbor alltoall operations from the viewpoint of a single process p_1.

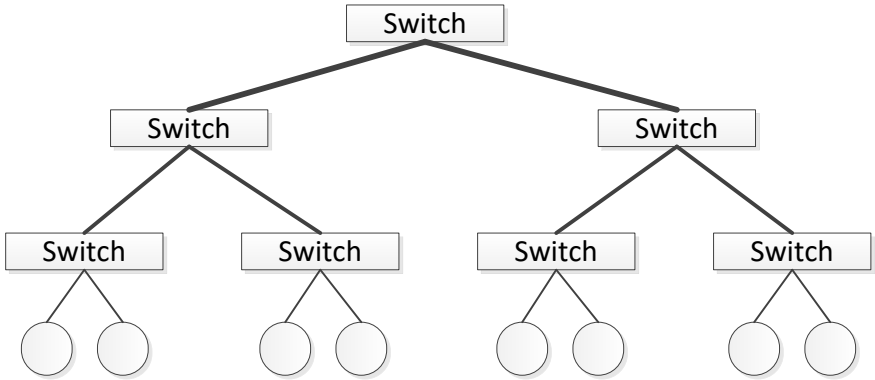
InfiniBand Trade Association (IBTA); a group of 180 or more companies organized in 1999 to develop IBA. The main goal of IBA is to provide an industry-standard high-bandwidth low-latency communication technology that can replace proprietary or low-performance communication architectures in HPC domain. There is a fundamental distinction between InfiniBand and other conventional networks such as Ethernet. In a conventional network, the TCP/IP protocol stack and the network interface card (NIC) are owned by the operating system (OS). In order to communicate, applications request the OS to do the message transfer on their behalf. In contrast, InfiniBand provides a messaging service that can directly be accessed by an application for either storage or inter-process communications [38]. This means that applications no longer rely on OS to provide them with communication services.

An InfiniBand network consists of at least one subnet. A subnet is a group of endnodes interconnected using switches and point-to-point links. IBA links are bidirectional and may be either copper or optical fiber. Within a single subnet, routing is done based on Local Identifiers (LIDs) assigned to endnodes and switches. Channel Adapters (CAs) provide the

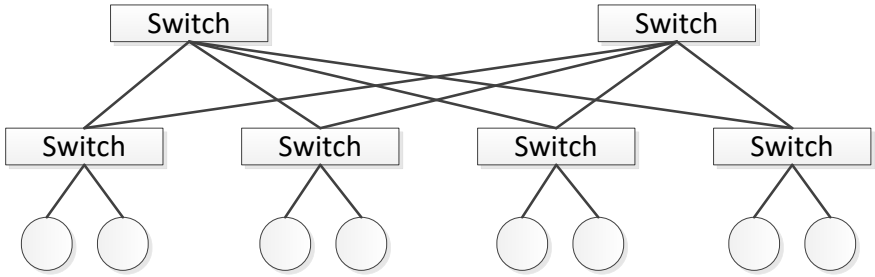
required interface between an endnode and a link. The channel adapter used for a host is called the Host Channel Adapter (HCA), whereas the channel adapter used for a device is referred to as the Target Channel Adapter (TCA). InfiniBand defines two semantics for message transmissions: *channel semantic* (Send/Receive), and *memory semantic* (Remote Direct Memory Access, RDMA). In the channel semantic, the message is sent to the receiver without the sender having explicit access to the buffers at the receiving side. Thus, it is up to the receiver to handle the incoming message properly and deliver it to a corresponding memory buffer. On the contrary, with the RDMA semantic, the sender (receiver) is given explicit access to the address space of the receiver (sender) on a remote node. RDMA is an important and desirable feature in HPC that provides support for communication optimizations at higher levels such as MPI.

With respect to the topology, fat-tree is the most commonly used topology for InfiniBand interconnects. In a fat-tree topology, the nodes are organized into a multi-level tree structure with the compute nodes at the leaves and the switches at the upper levels. The bandwidth of links is increased at each level of the tree as we go up to the root. Fig. 2.7(a) shows a sample three-level fat-tree consisting of 8 compute nodes and 7 switches. However, such single-rooted fat-trees are not practical to build because they require links and switches of different bandwidths [125]. Instead, real installations use multi-rooted fat-trees such as the one shown in Fig. 2.7(b) where the same switch and link type is used at all levels of the tree. Such uniformity of switches and links is an important factor for the total cost of the large-scale systems. The Stampede supercomputer at the Texas Advanced Computing Center (TACC) with 6,400 compute nodes uses a fat-tree topology. It consists of eight 648-port core switches and over 320 36-port endpoint switches.

Another widely used network topology in HPC systems is n-dimensional mesh/torus. In an n-dimensional mesh topology, the nodes are organized into an n-dimensional grid and each node is directly connected to its immediate neighbors along each dimension. A torus topology is similar to mesh except that the edge nodes are connected to each other



(a) Single-rooted—The thicker lines represent the increasing link bandwidth towards the root of the tree.

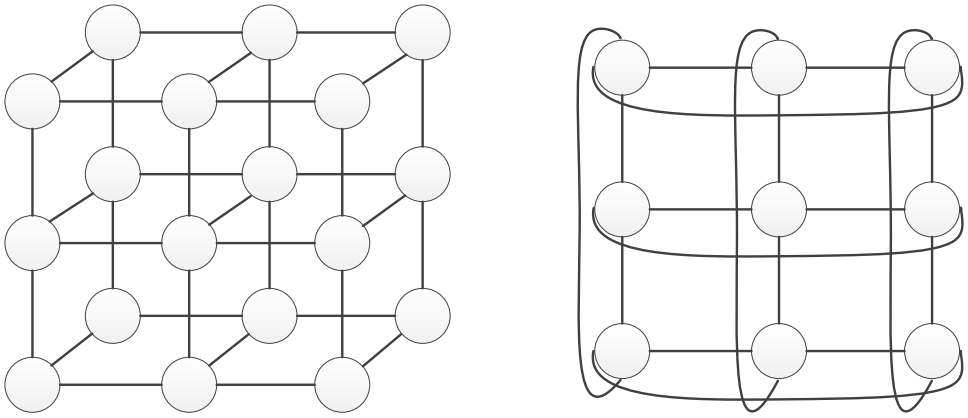


(b) Multi-rooted

Figure 2.7: Single- and multi-rooted fat-tree topologies.

through a wraparound link. 2-dimensional (2D), 3-dimensional (3D), and 5-dimensional (5D) mesh/torus are among the most widely used mesh/torus topologies. Fig. 2.8 shows a sample 3D mesh and 2D torus topology. The IBM Blue Gene series of supercomputers are the most notable systems that use the mesh/torus topologies.

Mesh/torus topologies are low-radix large-diameter networks. In a low-radix network, each node is directly connected to only a few other nodes. In contrast, multi-level direct topologies such as Dragonfly [62] exploit high-radix switches to provide low-diameter networks. High-radix switches provide more ports, allowing to establish more direct connections among the nodes. Multi-level direct topologies consist of a hierarchy of fully connected nodes. At each level, a certain number of nodes are directly connected to each other to form a clique. Each clique is then recursively used as a super node in the next levels. Fig. 2.9



(a) 3D mesh (b) 2D torus

Figure 2.8: n-dimensional mesh and torus topologies.

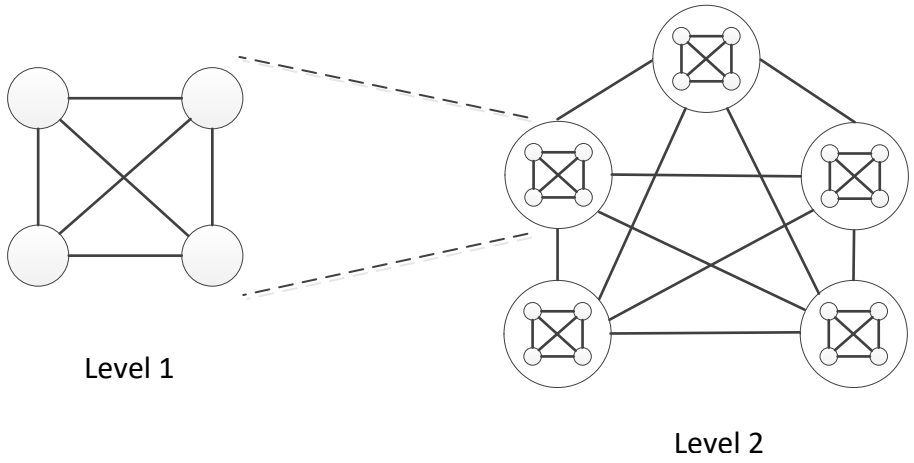


Figure 2.9: A two-level Dragonfly topology with all-to-all connections. The first level groups consist of 4 nodes. Each 4-node group becomes a super node at the second level.

shows a sample two-level Dragonfly topology. Dragonfly is the topology used in the latest series of the Cray supercomputers (XC [26]).

Another common interconnect for HPC is 10G Ethernet, which has a system share of 35.6% among the top 500 supercomputers. However, a major drawback of 10G Ethernet is its relatively higher latency which stems from the overheads associated with the kernel-level TCP protocol stack. In this regard, the Internet Wide Area RDMA Protocol (iWARP

[102]) helps to remove such overheads by offloading the protocol processing to hardware and allowing the application to bypass the OS and communicate with the NIC directly. In fact, iWARP implements some of the features of InfiniBand over Ethernet.

Chapter 3

Topology- and Routing-Aware Process Mapping

In this chapter, we propose a Parallel Topology- and Routing-Aware Mapping (PTRAM) approach to improve the communication performance by decreasing congestion. Specifically, we propose a topology- and routing-aware process-to-node mapping heuristic as well as a refinement algorithm. The proposed algorithms are parallel in nature, and take into account the routing information so as to derive a better evaluation of congestion across the network links. For InfiniBand clusters in particular, we take into account the static assignment of routes across the links. Previous studies [44] show that most communication patterns cannot achieve full bisection bandwidth in practical fat-tree InfiniBand networks. The problem is not the number of available physical links; it is the static routing scheme which might oversubscribe some links while leaving others idle. To the best of our knowledge, this is the first work that takes into account the static routing scheme of InfiniBand to derive a better process mapping.

In particular, we propose a greedy mapping heuristic accompanied by a mapping refinement algorithm [82]. The greedy mapping heuristic attempts to minimize a *hybrid metric* that is used to evaluate candidate mappings from multiple aspects. The refinement algorithm on the other hand attempts to explicitly reduce maximum congestion by refining the mapping output from the greedy heuristic. We take advantage of parallelism in the design and implementation of our proposed algorithms. We believe parallelism is the key for having

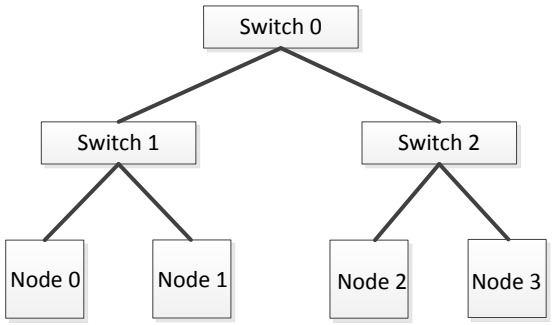
topology-aware mapping approaches with better scalability and lower order of complexity. To the best of our knowledge, this is the first attempt in proposing parallel algorithms for topology-aware process mapping.

3.1 Motivational Results

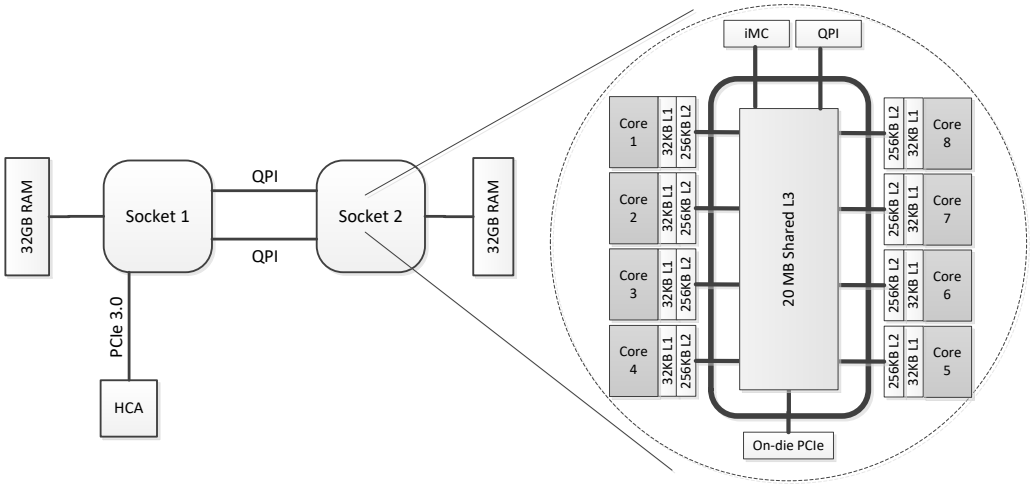
We conduct an experiment to shed more light on the heterogeneity of communications performance within a parallel computing system. In our experiment, we measure the latency and bandwidth of a single-pair communication at the intra- and inter-node levels of a small InfiniBand cluster. To this end, we use an MPI microbenchmark with two processes. The first process (source) iteratively calls the MPI nonblocking send function 1,000 times to send messages to the second (destination) process. The second process issues 1,000 nonblocking receive operations and waits to receive them all. After that, the second process sends an acknowledgment to the first process to signal the completion of all message receptions. The first process measures the total time and we report the corresponding averages for latency and bandwidth. For the congestion experiments, we use two other processes residing on two other nodes that keep communicating large messages with each other.

Fig. 3.1(a) and 3.1(b) respectively show the system topology at the network and node levels. The system consists of four nodes connected to each other by three switches with a tree topology. The network is QDR InfiniBand with 40 Gbps bandwidth per link and each node uses a Mellanox ConnectX-2 HCA. In addition, each node has two 8-core Intel Xeon Sandy Bridge sockets operating at 2.0 GHz with a total of 64 GB memory. The two sockets are connected to each other by two Intel QPI links. Each socket forms a NUMA node with 32 GB of local memory and 20 MB of L3 cache.

Fig. 3.2 to 3.4 show the communication latency and bandwidth results. According to Fig. 3.2, the intra-socket communications are shown to benefit from a lower latency compared to the inter-socket communications across all message sizes. For small messages (Fig.



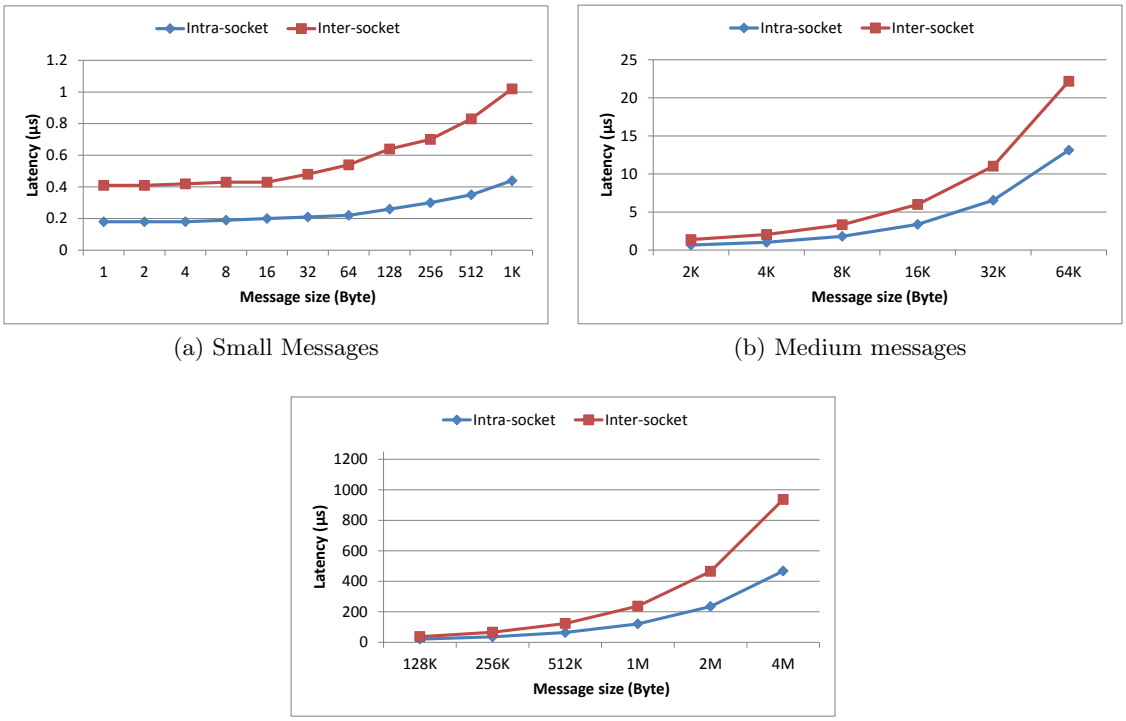
(a) Inter-node topology



(b) Intra-node topology

Figure 3.1: System topology at the inter- and intra-node levels.

3.2(a)), this comes from the fact that the cores within a single socket have a lower distance from each other (L3 shared cache) compared to those belonging to different sockets. However, the lower latency of the intra-socket communications for medium and large messages (Fig. 3.2(b) and 3.2(c)) is due to the higher bandwidth of the intra-socket channels compared to the QPI links that connect the two sockets. This is also verified by Fig. 3.4(a) where the intra-socket channel saturates at a higher bandwidth than the inter-socket one.



(a) Small Messages

(b) Medium messages

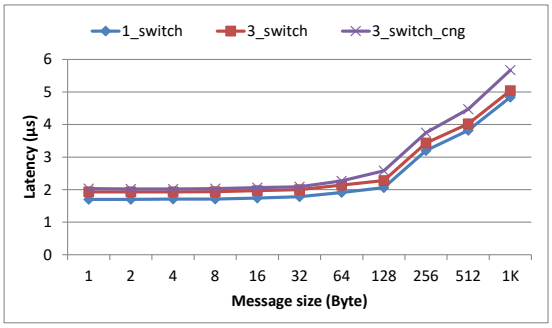
(c) Large messages

Figure 3.2: Intra-node single-pair communication latencies for various message sizes.

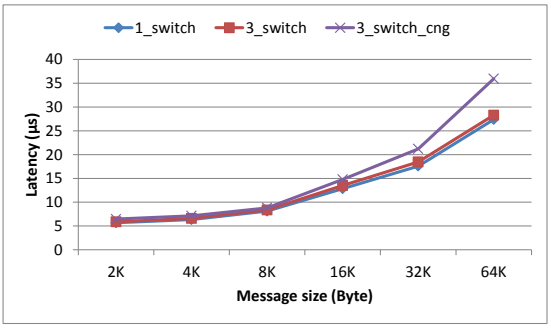
For the inter-node level, we measure the communications performance for the following three cases:

1. communication across a single switch ("1-switch"),
2. communication across 3 switches ("3-switch"),
3. communication across 3 switches in presence of congestion at the root of the network topology tree ("3-switch cng").

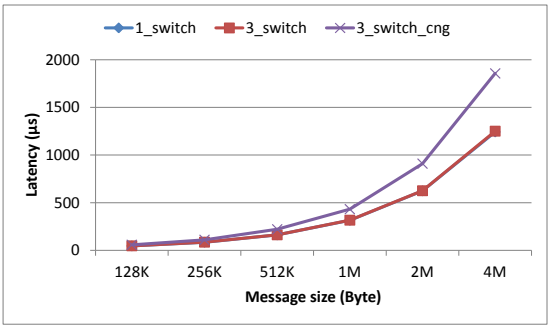
According to Fig. 3.3(a), we do not observe much of a difference in latency for small messages. The reason is that small messages are more sensitive to start-up latency and network distance rather than the channel bandwidth. In addition, the distance variation



(a) Small Messages

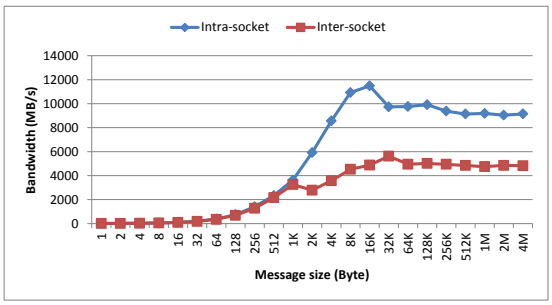


(b) Medium messages

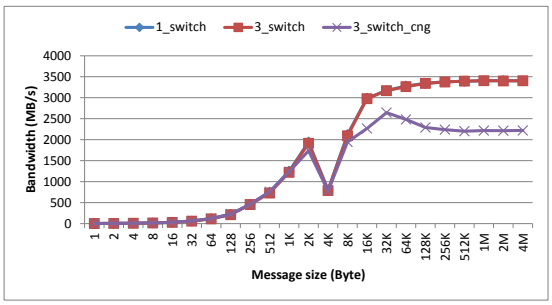


(c) Large messages

Figure 3.3: Inter-node single-pair communication latencies for various message sizes.



(a) Unidirectional intra-node bandwidth



(b) Unidirectional inter-node bandwidth

Figure 3.4: Communication bandwidth variations at the intra- and inter-node levels

in the experiments are not large enough to considerably affect the performance for small messages. Nevertheless, as shown in Fig. 3.3(a), latency is slightly lower for 1-switch communications. The difference will be more visible in a system with a larger network diameter. For medium and large messages, 1-switch and 3-switch communications achieve the same performance, whereas the 3-switch communications with congestion fall behind them. This is because medium and large messages are mainly affected by the bandwidth of the communication channel, and as the bandwidth is the same for both 1-switch and 3-switch cases, we do not see any difference in their corresponding performance. However, the presence of congestion reduces the achievable bandwidth for communication, leading to an increase in the latency of the 3-switch communication with congestion. This conforms to the results shown in Fig. 3.4(b), where we see the exact same bandwidth for communication across one and three switches without congestion, and a lower bandwidth for communication in the presence of congestion. Finally, by comparing the results in Fig. 3.2 and 3.3, we see that intra-node communications significantly outperform the inter-node ones.

3.2 Topology-Aware Mapping Overview

Topology-aware mapping attempts to optimize communications by exploiting the topology information. Topology information includes the application process topology, which gives the communication pattern of the processes running the application, as well as the physical topology of the target machine. There are two reasons for why a nontrivial mapping can improve the communication performance of an application based on the topology information. First, we see a certain communication pattern among the processes of a parallel application. Some pairs of processes communicate more often, some pairs send and receive higher volumes of data, and some pairs might not even communicate at all. The communication pattern is relatively sparse in many HPC applications. Second, as shown in Section 3.1, the latency and bandwidth of the communication channels are not the same at different layers

of the system hierarchy. Communication among particular cores can be significantly faster than others. As a result, an intelligent mapping of processes can help to better match the application communication pattern to the underlying communication channels.

In general, topology-aware mapping strategies can be broken down into the following three phases:

1. extracting the communication pattern of the application (also known as the process/virtual topology),
2. extracting the hardware topology information (physical topology),
3. mapping the process topology onto the physical topology by a mapping algorithm.

Naturally, both process and physical topologies are modeled by graphs, making topology-aware mapping an instance of the graph embedding problem [105] which is known to be NP-hard. Therefore, heuristics are used to find suboptimal solutions with respect to some *metric* that is used to assess the quality of a particular mapping.

3.2.1 Extracting the Application Communication Pattern

The application communication pattern mainly represents the volume and/or number of messages transfered among the processes. In particular, we use the total volume of messages transferred between each pair of processes to model the communication pattern in terms of a directed weighted graph. To this end, the application needs to be profiled first, for which we have developed a profiler by instrumenting the MPI library. The profiler will capture the volume and destination of all messages sent by each process. At each process, this information is saved into a vector with one element per destination process. Each element will represent the total volume of messages sent to a specific destination process. At the end, the vectors from all processes are gathered in one place to build a communication pattern matrix which is saved into a file for future use. We apply the instrumentation at

the device layer of the MPI library so as to capture the pattern for both the point-to-point and collective communications of an application.

It should be noted that the profiling is done only once for each application in an initial profiling stage. Accordingly, It is assumed that the application communication pattern remains (mostly) unchanged from one run to another. Such an assumption is valid for many HPC applications and has widely been used in topology-aware mapping research.

3.2.2 Extracting the Hardware Physical Topology

Physical topology consists of two main parts: a) inter-node network topology, and b) intra-node core topology. The physical topology information needs to be collected *once* for each system. The network topology can usually be obtained by exploiting the topology-related APIs provided by the software stack of the corresponding interconnect. For InfiniBand, the OpenFabrics Enterprise Distribution (OFED) [91] protocol stack provides some tools and low-level APIs to query the topology information. Some researchers have also designed higher-level tools on top of these APIs to make network topology information more accessible within a cluster [21, 111]. For our design, we use a modified version of the `ibtracert` tool [91] to extract the route between each pair of nodes in terms of a sequence of switches and ports. In addition, we have developed another tool based on the `ibnetdisc` [91] library to extract all the network links. Each link is represented by the Local Identifier (LID) and port number of its corresponding end points.

The intra-node core topology represents the specific NUMA node organization of cores within each node. In this regard, the hwloc library [16] provides a portable way for acquiring various information about the processors within a node including the number of sockets, cache hierarchies, memory distribution across NUMA nodes, etc. It can also provide a multi-level tree representation of the processing elements (cores/threads) based on such information. It is worth noting that we do not use the intra-node topology information in this chapter. However, we use it for our topology-aware collective design in Chapter 4.

3.2.3 Mapping Metrics

Hop-bytes and dilation

Hop-bytes is the metric based on which the majority of mapping heuristics have been designed [15, 80, 104, 101, 81, 56]. For each individual message, hop-bytes represents the product of the message size and the number of hops traversed from source to destination. The summation of such products over all communicated messages represents the hop-bytes value of a given mapping. Accordingly, mapping algorithms attempt to assign the processes to the processing elements so as to minimize the resulting hop-bytes value. Eliminating message sizes from hop-bytes will result in *dilation*; another mapping metric that represents the total number of hops traversed by the messages within the system. Thus, hop-bytes can be considered as a weighted dilation, with the weights being the volume of messages.

Compared to dilation, hop-bytes has the advantage of taking into account the size of messages when measuring hop counts. It is worth noting that with hop-bytes as the metric, topology-aware mapping becomes a Quadratic Assignment Problem (QAP) [71] which is known to be one of the hardest combinatorial optimization problems. In a QAP, there are n facilities and n locations. The facilities have a weighted flow graph, whereas the locations have a weighted distance graph. The goal is to map the facilities to the locations so that the sum of the location distances multiplied by the flow weight of their corresponding facilities is minimized.

Shortcomings of the hop-based mapping approaches

Most of the existing mapping heuristics have mainly been designed based on the distance among the communicating peers [15, 80, 104, 101, 81, 56]. In other words, they attempt to map the processes so as to decrease the number of hops traversed by messages. Decreasing hop-count will potentially improve communication performance by a) avoiding additional switch latencies, and b) reducing contention among messages by decreasing the amount of

link-sharing among them.

However, distance-based mapping techniques have two main shortcomings. First, distance-based mapping algorithms (based on dilation or hop-bytes) cannot capture the congestion imposed on individual links, whereas congestion is known to be a major communication bottleneck. In fact, a recent study [54] shows that the traditional metrics used for topology-aware mapping do not correlate well enough with the application run-time performance. Thus, even if the mapping heuristics built on top of such metrics can successfully optimize the metrics, the resulting mapping can still be far from optimal with respect to the actual application execution time.

Second, a pure distance-based approach will lose its effectiveness if the hop-count variations of the target topology is low. For instance, unlike mesh/torus topologies which impose a large number of hops among certain nodes, fat-trees tend to have a lower network diameter. In fact, the trend in HPC interconnects is moving toward the design and use of high-radix low-diameter topologies. For instance, Dragonfly [62] is the topology of choice for the Aries [3] interconnect used in the latest series of Cray supercomputers (XC [26]). The maximum hop-count in these systems is 4 with non-minimal routing (2 with minimal routing).

Congestion

Maximum congestion is another metric used in topology-aware process mapping. In this context, congestion represents the cumulative traffic load that will pass through the network links during the execution of an application. Therefore, it provides a static measure of congestion across the links. More specifically, the congestion value of each individual link is defined as the total volume of messages passing through that link, divided by the capacity of that link. Formally speaking, let $\tau : P \rightarrow N$ represent a mapping from the set of processes P to the set of nodes N. Moreover, let L denote the set of all links in the target system. For every pair of processes $(p, q) \in P$, we use $L(p, q, \tau) \subseteq L$ to denote the set of links used in the

path from $\tau(p)$ to $\tau(q)$. Moreover, we denote by $SD(l,\tau)$ the set of all source-destination processes $(p,q) \in G_M$ for which the route from $\tau(p)$ to $\tau(q)$ involves link l. Note that $(p,q) \in SD(l,\tau)$ if and only if $l \in L(p,q,\tau)$. Now, the congestion value of a given link l is given by

$$\text{Congestion}(l) = \frac{\sum_{(p,q)\in SD(l,\tau)} w(p,q)}{c(l)}, \tag{3.1}$$

where $w(p,q)$ denotes the weight of communication between p and q, and $c(l)$ represents the capacity of link l. The maximum congestion metric of a mapping τ is then defined as:

$$\text{Congestion}(\tau) = \max_{l\in L} \text{Congestion}(l). \tag{3.2}$$

3.2.4 Process Mapping and Rank Reordering in MPI

In general, topology-aware mapping can be considered in the context of any parallel programming paradigm. For MPI, the standard defines various topology-related APIs that can be used to facilitate topology-related optimizations in MPI (see Section 2.2.5). In particular, MPI topology functions can be used to establish a desired mapping through the rank reordering mechanism which allows us to change the default ranks that are assigned to each MPI process. In this regard, the processes are initially mapped onto the set of allocated nodes/cores by the process manager with a default rank assigned to each of them. After that, we change (reorder) the process ranks so as to implicitly modify the mapping with respect to our desired topology-aware outcome. MPI rank reordering can only change the mapping within an initial allocation of nodes/cores. However, this is not a restriction as, in real systems, node allocation is actually enforced by a system-wide resource manager. But, it is technically possible to also enforce the topology-aware mapping results at the resource manager layer.

3.3 PTRAM: Parallel Topology- and Routing-Aware Mapping

In this section, we explain our proposed heuristic and refinement algorithms for topology-aware assignment of processes to processing elements. We distinguish our design from the existing mapping algorithms with four main aspects.

First, we use a *hybrid metric* in our heuristic which is more capable of distinguishing among various candidate mappings in terms of their quality. Individual metrics such as hop-bytes or maximum congestion could easily fail to differentiate two given mappings that actually have different qualities. A previous study [54] on the performance of parallel applications shows that hybrid metrics have a better correlation with application execution time. However, hybrid metrics have not been used in any mapping algorithms.

Second, we take into account the underlying routing algorithm of the target system and the actual distribution of routes across the network links. This makes our mapping algorithms routing-aware, allowing them to capture the impacts of the underlying routing on the mapping. Exploiting routing information also enables us to have a more realistic measure of our mapping metric by keeping track of the load imposed over individual links.

Third, in most of the existing mapping heuristics, the quality metric is used for assessment purposes only. In other words, the metric is not directly optimized within the corresponding heuristic, and is rather used to measure the quality once the mapping is figured out. In contrast, we attempt to optimize the metric directly at each step of our proposed algorithms. In other words, our proposed heuristic uses the actual values of the metric to decide where each given process shall be mapped.

Fourth, we exploit the parallelism capabilities of the target system in our algorithm designs. To the best of our knowledge, this is the first attempt in proposing parallel algorithms for topology-aware mapping. We believe parallelism is the key for having truly scalable mapping algorithms for current and future HPC systems.

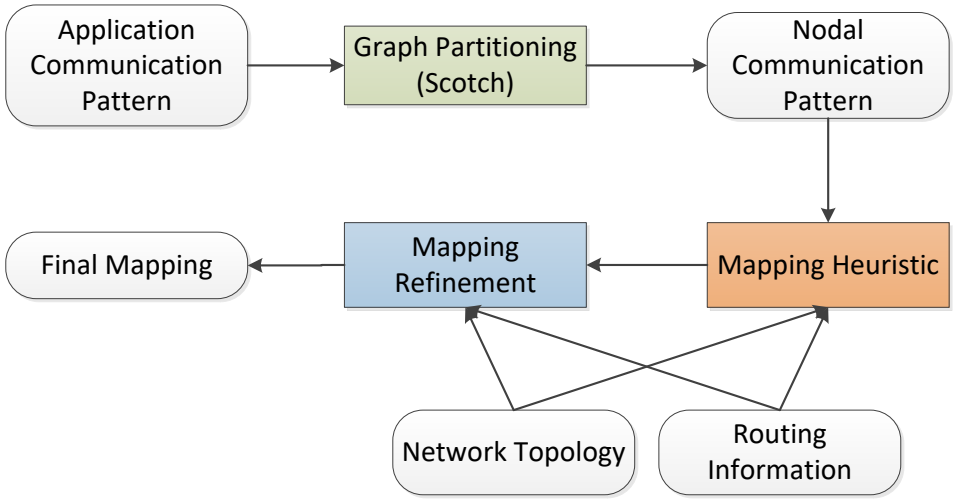


Figure 3.5: Mapping framework. High-level abstraction of various steps and components.

Mapping framework

Fig. 3.5 shows a high-level abstraction of the framework that we use to perform the mapping. It should be noted that a similar framework has been used in other topology-aware mapping studies as well [45], and we are not claiming it as our contribution. We use this general framework as the base for our mapping approach. Our contribution is the particular greedy heuristic and refinement algorithm that we use within this general framework, as well as the addition of routing information and network links into it.

There exist three main steps in the framework shown in Fig. 3.5: 1) an initial partitioning of the application communication pattern, 2) an intermediate mapping of processes using a parallel mapping heuristic, and 3) a final mapping refinement. In the following, we explain the details of each step.

3.3.1 Initial Graph Partitioning

The mapping procedure starts with converting the application communication pattern to a nodal communication pattern. Using a graph partitioning algorithm, we first partition

the process topology graph into a number of partitions equal to the number of compute nodes in the target system. Each node in the resulting graph will encapsulate a number of processes equal to the number of cores within each compute node. Thus, the resulting graph will be representative of the communication pattern among a set of *super processes*, each of which should be mapped onto a single compute node.

The partitioning can be done by any graph partitioning algorithm. There exist a number of well known libraries such as Scotch [95] and METIS [60] for graph partitioning. In this work, we will use the Scotch library to perform the initial partitioning of the application communication pattern graph. It is worth noting that hereinafter, we use the term process (or process topology graph) to refer to a super process in the nodal communication pattern graph.

We use the initial partitioning stage to address multicore nodes within the system as we are only concerned about the mapping at the network (inter-node) layer due to its larger scale. Initial partitioning will potentially put heavily communicating processes in one partition. By collectively mapping all the processes in each partition onto the same node, we will take advantage of the shared-memory channels for communications among such processes. Within each node, the individual processes of a partition are mapped to the cores in order of their corresponding MPI rank. However, our proposed algorithms can also be used across individual cores without any initial partitioning of the process topology graph.

3.3.2 Mapping Heuristic

The parallel mapping heuristic lies at the heart of our mapping procedure. Algorithm 3.1 shows the steps involved in our proposed heuristic. The heuristic takes the nodal communication pattern matrix, the routing information, and the network topology as input, and provides a mapping from the set of processes to the set of nodes as the output.

Algorithm 3.1 is parallel in nature and is run by one *leader* process on each node. Let

Algorithm 3.1: PTRAM: Parallel Topology- and routing-aware mapping: the core heuristic

Input : Set of all processes P, set of all nodes N, nodal communication pattern matrix C, routing information, network topology

Output: The mapping $\tau : P \rightarrow N$

1 $P_M \leftarrow \emptyset$; `// The set of mapped processes`

2 **while** $P_M^c \neq \emptyset$ **do**

`// There are more processes to map`

3 $\alpha = \frac{1}{|P_M|+1}$;

4 **for** $q \in P_M^c$ **do**

5 $\delta = \sum_{r \in P_M} (C_{qr} + C_{rq}) + \alpha \sum_{s \in P_M^c} (C_{qs} + C_{sq})$;

6 **end**

7 $p_{next} \leftarrow q_{max}$; `// Choose the process with the maximum value of` δ `to map next`

8 Temporarily assign p onto self node;

9 Calculate link congestions using routing information;

10 Find the hybrid metric value accordingly;

11 Gather the metric value from all other node leaders;

12 $n =$ node with the lowest value of metric;

13 $\tau(p) = n$; `// Map` p `onto` n

14 Update link congestions accordingly;

15 $P_M \leftarrow p$; `// Add` p `to the set of mapped processes`

16 **end**

P_M and P_M^c denote the set of mapped and unmapped processes, respectively. The main loop in line 2 runs until all processes are mapped onto a target node. Within each iteration, first a new process p_{next} is chosen as the next process to map. A smart choice of such a process is very important in terms of the quality of the resulting mapping, especially as the heuristic is greedy in nature and does not involve any backtracking mechanism; when a process is mapped onto a node at a given iteration of the heuristic, its mapping will remain fixed in the following iterations of the heuristic. Accordingly, we choose p_{next} with respect to a metric denoted by δ in line 5 of Alg. 3.1. For each process such as q, the first summation in line 5 gives the total communication volume of q with all its already-mapped neighbors. The second summation represents the total communication volume with the

unmapped neighbors. The process with the maximum value of δ is chosen as the next process for mapping at each iteration.

The parameter $0 < \alpha \leq 1$ is used to assign a relative (lower) weight to communications with unmapped neighbors in comparison to communications with mapped neighbors. The value of α is updated at every iteration (line 3) with respect to the number of mapped processes. Initially, α starts from 1, and decreases at each iteration as more processes are mapped. We use such an α value because the amount of communication with the set of unmapped neighbors is relatively more important at the initial steps of the algorithm as most of the neighbors of a given process have not yet been mapped. However, as more processes are mapped, we want to give a relatively higher weight to the communications with the set of mapped neighbors when the next process is chosen for mapping.

The next major step in Algorithm 3.1 is to find a target node for p_{next}. Target nodes are chosen based on the value of a *hybrid metric* which we will discuss in detail in Section 3.3.3. At each iteration, we seek to map p_{next} onto the node that will result in the lowest value of the hybrid metric. Thus, we explicitly measure the value of the metric at each iteration of the algorithm, and choose the target node accordingly. This is where we take advantage of parallelism in our heuristic. The leader process on each node[1] n is responsible for calculating the metric value resulting from mapping p_{next} onto n. This is done in lines 8 to 10 of Alg. 3.1. Next, all the leader processes communicate with each other so as to gather the metric values corresponding to all nodes. Specifically, we use `MPI_Allgather` to accomplish this step. Having gathered the metric values from all nodes, the target node for p_{next} is set to be the node with the lowest value of the hybrid metric. In case of having multiple such nodes, we simply choose the one with the lowest leader rank. Finally, before going to the next iteration, each leader process updates its own local copy of the link congestions with respect to the newly mapped process.

Choosing the target node as explained above has two advantages. First, it allows us to

[1]In fact, this is performed by non-occupied nodes only.

explicitly evaluate each non-occupied node before choosing a target node to map the next process. Second, by exploiting parallelism, we will pay a cost equal to evaluation of a single node at each iteration. In fact, each node is made responsible for evaluating itself with respect to the hybrid metric. This way, we increase the quality of the target node selection, and at the same time keep the corresponding costs bounded. More importantly, such a parallel approach will improve the scalability of our heuristic. When the system scales and the number of nodes increase, the search space for finding appropriate nodes is increased as well. This makes the whole mapping problem more challenging for any given mapping algorithm. However, using a leader process on each node allows our heuristic to scale its searching capability in accordance to increase in the system size. We believe that exploiting parallelism is the key to achieve better scalability in topology-aware mapping. Thus, we can exploit the parallelism capabilities of the target system itself to solve the mapping problem. Of course such an approach will require computation time on the target system, but we think the potential benefits will justify it. Moreover, in some scenarios (especially batch systems), the application is run only across a particular subset of all nodes within the system that are dynamically allocated by an underlying resource manager. The topology of such allocations are not known a priori. Therefore, in such cases, it is in fact necessary to solve the mapping problem on the target system itself.

Complexity The main loop of Alg. 3.1 performs one iteration per (super) process until all of them are mapped. With n denoting the number of nodes (super processes), we will have n iterations. In each iteration, finding the value of δ takes $\mathcal{O}(nd)$, where d denotes the highest number of neighbors per process (degree of the the communication pattern graph). Finding q_{max} can be done in $\mathcal{O}(n)$. Calculating link congestions takes $\mathcal{O}(dl)$, where l denotes the number of links that connect two nodes. For fat-tees, $l \propto h$, with h denoting the height of the tree. The next major step is gathering metric value which takes $\mathcal{O}(n)$ (using a ring allgather). Finding the node with lowest value of the metric is done in $\mathcal{O}(n)$.

Finally, updating the link congestions will be again of order $\mathcal{O}(dl)$. Thus, the complexity of Alg. 3.1 can be given by $\mathcal{O}(n(nd + n + dl + n + n + dl)) = \mathcal{O}(n^2d + ndl)$.

3.3.3 Hybrid Metric

Rather than using only hop-bytes or maximum congestion, we take advantage of a hybrid metric in our mapping heuristic. The metric is a combination of four individual metrics, each of which evaluate a given mapping from a particular aspect. Specifically, the hybrid metric consists of the following four individual metrics:

1. hop-bytes (HB)
2. maximum congestion (MC)
3. non-zero congestion average (NZCA)
4. non-zero congestion variance (NZCV)

In our design, we consider a linear combination of these metrics to build up our hybrid metric as

$$\text{Hybrid Metric} = HB + MC + NZCA + NZCV \tag{3.3}$$

The hop-bytes and maximum congestion components of our hybrid metric are the same as those defined in Section 3.2.3 and Section 3.2.3. We define *non-zero congestion average* as the average of the traffic that passes through each non-idle link in the network. More specifically, non-zero congestion average for a given mapping is equal to the total sum of the traffic across all links divided by the number of links that are actually utilized by the mapping. In other words, it is the average of traffic among the links with a non-zero volume of traffic passing through them. Similarly, *non-zero congestion variance* is defined as the variance of the traffic across the links with a non-zero load passing through them.

We use a hybrid metric because hop-bytes or maximum congestion alone is limited in terms of distinguishing among multiple candidate mappings. Hop-bytes will always give

a lower priority to a mapping that causes traffic to go across multiple links. There are two main shortcomings with such a behavior. First, it does not take into account the fact that using more links (i.e., more hops) can potentially result in better link utilization and traffic distribution. Better traffic distribution will in turn help to have a lower maximum congestion on the links. Second, hop-bytes alone cannot distinguish between two mappings that have the same hop-bytes value and yet are different in terms of congestion (see Section 3.3.4 for an example of such a case).

On the other hand, mapping processes solely based on the resulting maximum congestion at each iteration of the mapping algorithm will have its own shortcomings. The main issue is that several mappings can all result in the same maximum congestion at a given step of the mapping algorithm, while they provide different potentials for mapping the rest of the processes. Such cases can specially happen when the mapping of a particular process on different nodes does not affect the load on the link with the maximum congestion, but results in different traffic load on other links (or different hop-bytes values). Deciding solely based on the maximum congestion does not allow us to distinguish among such mappings. Even with the same maximum congestion, we would still like to choose the mapping that results in a lower traffic load on the set of all links. This will provide higher chances to decrease maximum congestion in future steps of the mapping algorithm.

Accordingly, we use non-zero congestion average and variance as a measure for the *balancedness* of traffic across the links for a given mapping. There is a reason why we do not use an ordinary average across all links in the network. An ordinary average will be equal to the total sum of the traffic divided by the total number of links in the target system. As the total number of links in the system is a fixed number, an ordinary average would be only reflective of the sum of the network traffic. Such a measure cannot tell us about the link utilization and traffic distribution characteristics of a given mapping. Non-zero congestion average on the other hand provides an implicit measure of both the traffic sum, and the degree of balancedness of the total traffic across the links. Similarly, non-zero congestion

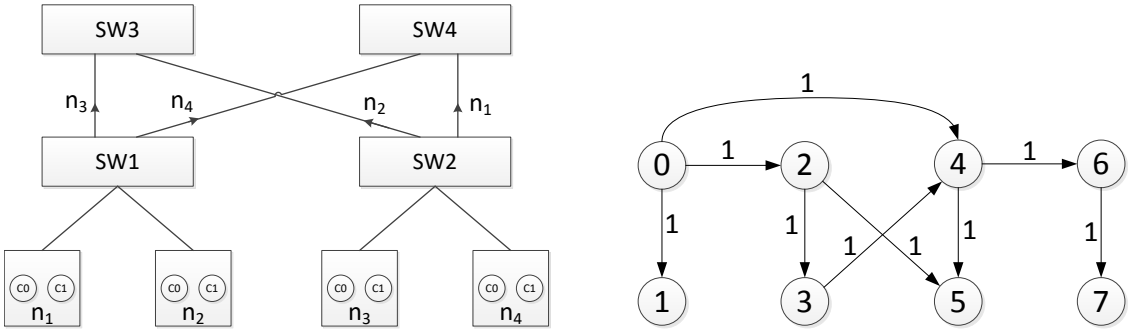
variance would provide us with additional information regarding the balancedness of traffic.

It is worth noting that one can use other combinations of the above metrics to build the hybrid metric. The linear combination used in our design is actually the simplest form in which all the individual metrics have the same weight. Determining the optimal combination of metrics is an interesting topic for research. For instance, a better approach would be to assign a different weight to each individual metric with respect to the corresponding impact of that metric on performance.

3.3.4 Impact of Routing Awareness

In this section, we try to shed more light on the potential impacts of routing awareness on process mapping. In a parallel system, the congestion imposed on the set of network links is a function of both the topology and the underlying routing algorithm. This is particularly true for InfiniBand where the underlying static routing can highly affect congestion [44]. Even if the interconnect provides full bisection bandwidth, a routing-ignorant process mapping could result in unnecessary congestion over the links. The issue becomes more significant in practical installations of InfiniBand systems that do not provide full bisection bandwidth. Therefore, in our mapping heuristic, we take into account the routing information when we want to calculate congestion on each network link. This way, we will have a more realistic congestion model throughout all steps of our mapping algorithm. In the following, we use a simple example to clarify how routing awareness can make a difference in process mapping.

Consider a target system with the topology shown in Fig. 3.6(a). The system consists of 4 nodes n_1, n_2, n_3, n_4, each having 2 cores. We would like to map 8 processes with the topology shown in Fig. 3.6(b) onto this system. The edge weights in Fig. 3.6(b) represent the communication volumes. For the sake of simplicity, we consider equal weights for all edges. The underlying routing of the system has been partially shown by the labels (n_1 to n_4) assigned to the four top links in Fig. 3.6(a). We only represent the routes in one direction (i.e., bottom-up) as it suffices for the purpose of this example. The link labels



(a) Sample target system (b) Sample communication pattern graph

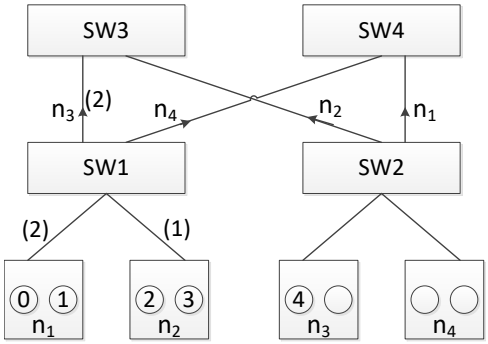
Figure 3.6: A sample target system topology and the communication pattern graph of the processes.

represent the destination nodes associated to each link.

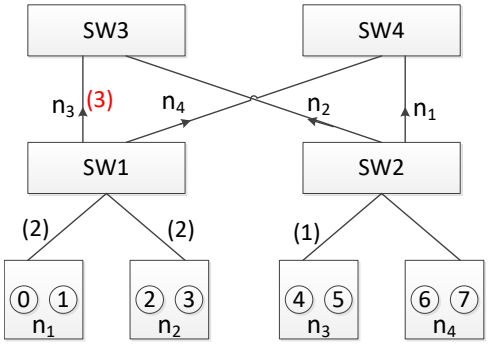
We start the mapping by assigning rank 0 to one of the cores (could be any of the cores). Having mapped four other processes, the mapping will be as shown in Fig. 3.7(a). The link labels in this figure represent the corresponding congestion values. For the sake of brevity, we represent the congestion values of all links in one direction only (i.e., bottom-up direction in the tree). Up to this point, routing awareness does not play any role and hence, the result will be the same for a routing-ignorant approach as well. However, when it comes to the next process (i.e., rank 5), we can see the impact of routing awareness.

Rank 5 can be assigned to a core on either n_3 or n_4. Fig. 3.7(b) and 3.7(c) show the final mapping and the corresponding link congestions with respect to each of such assignments. It can be seen that mapping rank 5 to a core on n_4 (Fig. 3.7(c)) will lead to a lower maximum congestion across the links. By exploiting the routing information, a routing-aware approach will choose the mapping shown in Fig. 3.7(c), whereas a routing-ignorant approach cannot distinguish between the two mappings. More specifically, a routing-ignorant approach cannot distinguish between n_3 and n_4 while mapping rank 5.

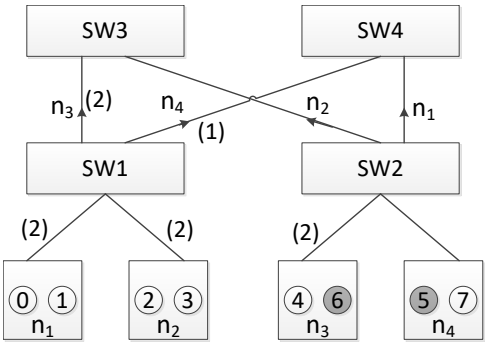
The above scenario is an example for how process mapping can potentially benefit from routing awareness. This is just a simple example on a very small system. As the system



(a) The mapping layout for the first five processes.



(b) The mapping outcome from a routing-ignorant approach. Maximum congestion is equal to 3.



(c) The mapping outcome from a routing-aware approach. Maximum congestion is equal to 2.

Figure 3.7: Different mappings of processes leading to different values of maximum congestion across the links.

scales and the number of nodes and switches increase, the topology becomes more complex, and the impacts of routing awareness on congestion will increase too. It is worth noting that we feed the routing information to Alg. 3.1 in terms of a file. The routing file consists of N^2 lines where N denotes the number of end nodes in the system. Each line corresponds to the route that connects a pair of nodes. Initially, each leader process loads the routing and links information into its memory. Because each leader process is only responsible for evaluating its own node n, it only needs to load a portion of the routing file; that is, the route from n to every other node, and the route from every other node to n. The advantage is a significant reduction in memory requirement from $\mathcal{O}(N^2)$ to $\mathcal{O}(N)$.

3.3.5 Refinement Algorithm

A shortcoming of the heuristic explained in Section 3.3.2 is its lack of a backtracking mechanism. The heuristic is greedy in nature, and thus, optimizing the hybrid metric at each stage does not necessarily guarantee optimality at the end. Therefore, we might still be able to tune the mapping further in a final refinement step. Accordingly, we propose a refinement algorithm as outlined by Algorithm 3.2. At the refinement stage, our main goal is to specifically decrease the maximum congestion across the links by swapping particular processes.

Algorithm 3.2: PTRAM: The refinement algorithm

Input : Set of all nodes N, communication pattern matrix C, routing information, network topology
Output: The refined mapping $\tau : P \rightarrow N$

```
1  n = self node;
2  p = process assigned to n;
3  maxCong = findMaxCong() ; // Find the current value of maximum
     congestion
4  l = findMaxLink() ; // Find link with maximum congestion
5  for q where C[p][q] ≠ 0 and l ∈ L(p, q, τ) do
6      M = list of k closest nodes to n;
7      for m ∈ M do
8          r = process mapped to m;
9          Temporarily swap p with r;
10         Update link congestions accordingly;
11         newMaxCong = findMaxCong() ; // Find the new value of maximum
             congestion
12         if newMaxCong < maxCong then
13             maxCong = newMaxCong;
14             Save p and r;
15         end
16     end
17 end
18 Gather maxCong and the associated swappings from all leaders;
19 Find the lowest value of maxCong;
20 Enforce the corresponding swapping;
21 Update link congestions accordingly;
```

Alg. 3.2 is run in parallel by one leader process on each node. At the beginning of the algorithm, all the leader processes have a copy of the mapping output from Alg. 3.1 in the form of an array. For a given node such as n, let p denote the process assigned to n based on the mapping array. In Alg. 3.2, the leader process on node n checks to see whether swapping p with other processes can decrease maximum congestion. To this end, we first find the link l with the maximum congestion (line 4), and see whether p communicates with any other process (q) across a route that involves l (line 5). If so, we check whether swapping p with any of the processes on a node close to n will result in a lower maximum congestion. Among all the nodes, only the k nearest ones to n are considered. Thus, each individual leader performs only a local search that is limited to a few number of its neighbor nodes. This will help to significantly reduce the high costs that are typically associated with refinement algorithms. At the same time, because every leader process performs such a local search simultaneously, the algorithm will still explore a large portion of all possible swappings.

Among all local swappings, each leader process saves (temporarily) the swapping that leads to the highest reduction in maximum congestion. Next, all the leader processes communicate with each other to collectively find the swapping that results in a globally lowest maximum congestion (across all leaders). The corresponding swapping is enforced by all the leaders, and the congestion information is updated accordingly. The rationale behind our swapping strategy is rooted in the fact that InfiniBand networks often use destination-based routing based on which the adjacent nodes will end up using different links. Also, note that Alg. 2 can be repeated multiple times to decrease congestion iteratively. In our experiments, the algorithm runs until there is no congestion reduction between two consecutive iterations, or it reaches 10 iterations (whichever happens earlier). This is used as a mechanism to avoid unbounded number of refinement iterations.

Complexity The loop in lines 5 to 17 of Alg. 3.2 performs d iterations, where d denotes the degree of the process topology graph. Each iteration takes $\mathcal{O}(kdl)$, where k and l denote

the local search radius and the number of links connecting two nodes, respectively. The dl term is for updating the link congestions. Consequently, the loop in lines 5 to 17 is of order $\mathcal{O}(kd^2l)$. After that, gathering the new maximum congestion values takes $\mathcal{O}(n)$. The lowest result can be found in $\mathcal{O}(n)$, and finally, updating the link congestions takes another $\mathcal{O}(dl)$. Thus, the complexity of Alg. 3.2 can be given by $\mathcal{O}(n + kd^2l)$.

3.4 Experimental Results and Analysis

We carry out all the experiments on the General Purpose Cluster (GPC) at the SciNet HPC Consortium [72]. GPC consists of 3780 nodes with a total of 30240 cores. Each node has two quad-core Intel Xeon Nehalem sockets operating at 2.53 GHz, with a total of 16 GB memory. The two sockets are connected to each other by Intel QPI links. Each socket forms a NUMA node with 8 GB of local memory and 8 MB of L3 cache. Approximately one quarter of the cluster is connected with non-blocking DDR InfiniBand, whereas the rest of the nodes are connected with a 5:1 blocked QDR InfiniBand. For our experiments, we only use the nodes with QDR InfiniBand. These nodes are interconnected via Mellanox ConnectX 40Gb/s InfiniBand adapters.

The network topology (the QDR partition) is a fat-tree consisting of two 324-port core switches and 103 36-port leaf switches. Fig. 3.8 shows the corresponding details. The numbers on the links represent the number of links that connect each pair of switches. In particular, each leaf switch is connected to 30 compute nodes and has 3 uplinks to each of the two core switches. Each core switch itself is in fact a 2-layer fat-tree consisting of 18 *line* and 9 *spine* switches (with 36 ports in each switch). Each line switch is connected to 6 leaf switches[2], and also has 2 uplinks to each of the 9 spine switches. Finally, it is worth noting that each node in the system runs Centos 6.4 along with Mellanox OFED-1.5.3-3.0.0, and we use MVAPICH2-2.0 and Scotch-6.0.0.

Due to limited access to the cluster, the experiments are executed for a limited number

[2]except for the last line switch which is connected to one leaf switch only.

of iterations and repetitions. Ideally, the experiments should be repeated many times so as to gather enough results to conduct various statistical analyses. However, this is not feasible for experiments that are conducted on production supercomputers such as GPC because it will require excessive resource allocation on the supercomputer. Such allocations are not granted due to the high demand that exists for accessing the compute resources on such systems.

In addition, the experiments are conducted on a semi-dedicated allocation on the cluster. Each compute node in the allocation is used exclusively by our experiments. The top-level network links and switches, however, are shared among all the jobs submitted to the cluster. Our design does not address congestion and interference from other jobs that might be running on a non-dedicated system. Decreasing congestion and interference related to other jobs falls into the scope of system-wide resource manager entities.

We use three microbenchmarks as well as four real applications to evaluate the performance of our proposed mapping approach. In addition, we also compare our proposed heuristics against those in the LibTopoMap library [45]. LibTopoMap provides 4 mapping algorithms: 1) a general greedy heuristic (Greedy), 2) an algorithm based on recursive graph bi-partitioning that uses the METIS library (Rec), 3) an algorithm based on matrix bandwidth reduction that uses the Reverse Cuthill McKee algorithm (RCM), and 4) the graph mapping functionality provided by the Scotch library (Scotch). In all cases, LibTopoMap first uses the ParMetis [61] graph partitioning library to handle multicore nodes.

We have integrated all the heuristics into the MPI distributed graph topology function to enforce particular mappings through rank reordering. The results report the improvements achieved over a block in-order mapping of processes. In a block mapping, adjacent ranks are mapped onto the same node as far as possible before moving to any other node.

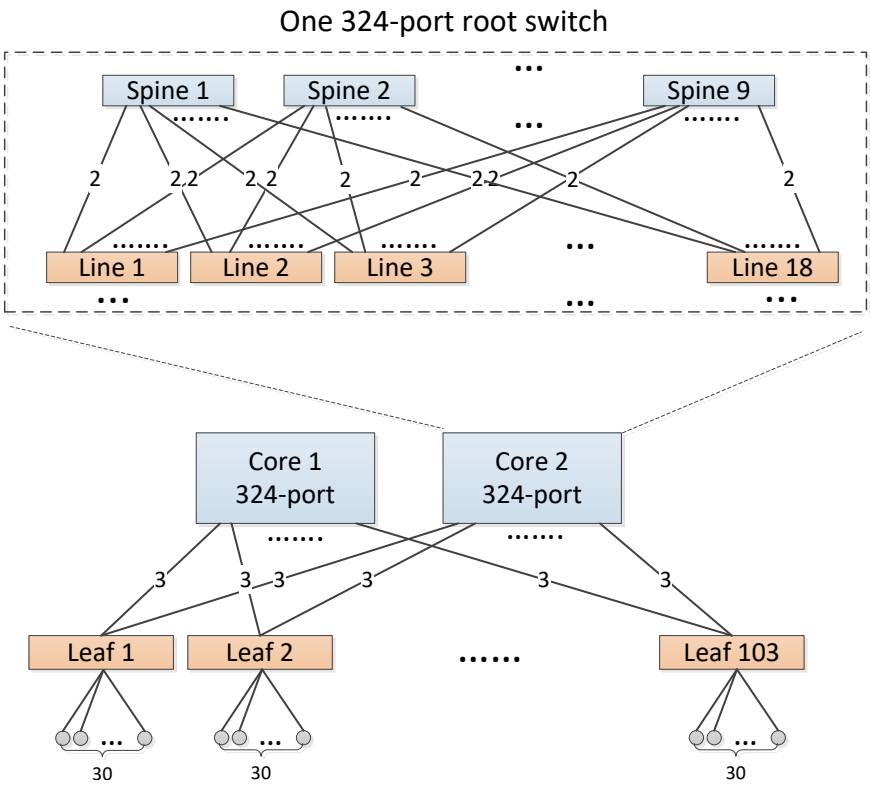
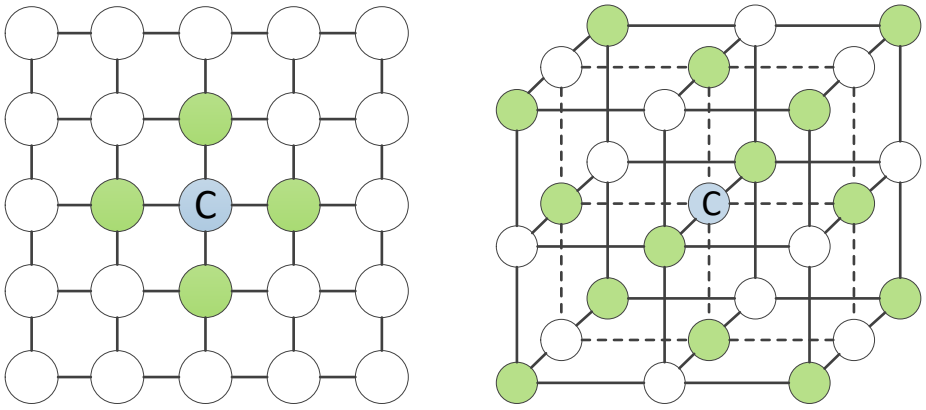


Figure 3.8: The network topology of the GPC cluster at SciNet—QDR partition.

3.4.1 Microbenchmark Results

For our evaluations in this section, we use three microbenchmarks that mimic the communication patterns seen in many parallel applications. The first microbenchmark models a 2D 5-point halo exchange, also known as 2D 5-point stencil. In this microbenchmark, the MPI processes are organized into a virtual 2-dimensional grid, and each process communicates with its two immediate neighbors along each dimension. The second microbenchmark models a 3D 15-point halo exchange in which MPI processes are arranged into a virtual 3-dimensional grid. Each process communicates with its two immediate neighbors along each dimension (6 neighbors), as well as 8 corner processes. Fig. 3.9 provides a schematic diagram of the 2D 5-point and 3D 15-point communication patterns.

We consider two versions of each of the 2D 5-point and 3D 15-point microbenchmarks:



(a) 2D 5-point (b) 3D 15-point

Figure 3.9: A schematic diagram of the 2D 5-point and 3D 15-point communication patterns. The neighbors are shown in green for the 'C' node only.

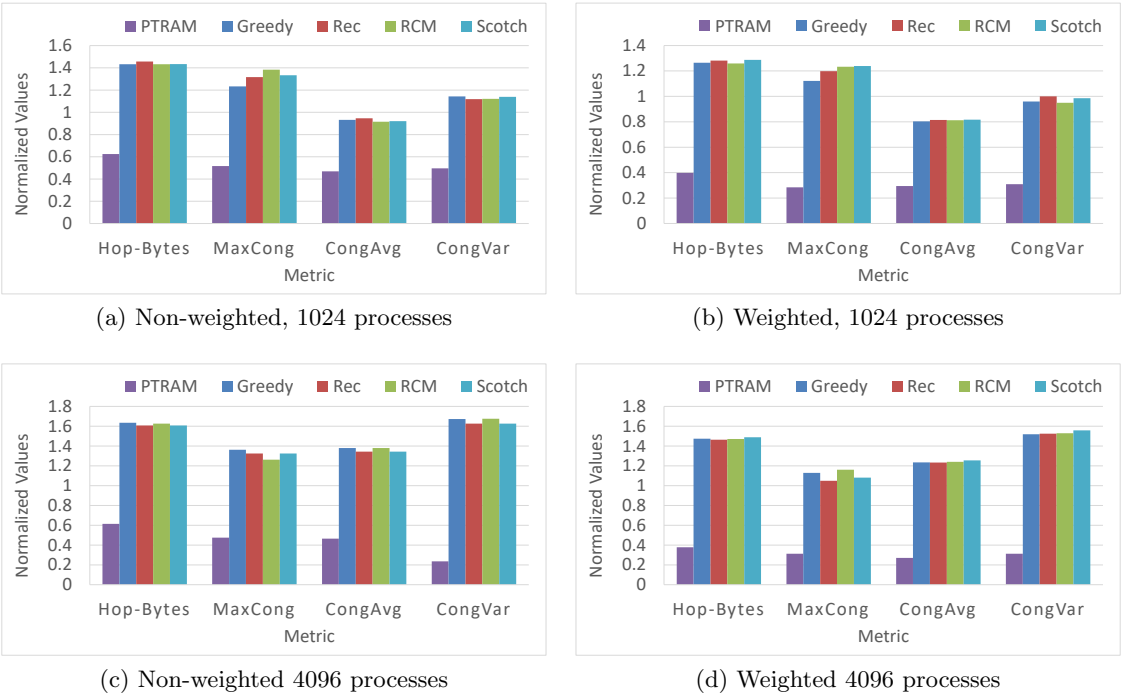
weighted and *non-weighted*. In the weighted version, we assign a higher weight to the communications carried over one of the dimensions of the logical process grid. This is done by sending messages of a relatively larger size over the weighted dimension. In particular, we use a weight factor of 3. In the non-weighted version, the same message size is used for communications along all dimensions.

The third microbenchmark models an all-to-all communication over sub-communicators. We consider two versions of this microbenchmark. In the first one, the processes are arranged into a logical 2-dimensional grid with each column representing a separate sub-communicator. In the second one, the processes are arranged into a 3-dimensional grid, and each plane along the first and last dimensions represents a separate sub-communicator. In both versions, an `MPI_Alltoall` is performed over each sub-communicator. Note that in the first version we will have linear sub-communicators (1D all-to-all), whereas in the second one we will have 2-dimensional sub-communicators (2D all-to-all).

Metric values

Fig. 3.10 to Fig. 3.12 show the resulting metric values for each microbenchmark with 1024 and 4096 processes. The numbers show the normalized values over the default mapping. As we can see, for all three microbenchmarks, PTRAM can successfully decrease the value of all four metrics. Moreover, it outperforms the heuristics provided by LibTopoMap. For the 2D 5-point microbenchmark in particular, we can see more than 60% reduction in the value of all four metrics in Fig. 3.10(d). The highest reduction is seen for congestion average which is about 73%. Maximum congestion has also been decreased by 68%. At the same time, we see that LibTopoMap has actually led to an increase in the metric values. PTRAM outperforms LibTopoMap for multiple reasons. In PTRAM, we explicitly evaluate all the potential nodes for mapping a process at each step. In addition, PTRAM takes into account the routing information, as well as the precise topology of the network which allows the algorithm to have a more realistic evaluation of various metrics. Moreover, using a hybrid metric provides a stronger means to distinguish different mappings from each other.

For the 3D 15-point microbenchmark we can see about 50% reduction in maximum congestion for PTRAM (Fig. 3.11). We can also see that LibTopoMap performs relatively better for the 3D 15-point microbenchmark than the 2D 5-point; there is a lower increase in the metric values, and maximum congestion has actually been decreased by about 7%. This is despite the fact that the value of the hop-bytes metric has increased at the same time. In addition, in both Fig. 3.10 and Fig. 3.11, we can see higher improvements achieved for the weighted versions of the 2D 5-point and 3D 15-point microbenchmarks. This has two reasons. First, the non-weighted versions of these microbenchmarks induce a communication pattern that is quite symmetric in the sense that optimizing the mapping for one part of the communication graph will adversely affect other parts. Second, the default in-order mapping happens to be a good match for the non-weighted version of these microbenchmarks. However, by using a different message size along one of the dimensions



(a) Non-weighted, 1024 processes

(b) Weighted, 1024 processes

(c) Non-weighted 4096 processes

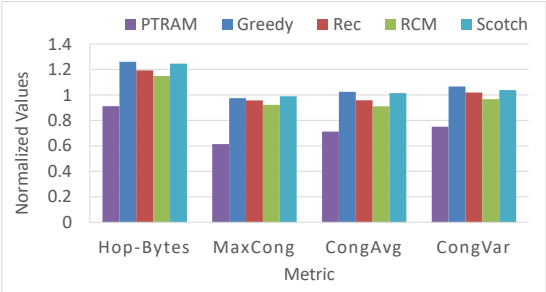
(d) Weighted 4096 processes

Figure 3.10: The normalized resulting metric values over the default mapping for the 2D 5-point microbenchmark and different mapping algorithms (lower is better)—1024 and 4096 processes.

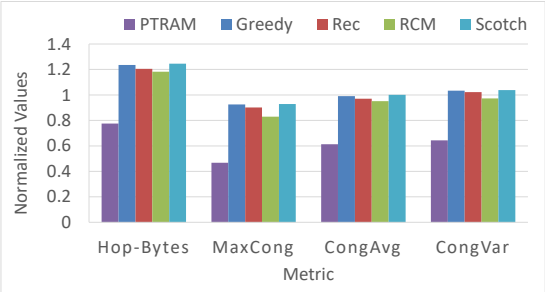
of the grid, the weighted versions of these microbenchmarks add some irregularity into the corresponding communication pattern and decrease its symmetry. This will provide higher opportunities for optimization through topology-aware mapping algorithms.

For the all-to-all microbenchmark results shown in Fig. 3.12, we see higher improvements for the version with the linear (1D) sub-communicators. Moreover, we see better results for LibTopoMap compared to the 2D 5-point and 3D 15-point microbenchmarks; the value of the metrics have either remained unchanged (Fig. 3.12(a)) or been improved (Fig. 3.12(b)). For instance, we can see about 20% improvement in the congestion average metric achieved by LibTopoMap heuristics in Fig. 3.12(b). However, it can be seen that PTRAM will still achieve higher improvements than LibTopoMap.

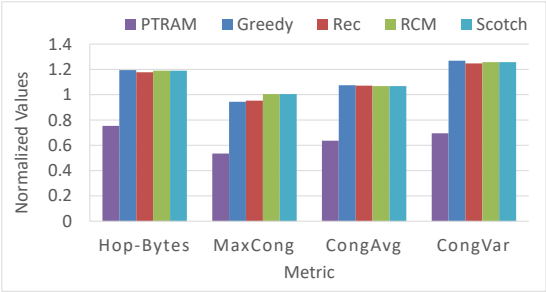
We are also interested in evaluating the performance of each heuristic with respect to



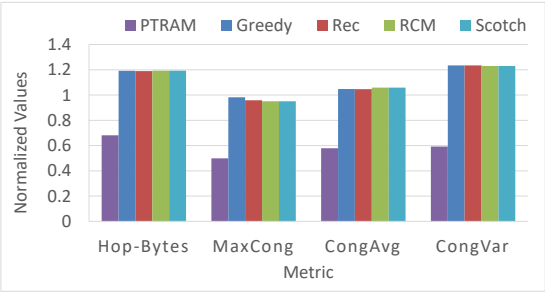
(a) Non-weighted, 1024 processes



(b) Weighted, 1024 processes

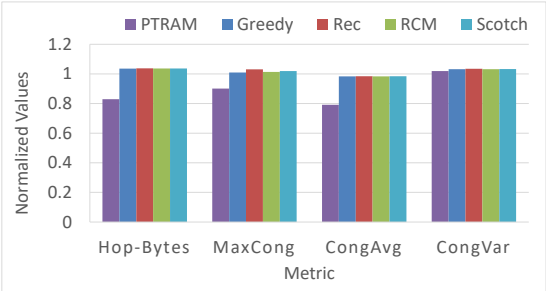


(c) Non-weighted 4096 processes

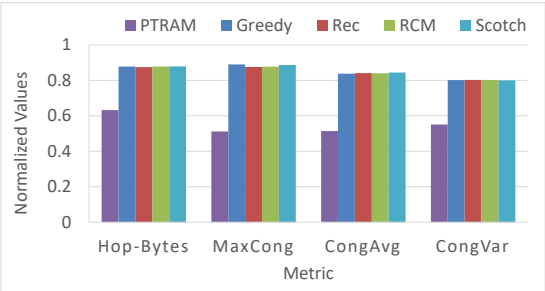


(d) Weighted 4096 processes

Figure 3.11: The normalized resulting metric values over the default mapping for the 3D 15-point microbenchmark and different mapping algorithms (lower is better)—1024 and 4096 processes.

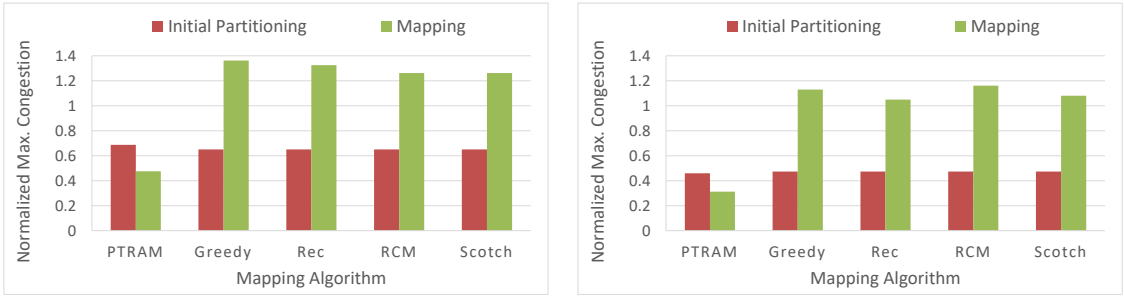


(a) 2D sub-communicator all-to-all, 1024 processes



(b) 1D sub-communicator all-to-all, 4096 processes

Figure 3.12: The normalized resulting metric values over the default mapping for the sub-communicator all-to-all microbenchmark and different mapping algorithms (lower is better)—1024 and 4096 processes.

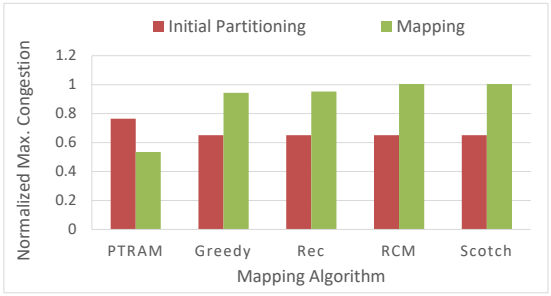


(a) Non-weighted (b) Weighted

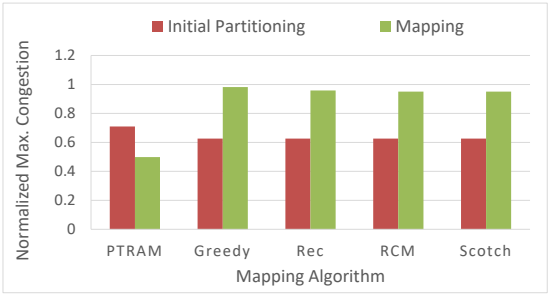
Figure 3.13: Normalized maximum congestion improvement details over the default mapping for the 2D 5-point microbenchmark with respect to the initial graph partitioning stage (lower is better)—4096 processes.

the initial partitioning stage that is used for multicore nodes. Recall that in both PTRAM and LibTopoMap, there are two major stages. First, a graph partitioner library is used to cluster multiple individual processes into a single group (we use Scotch for PTRAM, and LibTopoMap uses ParMetis). Second, the heuristics are used to collectively map each group of processes onto a particular node in the system. Thus, in Fig. 3.13 to Fig. 3.15, we show how the metric value improvements reported in previous figures are broken down across these two stages. Note that 'Initial Partitioning' shows the results corresponding to a simple in-order node mapping of process groups returned by the initial graph partitioning stage. The 'heuristic' represents the results for each particular heuristic. We only show these results for the maximum congestion metric and with 4096 processes. The trend is almost the same for other metrics and 1024 processes.

First, we can see that PTRAM has successfully decreased maximum congestion on top of the initial partitioning for all three microbenchmarks. Second, according to Fig. 3.13 and Fig. 3.14, the mapping results from LibTopoMap heuristics cause an increase in the final maximum congestion for the 2D 5-point and 3D 15-point microbenchmarks. In other words, an in-order node-mapping of process partitions returned by ParMetis in LibTopoMap results in a lower maximum congestion than those returned by the heuristics. This shows

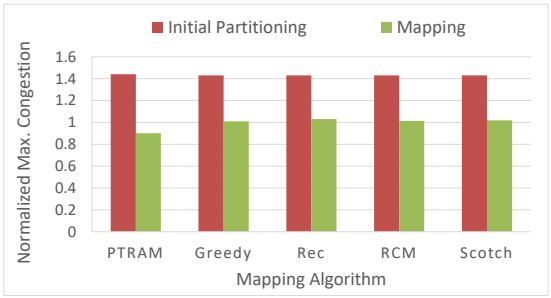


(a) Non-weighted

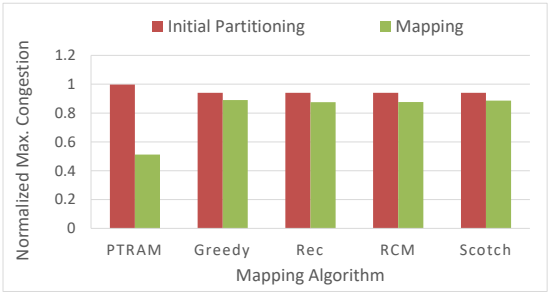


(b) Weighted

Figure 3.14: Normalized maximum congestion improvement details over the default mapping for the 3D 15-point microbenchmark with respect to the initial graph partitioning stage (lower is better)—4096 processes.



(a) 2D sub-communicator all-to-all, 1024 processes



(b) 1D sub-communicator all-to-all, 4096 processes

Figure 3.15: Normalized maximum congestion improvement details over the default mapping for the sub-communicator all-to-all microbenchmark with respect to the initial graph partitioning stage (lower is better).

the importance of having efficient topology-aware node-mapping algorithms. We can also see that the initial partitioning mostly provides a similar performance in both PTRAM and LibTopoMap. For the 3D 15-point microbenchmark however, LibTopoMap achieves a lower maximum congestion in the initial partitioning phase compared to PTRAM. This implies that ParMetis has done a better job than Scotch for the initial partitioning. Despite this, PTRAM still outperforms the other 4 heuristics by providing about 20% improvement on top of the initial partitioning stage.

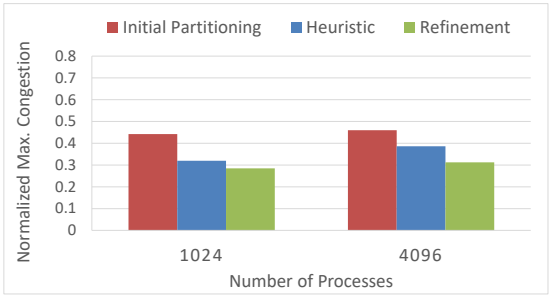
A different trend is seen for the all-to-all microbenchmark results in Fig. 3.15. For

PTRAM, all the improvement comes directly from the node-mapping algorithms. The initial partitioning stage has either worsened the maximum congestion (Fig. 3.15(a)) or has had no impact on it (Fig. 3.15(b)). Still, in both cases, PTRAM node-mapping algorithms have been able to decrease maximum congestion by about 50%. This is an important observation as it shows that an appropriate node-mapping of processes can still significantly decrease maximum congestion on top of an initial multicore partitioning. A similar trend is seen for the LibTopoMap heuristics as well. Also, unlike the case for the 2D 5-point and 3D 15-point microbenchmarks, LibTopoMap does not increase the maximum congestion after the initial partitioning stage of the all-to-all microbenchmark.

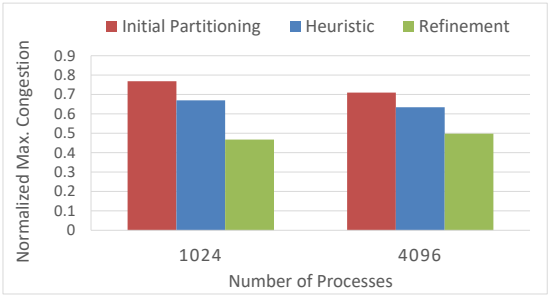
Finally, in Fig. 3.16, we show how maximum congestion improvements are further broken down across all three stages of PTRAM. In particular, the figure compares the greedy heuristic (Alg. 3.1) and the refinement algorithm (Alg. 3.2) of PTRAM in terms of their impact on maximum congestion reductions. As we can see, in most cases, both algorithms help to improve maximum congestion. However, the relative effect between the two algorithms depends on the specific pattern of the microbenchmark and/or the number of processes.

Communication time

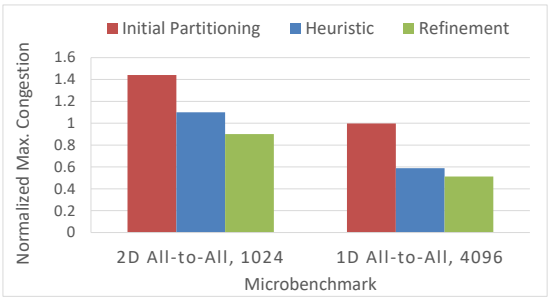
Fig. 3.17 to Fig. 3.19 present the communication time results for each microbenchmark. The results represent the average of 500 communication iterations for various message sizes. We report the improvement percentage with respect to the communication time corresponding to the default mapping. For message sizes above 32KB, PTRAM can provide a consistent improvement of about 25% and 20% for the weighted versions of the 2D 5-point and 3D 15-point microbenchmarks, respectively. However, we see that LibTopoMap mappings have led to an increase in communication time. This is expected with respect to the results shown previously for the metric values. As the mapping result from LibTopoMap increases the mapping metric values, the communication time is expected to increase. On the other hand,



(a) 2D 5-point weighted



(b) 3D 15-point weighted

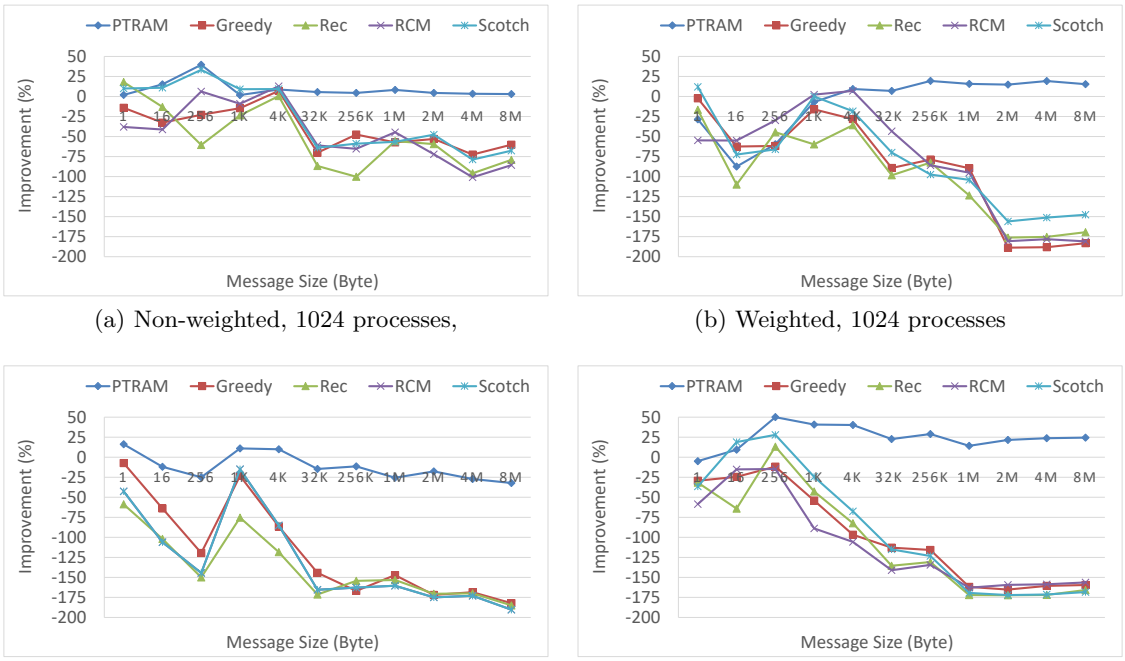


(c) Sub-communicator all-to-all

Figure 3.16: Normalized maximum congestion improvement over the default mapping breakdown across different steps of PTRAM for various microbenchmarks (lower is better).

we see that PTRAM's success in decreasing the metric values has actually been reflected in terms of lower communication times. For the non-weighted versions of the 2D 5-point and 3D 15-point microbenchmarks, we generally see lower improvements for similar reasons discussed in the previous section.

With the all-to-all microbenchmark and 2D sub-communicators (Fig. 3.19(a)), PTRAM is shown to provide up to 30% improvement in communication time for messages larger than 1KB. Moreover, with linear sub-communicators (Fig. 3.19(b)), both PTRAM and LibTopoMap have successfully reduced communication time across all message sizes. For messages below 4KB, the improvements are lower and almost the same for all heuristics. However, PTRAM starts to outperform LibTopoMap for larger messages, and provides up to 50% improvement versus about 30% provided by LibTopoMap. These results are again in accordance with the metric results shown in Fig. 3.12, where we see improvement in all



(a) Non-weighted, 1024 processes,

(b) Weighted, 1024 processes

(c) Non-weighted 4096 processes

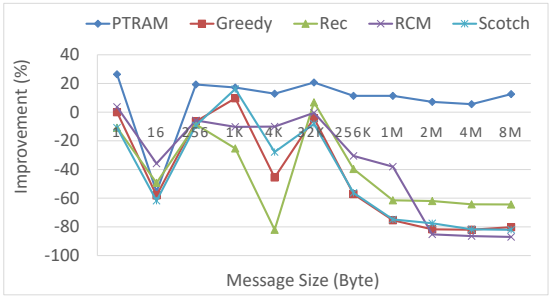
(d) Weighted 4096 processes

Figure 3.17: Communication time improvements over the default mapping for the 2D 5-point microbenchmark and different mapping algorithms—1024 and 4096 processes.

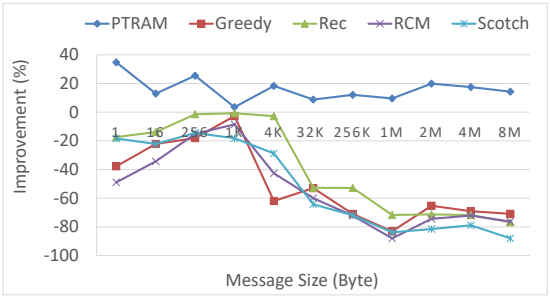
metric values for all heuristics. However, higher improvements achieved by PTRAM result in higher improvements in communication time for messages above 4KB. In general, we see a good correlation between the results for communication time improvements in Fig. 3.17 to Fig. 3.19 and those shown for the metrics in Fig. 3.10 to Fig. 3.12.

3.4.2 Overhead Analysis

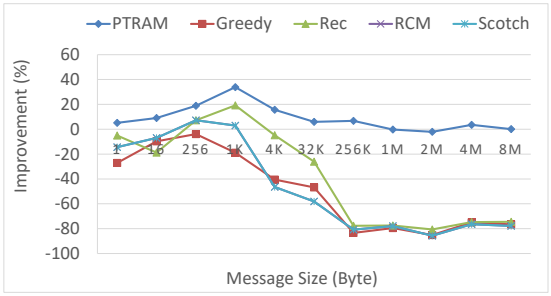
In this section, we report the overheads of our mapping approach, and compare it against LibTopoMap. Table 3.1 shows the total mapping time for each of the microbenchmarks with 4096 processes. The mapping time is almost the same for all 4 four heuristics provided by LibTopoMap. Therefore, we only report the time corresponding to one of them (i.e., the greedy heuristic). As shown by the table, PTRAM imposes a significantly lower overhead compared to LibTopoMap for all three microbenchmarks. This is mainly due to



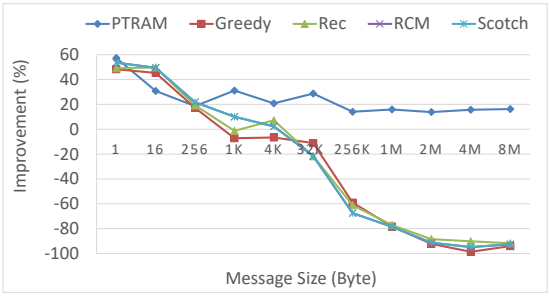
(a) Non-weighted, 1024 processes,



(b) Weighted, 1024 processes

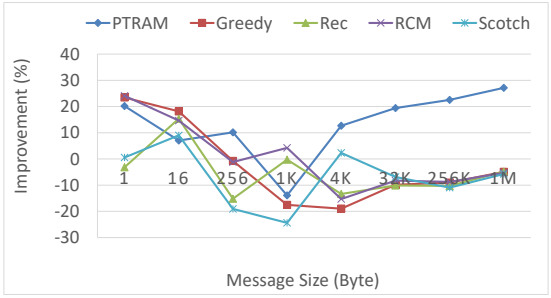


(c) Non-weighted 4096 processes

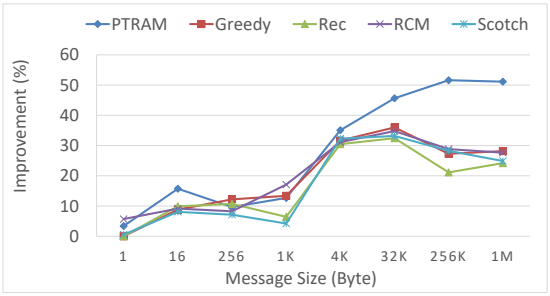


(d) Weighted 4096 processes

Figure 3.18: Communication time improvements over the default mapping for the 3D 15-point microbenchmark and different mapping algorithms—1024 and 4096 processes.



(a) 2D sub-communicator all-to-all, 1024 processes



(b) 1D sub-communicator all-to-all, 4096 processes

Figure 3.19: Communication time improvements over the default mapping for the sub-communicator all-to-all microbenchmark and different mapping algorithms—1024 and 4096 processes.

Table 3.1: Total time spent by PTRAM and LibTopoMap to find the mapping for each microbenchmark—4096 processes

	PTRAM	LibTopoMap
2D 5-Point	2.38 s	40.67 s
3D 15-Point	5.00 s	88.55 s
all-to-all	22.33 s	367.95 s

the underlying parallel design of our heuristics that allows for fast (and yet high-quality) mappings. In particular, PTRAM spends 2.38 and 5 seconds to derive the mapping for the 2D 5-point and 3D 15-point microbenchmarks, respectively. This increases to 22.33 seconds for the sub-communicator all-to-all microbenchmark. The reason is that in the all-to-all microbenchmark, each process communicates with a relatively higher number of other processes. Therefore, the corresponding communication pattern matrix will be denser, which in turn adds to the volume of computations performed by each node-leader. The same behavior is seen for LibTopoMap as well.

Table 3.2 shows the overhead break down of PTRAM across various stages. As expected, the majority of the overhead comes from the heuristic, and the refinement stage imposes only a slight overhead. The "Miscellaneous" column represents the total time spent in other parts of our implementation not captured in the other three stages. This mainly includes loading various data structures such as the communication pattern matrix, etc.

In practice, mapping costs are considered to be incurred only once for each application. The corresponding output mappings can be saved and reused later on for subsequent runs of an application. Also, many HPC applications are run over a long course of time (e.g., several days) which amortizes mapping overheads. However, it is still important for the mapping algorithms to impose lower amounts of overhead. This is particularly true in batch production systems where the user is given a dynamic allocation for which the previously calculated mapping results cannot be reused.

Finally, in Fig. 3.20 we show the mapping overheads for three different numbers of

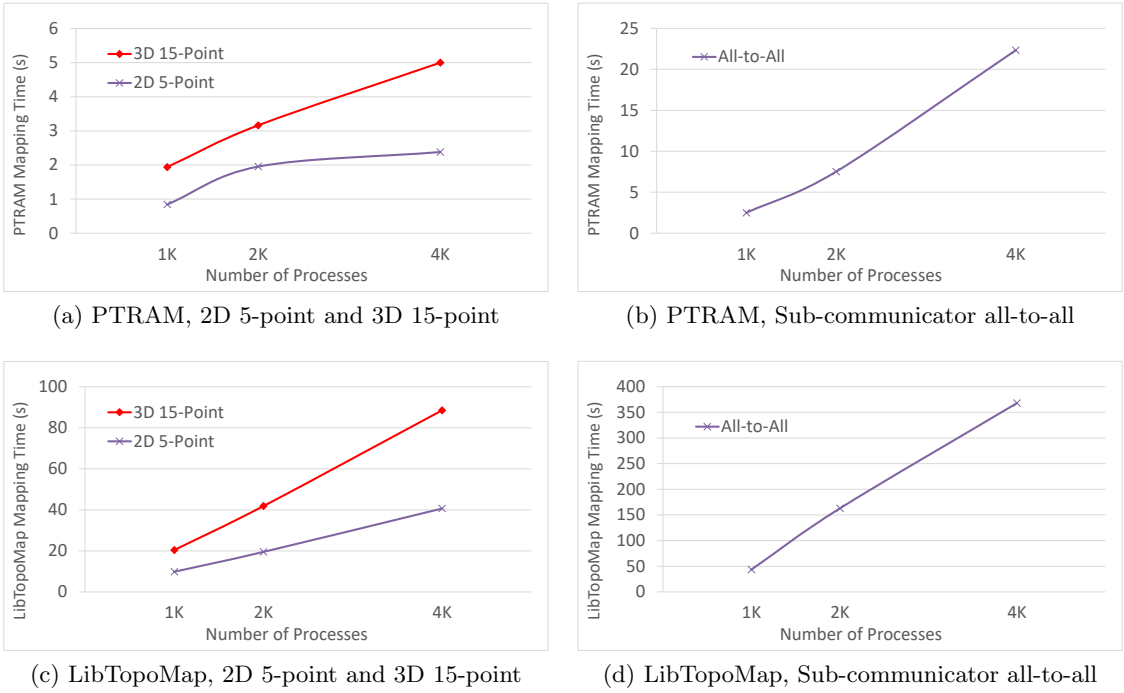
Table 3.2: The overhead breakdown across various stages of PTRAM for each microbenchmark—4096 processes

	Initial Partitioning	Heuristic	Refinement	Miscellaneous
2D 5-Point	0.04 s	1.64 s	0.04 s	0.64 s
3D 15-Point	0.12 s	2.33 s	0.71 s	1.82 s
all-to-all	0.38 s	13.00 s	0.63 s	8.30 s

processes. According to Fig. 3.20(a) and Fig. 3.20(b), PTRAM shows a better scalability with the 2D 5-point and 3D 15-point microbenchmarks compared to the all-to-all microbenchmark. This is again rooted in the difference between these microbenchmarks in terms of the sparsity of their corresponding communication patterns. With the all-to-all pattern, the increase in the number of processes also increases the number of neighbors of each process, adding more computation load to node-leaders. With the 2D 5-point and 3D 15-point patterns however, the number of neighbors of each process does not increase with the number of processes and remains fixed (4 for 2D 5-point and 14 for 3D 15-point). Compared to LibTopoMap results shown in Fig. 3.20(c) and Fig. 3.20(d), we can see a better scalability for PTRAM and the 2D 5-point/3D 15-point microbenchmarks, whereas LibTopoMap performs better for the all-to-all microbenchmark. The increase in the number of neighbors has a higher impact on PTRAM overheads because in PTRAM we consider all the neighbors of each process in various stages of our algorithms.

3.4.3 Application Results

In this section, we evaluate our proposed mapping approach in the context of real parallel applications. To this end, we use four applications: MiniGhost [8], Nbody [107], Radix [107], and Gadget [109]. MiniGhost is a finite difference mini application which implements a difference stencil across a homogeneous three-dimensional domain. Nbody simulates the interaction of a system of bodies in three dimensions over a number of time steps. Radix is a parallel implementation of the radix sort algorithm for integers that is mainly used as a sorting component in other applications. Gadget [109] is an application for various



(a) PTRAM, 2D 5-point and 3D 15-point

(b) PTRAM, Sub-communicator all-to-all

(c) LibTopoMap, 2D 5-point and 3D 15-point

(d) LibTopoMap, Sub-communicator all-to-all

Figure 3.20: PTRAM and LibTopoMap overheads with different number of processes and various microbenchmarks.

cosmological simulations, ranging from colliding and merging galaxies, to the formation of large-scale structure in the Universe.

We first report the results for the first three applications with 1024 processes. Fig. 3.21 shows the normalized execution time of MiniGhost, Nbody, and Radix with 1024 processes and different mapping approaches. The values have been normalized with respect to the execution time of each application with the default mapping. The absolute values of the execution times are reported in Table 3.3. As we can see, for MiniGhost, the execution time remains unchanged, whereas for Nbody and Radix, both PTRAM and LibTopoMap successfully decrease the execution time. In particular, PTRAM is shown to decrease Nbody's execution time by about 70%, and outperforms LibTopoMap which provides about 20% improvement. For Radix however, we see a similar performance for PTRAM and LibTopoMap heuristics, and they all provide 30% reduction in execution time.

Table 3.3: Absolute values of the execution times of applications with different mappings—1024 processes.

	Default	PTRAM	Greedy	Rec	RCM	Scotch
MiniGhost	20.85 s	20.76 s	20.42 s	20.47 s	20.15 s	20.39 s
Nbody	114.59 s	38.21 s	89.54 s	90.61 s	94.00 s	80.95 s
Radix	17.86 s	12.81 s	12.96 s	13.29 s	12.59 s	13.66 s

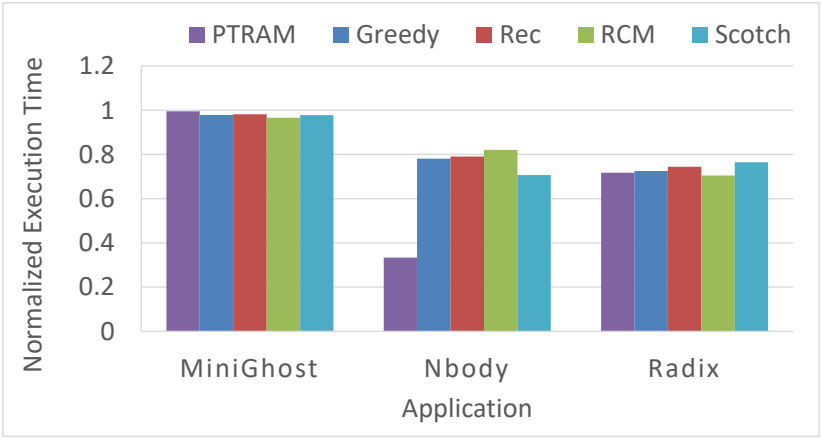
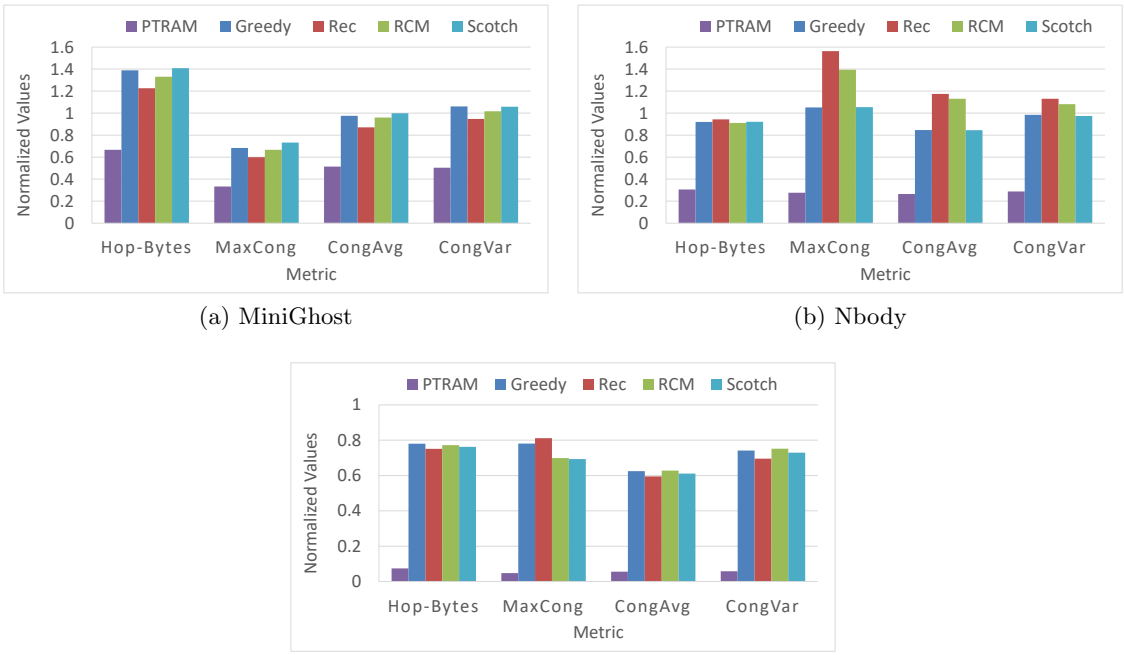


Figure 3.21: Normalized execution time improvements over the default mapping for three applications with PTRAM and LibTopoMap (lower is better)—1024 processes.

For further analysis, we show how the metric value results for these applications in Fig. 3.22. The first observation is that for all applications, PTRAM has successfully improved the metric values. For MiniGhost (Fig. 3.22(a)) however, these improvements do not lead to execution time reductions. This is because the ultimate execution time of an application depends on many different factors, and that an application may not even be communication-bound. The second observation is the increase in Nbody's maximum congestion with LibTopoMap heuristics (Fig. 3.22(b)). Despite the increase in maximum congestion, LibTopoMap could still decrease Nbody's execution time in Fig. 3.21. This is because we see improvement in other metrics, especially hop-bytes. However, since PTRAM also improves the value of other metrics (such as maximum congestion), it results in a lower execution time for Nbody. These results show that the mapping problem cannot be



(a) MiniGhost

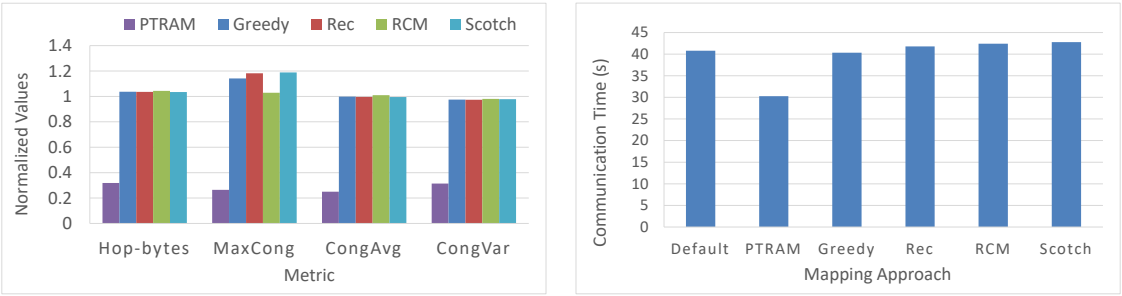
(b) Nbody

(c) Radix

Figure 3.22: Normalized mapping metrics improvements over the default mapping for three applications with PTRAM and LibTopoMap (lower is better)—1024 processes.

easily simplified into a single metric; it is required to evaluate mappings with respect to various criteria (multiple metrics) that can help to capture different characteristics of a given mapping. In this regard, an important positive point for PTRAM is the consistent improvement seen for all metrics across different microbenchmarks and applications.

For larger-scale application results, we use the Gadget application. In particular, we use the version 3 of the application [35] provided by PRACE [98], and run it with 4096 processes. Fig. 3.23(a) shows the normalized metric values. PTRAM is seen to decrease the metric values to about 0.3 of the default mapping. LibTopoMap heuristics mostly leave the metric values intact, though we can see an increase in maximum congestion with three of the heuristics. Similar to the MiniGhost application, no execution time improvements were obtained for Gadget. However, Fig. 3.23(b) shows the communication time of the application for different mappings. We can see that PTRAM provides 10 seconds reduction



(a) Metric values

(b) Communication time

Figure 3.23: Communication time and normalized mapping metrics improvements over the default mapping for the Gadget-3 application (lower is better)—4096 processes.

in communication time of the application compared to the default mapping, and again outperforms the other heuristics.

3.5 Related Work

Several experiments carried out on large-scale systems such as the IBM Blue Gene and Cray supercomputers verify the adverse effects of high contention and hop-counts on message latencies [14]. Balaji et al. [7] have shown that different mappings of an application on large-scale IBM Blue Gene/P systems can significantly affect the overall performance. Accordingly, they propose a framework for finding the mapping with the least amount of contention among different mapping options supported by the system. Bhatelé et al. [12] show the impacts of task mapping on application performance in the IBM Blue Gene/Q systems with a 5D torus network topology. Jain et al. [53, 13] conduct similar studies for the Dragonfly topology with different routing and job placement strategies. Bhatelé and Kalé [15] propose several heuristics based on the hop-bytes metric for mapping irregular communication graphs onto mesh topologies. The heuristics generally attempt to place highly communicating processes closer together.

Mercier and Clet-Ortega [80] use the Scotch library [95] for topology-aware mapping. The communication pattern of the application is first modeled as a complete weighted graph

where the volume of messages represent the edge weights. The physical topology is modeled by another complete weighted graph with the edge weights determined based on the memory hierarchies among the cores. Rodrigues et al. [104] propose a similar approach that uses Scotch as the underlying mapping algorithm. However, they use the communication speed (measured by a ping-pong benchmark) between each pair of cores as the metric for assigning edge weights in the physical topology graph. Rashti et al. [101] use the hwloc library [16] and InfiniBand tools to extract the intra-node and network topologies respectively. The results are merged to build an undirected graph with edge weights representing the communication performance among the cores based on the distances among them. The mapping is then performed by the Scotch library. Ito et al. [52] propose a similar mapping approach except that they use the actual bandwidth among nodes measured at the execution time to assign the edge weights in the physical topology graph. The mapping itself is again done by the Scotch library. Jeannot and Mercier [55, 81, 56] also use hwloc to extract the physical topology of a single node, and represent it as a tree. For the mapping algorithm, they propose TreeMatch which partitions the processes into a multi-level tree, and matches its nodes with their corresponding peers in the physical topology tree.

Hoefler and Snir [45] consider the mapping problem in its general form for arbitrary process topologies and interconnects. They model the process topology by a directed weighted graph G and use the volume of communication to designate the edge weights. The target network is also modeled by a directed weighted graph H with edge weights representing the bandwidth of the channel connecting two nodes. Three heuristics are proposed to map G onto H. In the greedy one, the vertices with higher individual outgoing edge weights are placed as close as possible to each other. The second heuristic uses a graph partitioning library (METIS) to perform the mapping by recursive bi-partitioning of G and H into two equal-sized halves with minimum edge-cut weights. The third heuristic is based on graph similarity, and models the mapping problem in terms of a bandwidth reduction problem for

sparse matrices. Considering adjacency matrices of the graphs representing G and H, matrix bandwidth reduction techniques are used to first bring the two matrices into a similar shape and then find the final mapping. Average dilation and maximum congestion are the two metrics used for assessing the quality of mappings.

Chung et al. [20] propose a hierarchical mapping algorithm which uses the normalized cut algorithm from spectral graph theory to partition the process and physical topology graphs into a number of *supernodes*. Then, an initial mapping assigns the supernodes of the process topology graph to their corresponding peers in the physical topology graph. A final pairwise exchange is also used to fine-tune the mapping further. Sudheer and Srinivasan [115] propose three mapping heuristics based on the hop-bytes as the metric. The first heuristic finds the local optima among several random initial solutions, and chooses the one with the lowest hop-bytes value. The second heuristic is a modified version of the greedy heuristic proposed by Hoefler and Snir [45]. The third heuristic is a combination of the first heuristic and a graph partitioning technique.

Li et al. [69] model the physical topology into separate network and node trees. The mapping is done in two phases. First the process topology graph is partitioned (using Scotch) to build a tree of process partitions isomorphic to the network tree. Next, the leaves of the resulting process tree are identically mapped onto the leaves of the network tree. A similar approach is used to perform the mapping within each node. Deveci et al. [23] use the Multi-Jagged geometric partitioning algorithm for task mapping with a focus on the mesh/torus systems that allow non-contiguous allocation of nodes. The algorithm partitions tasks and cores into the same number of parts. Tasks and cores with the same part numbers are then mapped onto each other. Tuncer et al. [121] propose PaCMap; an algorithm to perform allocation and mapping simultaneously. Pacmap first uses METIS to partition the communication graph for multicore nodes. Then, the nodes are chosen by an expansion algorithm which assigns a score value to each node with respect to the number of free nodes close to it. Tasks are mapped by iteratively finding the one with the highest

communication volume, and mapping it onto the node that leads to the least hop-bytes value.

Abdel-Gawad et al. [1] propose RAHTM which uses a three-phased approach for mapping. The first phase clusters tasks for multicore nodes. The second phase uses mixed integer linear programming to hierarchically map the clusters onto a series of 2-ary n-cubes. A greedy merge heuristic is used in the third phase to combine subproblem mappings. RAHTM uses maximum congestion as the metric, and also takes into account the underlying routing in the second and third phases. We exploit routing information in our proposed algorithms too. However, RAHTM is more suited for mesh/torus topologies, whereas we mainly target indirect fat-trees. Zimmer [126] et al. utilize machine learning techniques to derive job placement features that correlate with applications performance. Using the corresponding result, they propose the Dual-Ended scheduling mechanism for the Titan supercomputer that can decrease hop-count and improve performance.

Deveci et al. [22] propose a greedy mapping heuristic plus two other refinement algorithms. The greedy algorithm as well as one of the refinement algorithms are designed to minimize hop-bytes, whereas the second refinement algorithm is designed to decrease congestion. The greedy heuristic iteratively chooses the process with the highest connectivity to the set of mapped processes, and assigns it to the node that results in the lowest value of hop-bytes. The corresponding node is found by performing a BFS search on the physical topology graph. The refinement algorithms attempt to decrease hop-bytes or congestion by task swapping. Unlike the work done by Deveci et al., we do not use any graph search algorithm in our greedy heuristic; we find the desired target node by explicit evaluation of each node. In addition, we use a hybrid metric rather than hop- bytes. Moreover, we exploit parallelism in the design and implementation of our algorithms.

3.6 Summary

Modern multicore parallel computing systems provide different types of communication channels at different layers of the system hierarchy. We demonstrated through experimentation that the performance of these channels can be considerably different from each other. Therefore, the specific placement of the processes of an application on the physical cores of the target system can have a significant impact on the overall communication latency. In this regard, topology-aware mapping techniques attempt to find a nontrivial placement of processes based on the application communication pattern and the physical topology of the system. However, the NP-hard nature of the mapping problem problem on one hand, and the ever-increasing size and complexity of HPC systems on the other hand, make it quite challenging to find a high-quality and yet scalable solution to the problem.

In this chapter, we proposed a parallel topology- and routing-aware mapping approach. To achieve a better mapping quality, we added routing awareness to our design by taking into account the routing information gathered from the interconnection network. Using the routing information, we could precisely measure the static congestion imposed on each link by any candidate mapping. This helped to have a more realistic measure of our mapping metrics which in turn led to find a better mapping. We also used a hybrid metric in our design and discussed how it helps to better distinguish different candidate mappings from each other and choose those that would potentially lead to a better distribution of traffic across all links. To increase scalability and decrease overheads, we used parallelism in our design by making each node responsible for evaluating itself with respect to the metrics. Our experimental results showed that using our proposed mapping approach, we can improve communication performance for various communication patterns compared to both a trivial mapping as well as some other nontrivial approaches. The results also showed that we can achieve lower overheads with our proposed algorithms.

Chapter 4

Topology-Aware Mapping Heuristics for Collective Communications

In Chapter 3, we discussed how topology awareness can help improve communication performance through a nontrivial placement of processes onto the set of compute nodes within a large-scale system. The proposed mapping approach considered all the application communications as a whole, including both point-to-point and collective communications. However, mapping and topology awareness can also be specifically geared towards the collective communications so as to fine-tune their performance further.

Collective communications constitute a major portion of the communications that occur in parallel applications, and their performance plays a critical role in the total performance of the application [100]. Accordingly, various algorithms have been proposed for efficient implementation of collective communications. However, the core merit of such algorithms is only achieved with an appropriate mapping of processes because the performance of a given collective can significantly change under different mappings of processes [78]. The appropriate mapping depends on the physical topology of the target system which can change from one system to another.

In this chapter, we focus on topology-aware collective communication design. In particular, we propose four fine-tuned mapping heuristics for various communication patterns

and algorithms commonly used in `MPI_Allgather` [83]. The allgather operation is a data-intensive collective communication that can be a major contributor to the communication time of an application. This operation can also be used as a building block for other collective operations such as broadcast and allreduce. Two of our proposed heuristics can also be used for `MPI_Bcast` and `MPI_Gather` operations. For our work in this chapter, we take into account the intra-node physical topology as well so as to widen the scope of topology awareness in our design.

4.1 Collective Pattern, Topology, and Mapping

The well-known generic collective communication algorithms have mainly been designed with respect to Hockney's $\alpha + m\beta$ performance model [40]. In this model, α represents the startup latency cost for each message, and $m\beta$ is representative of the bandwidth requirements. Specifically, m denotes the message size and β represents the bandwidth cost per byte. The α term of a collective algorithm is simply the number of its stages, whereas the β term is found by summing the maximum message size communicated in each stage over all communication stages. Depending on the amount of data involved in a collective communication, the strategy for reducing the cost of the operation differs. When the amount of data is small, it is the cost of initiating messages (α) that dominates and hence, the algorithms should strive to reduce this cost.For large data, it is the cost per item sent (β) that becomes the dominating factor and hence, the algorithms should focus on the bandwidth requirements.

In general, the generic algorithms often deliver good performance for collective communications. Despite the fact that they have been designed in 1990s (and earlier), they still represent the state-of-the-art from an algorithmic point of view. However, a major shortcoming of such algorithms is that they use predefined communication patterns built solely based on MPI logical ranks. The underlying hardware architecture and the specific

mapping of processes is totally ignored by these algorithms. But, both of these factors can considerably affect the performance of collective communications.

When used for collective communication design, Hockney's model implies the following as its underlying assumptions:

1. Communication performance is the same between all pairs of cores. In other words, the whole system is assumed to have a flat architecture.

2. There is full concurrency of communications within each individual stage of a collective. In other words, it is assumed that message exchanges that fall within a single stage can all happen simultaneously without any contention.

In practice, none of the above actually holds. The first assumption is wrong due to the heterogeneity of communication channels in modern multicore clusters. The second assumption does not necessarily hold due to potential congestion among communications within each single stage. Rico-Galego and Díaz-Martín [103] show that even inside an individual node, congestion prohibits full concurrency of communications due to memory bandwidth sharing among cores. As a result, the actual performance of generic algorithms developed on top of Hockney's model could be lower than those modeled in theory. The characteristics of the underlying architecture could prohibit the core merits of such algorithms to be achieved.

More specifically, the performance of a collective communication is the result of interaction among the following three main components:

1. the physical topology of the target system,

2. the mapping of processes onto nodes/cores, and

3. the specific communication pattern used by the collective.

The first component is fixed and cannot be changed dynamically to optimize collectives. However, one can attempt to manipulate the second and third components as a way to

optimize collective communication performance with respect to the underlying topology of the target system. Therefore, we can consider the following two approaches for topology-aware collective communication designs:

1. Changing the mapping of processes with respect to the collective communication pattern (i.e., the algorithm) and the physical topology of the target system

2. Changing the collective communication pattern with respect to the mapping of processes and the physical topology of the target system.

Our work in this chapter falls within the scope of the first approach mentioned above. Almost all previous efforts on collective communication optimizations devise the corresponding algorithms based on MPI's logical process ranks. A key factor missing in such algorithms is the actual mapping of the logical ranks onto the physical cores of the target system. Hence, collective communications have been devised in a mapping-ignorant fashion. In fact, current mapping-ignorant algorithms implicitly assume a regular in-order mapping of process ranks onto the processing elements where adjacent ranks end up residing on the neighboring cores. This assumption plays an important role in performance modeling of various collective algorithms proposed in the literature. It also acts as an implicit factor to favor one algorithm over another in current implementations of MPI libraries. However, such an assumption does not necessary hold in practice.

For instance, topology-aware mapping of processes to processor cores which is commonly performed based on bulk communications of an application can result in a different ordering of processes. Moreover, in modern large-scale systems, a job can potentially be mapped and allocated resources in quite a large number of different ways. Resource management tools commonly used for starting parallel jobs provide various options that affect how different processes are mapped onto the target system. For instance, both SLURM [124] and Hydra [48] provide various options for choosing the number and order of nodes, sockets, and cores assigned to each job. In addition, collective communications over sub-communicators

happen within a subset of processes that can reside over a random set of nodes. On the other hand, recent studies show that both topology-aware mapping and collective communications over sub-communicators tend to be inevitable at the exascale level [10].

4.2 Topology-Aware Rank Reordering for Collective Communications

4.2.1 The Framework

Variations of communication performance at different levels of system hierarchies along with the variety of mapping options used in large-scale systems necessitate topology and mapping awareness for collective communications. A general topology-aware mapping technique such as the one discussed in Chapter 3 can implicitly consider collective communications as they are implemented in terms of a series of point-to-point communications. However, various reasons limit the efficiency of such approaches for optimizing collective communications. First, different collectives have different communication patterns which makes it very difficult to optimize all of them with the same mapping of processes. Second, a general optimization of point-to-point communications can even leave some collective communications out in favor of higher-weight point-to-point communications. Third, unlike the native point-to-point communications, it is difficult to precisely capture the communication pattern of collectives because it depends on different factors such as message and communicator size.

Alternatively, we can change the mapping of processes for each collective communication pattern individually at runtime. In particular, we keep the collective algorithms intact, and reorder the ranks with respect to a mapping from the collective communication pattern to the physical topology of the system. The broad objective of the mapping is to decrease the physical distances among the communicating processes, specially those that communicate more often and/or use larger messages. For the physical topology, we use the hwloc library [16] and the InfiniBand [51] tools to extract the distances among all the cores. For each

system, the physical distances need to be extracted only once and will be reused for different collective communication patterns.

To perform the mapping, various approaches can be used. For instance, one can use an external graph mapping library such as Scotch [95] which employs graph partitioning techniques to recursively partition and map a given *guest* graph (communication pattern) onto a *host* graph (physical topology). Alternatively, fine-tuned heuristics can be devised for each collective communication pattern to achieve better mapping results and avoid some overheads (such as creating process topology graphs). In the following, we propose four such heuristics for various communication patterns commonly used in `MPI_Allgather`.

Having found the mapping, a new communicator is created with process ranks reordered based on the mapping results. The entire process can be repeated to create reordered communicators for each desired collective communication pattern. Later, any subsequent calls to the corresponding collective in an application will be conducted over the reordered copy of the given communicator. Note that the entire rank reordering process happens only once at runtime. In addition, we can also use an MPI `info` key to allow the programmer to enable/disable the whole approach for each communicator separately.

4.2.2 Mapping Heuristics for `MPI_Allgather`

In the general forms of topology-aware mapping, the communication pattern changes from one application to another. Therefore, the design of the corresponding mapping heuristics should be generic enough so as to handle various arbitrary communication patterns. To this end, the communication pattern is provided to the mapping algorithm in terms of a weighted graph which is later traversed (usually by a greedy approach) to figure out the desired mapping. For the allgather collective communication, we improve this general framework in two aspects[1].

First, for a particular collective such as allgather, the communication patterns are fixed

[1]In fact, with an appropriate set of mapping heuristics, the improvements apply to other collectives too.

and hence, there is no need to employ a generic mapping algorithm for all of them. Instead, we can utilize fine-tuned heuristics for each pattern so as to achieve better mapping results. Second, with fine-tuned heuristics, it is not required to build a process topology graph to describe the communication pattern to the mapping heuristic. The communication patterns can be systematically derived from the corresponding algorithms used by the MPI library. Therefore, with our proposed heuristics, we can skip the step for building the collective topology graph and proceed directly to the mapping step. However, with a generic mapping library such as Scotch, we still need to build the collective topology graph first.

For our heuristics, we pursue the following three main goals:

1. they should be capable of modifying the initial layout of processes so as to closely match it to each collective communication pattern even if the initial mapping is quite far from ideal,

2. they should not cause performance degradation if the initial layout of processes is already a good match for the target collective communication pattern, and

3. they should not impose any significant overheads.

General scheme of the heuristics

Alg. 4.1 illustrates the general scheme of our proposed mapping heuristics used in rank reordering for `MPI_Allgather`. The algorithm takes as input the physical distances in the form of a square matrix D, and outputs a mapping array M that represents the new rank of each process. In the first step and without loss of generality, the process with rank 0 is fixed on the core that already hosts it. Next, in Steps 3 to 8, we iteratively choose a new process and find a target core for hosting it. The target core is always chosen with respect to what we call a *reference core*. The reference core is a core that has already been assigned to one of the processes in some previous iteration of the algorithm. The target core is always chosen to be a free core that has the minimum distance from the reference

core. If more than one core satisfies this condition, one of them is chosen randomly. Having mapped the new process, the reference core *might* be updated in Step 7. Initially, the core hosting rank 0 is set as the reference core in Step 2.

Alg. 4.1 maps each process as close as possible to some other process that has already been mapped onto the system. The important question is how we should choose the next process and the reference core in each iteration. In other words, the key steps in Alg. 4.1 are Steps 4 and 7. Step 4 determines the order in which we select the processes for mapping, whereas the choice of the reference core in Step 7 implicitly designates the group of processes that will be placed closer to the selected process. The strategies used in these two steps depend on the underlying communication pattern among the processes and hence, vary with each particular allgather algorithm. That is why in Step 7 of Alg. 4.1 we say the reference core is updated *if necessary*. In the following, we will propose a specific version of Alg. 4.1 for different communication patterns commonly used by an allgather operation. In particular, we cover recursive doubling, ring, binomial-tree broadcast, and binomial-tree gather. It is important to note that hereinafter, we interchangeably use process ranks to refer to a particular process or the core hosting it.

Algorithm 4.1: General scheme of our topology-aware mapping heuristics for collective communications

Input : Number of processes p, Physical topology distance matrix D
Output: Mapping array M representing the new rank for each process

1 Fix rank 0 on its current core;
2 Choose 0 as the reference core;
3 **while** *there exist more processes to map* **do**
4 Select a new process to map;
5 Find a target core among the free cores. The target core is a core that has not already been assigned to any other process and has a minimum distance from the reference core;
6 Map the new process onto the target core;
7 Update the reference core with the target core *if necessary*;
8 **end**

Recursive doubling

At each stage of the recursive doubling algorithm, the processes communicate in pairs, and the communications happening at further stages of the algorithm represent larger messages. Therefore, in our heuristic for recursive doubling we will try to choose the new process based on the communications of further stages as much as possible. Accordingly, we start with the pairs of communications that fall in the last stage. The first process selected for mapping (after rank 0) will be the one that exchanges messages with rank 0 in the last stage. We already know that this will be rank $0 \oplus \frac{p}{2}$, where p denotes the total number of processes. Obviously, rank 0 will be our reference core because it is the only process that has already been mapped. Thus, in the first iteration, rank $0 \oplus \frac{p}{2}$ is mapped as close as possible to rank 0. Now, for the next process, we will have two options: 1) choosing with respect to rank 0, or 2) choosing with respect to the recently mapped process, i.e., rank $0 \oplus \frac{p}{2}$. Maintaining rank 0 as our reference core, we choose the new process based on the communications of rank 0 in the second-to-last stage of the recursive doubling algorithm. We already know that rank 0 exchanges a message with rank $0 \oplus \frac{p}{4}$ in that stage. Thus, rank $0 \oplus \frac{p}{4}$ will be the new process and is mapped as close as possible to rank 0. For the next new process, we will again have two options: 1) choosing with respect to rank 0, or 2) choosing with respect to the recently mapped processes, i.e., rank $0 \oplus \frac{p}{4}$ or rank $0 \oplus \frac{p}{2}$. This time, we choose the new process with respect to rank $0 \oplus \frac{p}{4}$ and update our reference core with rank $0 \oplus \frac{p}{4}$ accordingly. There are two reasons for this choice, as described below.

1. If we want to choose the next process with respect to rank 0 or rank $0 \oplus \frac{p}{2}$, then we have to choose based on the pairs of communications that fall in the third-to-last or second-to-last stages respectively. However, with rank $0 \oplus \frac{p}{4}$ we can choose the new process from the communication pairs that belong to the last stage and hence, will include larger messages.

2. The process that exchanges data with rank $0 \oplus \frac{p}{4}$ in the last stage, also communicates

with rank $0 \oplus \frac{p}{2}$ (already mapped) in the second-to-last stage. Thus, it communicates with a larger number of processes that have already been mapped.

With rank $0 \oplus \frac{p}{4}$ as the new reference core, we repeat the above procedure in the next two iterations so as to map two more new processes before updating the reference core again.

Algorithm 4.2 depicts our proposed mapping heuristic (RDMH) designed based on the above discussion. Lines 5-8 correspond to Step 4 of Alg. 4.1. Starting with $i = p/2$ in line 3, RDMH gives a higher priority to those ranks that communicate with the reference core in further stages of recursive doubling. With the loop in lines 5-7, new ranks are chosen from an earlier stage only if the ones corresponding to further stages have already been mapped. Lines 11-14 correspond to Step 7 in Algorithm 4.1. At the end of each iteration, the reference core is updated with the new process *only if* two processes have already been mapped with respect to the current reference core.

Algorithm 4.2: RDMH—Mapping heuristic for the recursive doubling communication pattern

Input : Number of processes p, Physical topology distance matrix D
Output: Mapping array M representing the new rank for each process

```
1  M[0] = 0 ; // Fix rank 0 on its current core
2  ref_rank = 0 ; // Choose 0 as the reference core
3  i = p / 2 ; // Starting from the last stage
4  while there exist more processes to map do
5      while ref_rank ⊕ i is already mapped do
6          i = i / 2;
7      end
8      new_rank = ref_rank ⊕ i;
9      target_core = find_closest_to(ref_rank, D) ; // Find the free core closest to
        the reference core
10     M[new_rank] = target_core ; // Map the new process onto the target core
11     if already mapped two processes with respect to ref_rank then
12         ref_rank = new_rank ; // Update reference core
13         i = p / 2 ; // Restarting from the last stage
14     end
15 end
```

It is worth mentioning that MVAPICH2 also exploits some sort of rank reordering for

recursive doubling. However, it is limited to a specific layout of processes only. There is no actual mapping heuristic or topology awareness in the sense that we discuss in here. The rank reordering in MVAPICH2 just changes a block initial layout of processes to a cyclic one. With our heuristic, we calculate an appropriate mapping by taking into account the initial mapping of processes as well as the physical topology of the system.

Ring

Our heuristic for ring (RMH) is quite straightforward. The main reason is because in the ring algorithm, each process communicates with one and only one other process that is fixed across all the stages of the algorithm. For ring, Alg. 4.3 chooses the processes for mapping in a simple increasing order of their ranks. Moreover, the reference core is updated with the new process at every iteration of the mapping algorithm. Having fixed rank 0, RMH maps rank 1 as close as possible to 0. Then rank 1 will be the reference core and the next process to be mapped will be rank 2. Rank 2 is mapped as close as possible to rank 1 and is set to be the new reference core. This procedure is repeated until there are no more processes to map.

Algorithm 4.3: RMH—Mapping heuristic for the ring communication pattern

Input : Number of processes p, Physical topology distance matrix D
Output: Mapping array M representing the new rank for each process

```
1 M[0] = 0 ; // Fix rank 0 on its current core
2 ref_rank = 0 ; // Choose 0 as the reference core
3 while there exist more processes to map do
4 |   new_rank = (ref_rank + 1) % p;
5 |   target_core = find_closest_to(ref_rank, D) ; // Find the free core closest to
  |    the reference core
6 |   M[new_rank] = target_core ; // Map the new process onto the target core
7 |   ref_rank = new_rank ; // Update the reference core
8 end
```

Binomial broadcast

We cover the binomial tree communication pattern because it can be used for the broadcast in the third phase of a hierarchical allgather. An advantageous characteristic of the binomial broadcast (gather) is that every rank receives (sends) data from (to) only one other rank. In addition, binomial broadcast has the advantage of having a fixed message size across all the stages of algorithm. Therefore, we seek to perform the mapping only in such a way that communicating ranks are placed as close to each other as possible. In other words, we are not concerned about the size of the communicated messages. Hence, it is enough to traverse the tree in some way, and map the nodes that we come across as close to each other as possible. For example, we could use a pure Depth-First Traversal (DFT) or Breadth-First Traversal (BFT) algorithm. However, we are interested to know if there are any particular traversing fashions that could heuristically represent better candidates.

A potential approach is to traverse the tree so as to visit the nodes with larger subtrees sooner than others. Each node is then mapped as close as possible to its corresponding parent node in the binomial tree. This way, a higher priority (in terms of being placed closer to a communicating peer) is given to those ranks on which a higher number of other ranks depend for receiving the message. This approach is essentially same as the one used by Subramoni et al. [113] for designing their network-aware broadcast algorithm. A second approach that we propose here is a variation of DFT which unlike the previous approach, visits the nodes with smaller subtrees first. With such a DFT, we will be giving higher priority to communications that happen at further stages of binomial broadcast. The rationale for this approach is that as we move toward the final stages of a binomial broadcast, the number of pair-wise communications increase. With p ranks, there is only one communication in the first stage, whereas in the last stage we will have $\frac{p}{2}$ communications. Therefore, communications happening at further stages are more likely to create contention. Hence, we want to map their corresponding ranks closer to each other.

Alg. 4.4 (BBMH) depicts the details of our mapping heuristic for binomial broadcast. Note that without loss of generality we assume rank 0 is the root of broadcast. The mapping is done in a recursive fashion by initially calling the recursive procedure `RecBinomialMap` with rank 0 as its argument. At the heart of the procedure lies the loop in lines 4-10. It determines how we choose a new rank for mapping with respect to each reference core r. So, it corresponds to Step 4 in Alg. 4.1. The conditions of the loop come from the underlying structure of the binomial tree and make sure that the new process represents a valid child of the reference core. The reference core is updated with the new rank by recursively calling `RecBinomialMap` in line 8. Thus, it represents Step 7 in Alg. 4.1.

Algorithm 4.4: BBMH—Mapping heuristic for the binomial broadcast communication pattern

```
   Input  : Number of processes p, Physical topology distance matrix D
   Output: Mapping array M representing the new rank for each process
 1 M[0] = 0 ; // Fix rank 0 on its current core
 2 Rec_binomial_map(0) ; // Calling the recursive mapping function with rank
    0
 1 Procedure RecBinomialMap(rank r)
 2     ref_rank = r ; // Choose r as the reference core
 3     i = 1;
 4     while (ref_rank & i = 0) and (i ≤ p / 2) do
 5         new_rank = ref_rank + i;
 6         target_core = find_closest_to(ref_rank, D);
 7         M[new_rank] = target_core;
 8         Rec_binomial_map(new_rank) ; // Calling the algorithm recursively
            for the new rank
 9         i = i * 2 ; // Move on to the next child
10     end
```

It is worth noting that for medium and large messages, broadcast is commonly implemented by a scatter-allgather algorithm [116]. However, we do not propose a separate mapping heuristic for it because RDMH and RMH already cover the allgather phase, and the scatter phase can be covered by the proposed heuristic for gather in the next section.

Binomial gather

Gather constitutes the first phase of the hierarchical allgather, and the binomial tree is again a major pattern used for it. Although the communication pattern is binomial, the size of messages are not fixed in a binomial gather. As we get closer to the root of the tree, the size of exchanged messages is increased. This time, we want to pick the heaviest edge of the tree each time, and map its unmapped endpoint as close to the mapped one as possible. There are two reasons for this. First, this way we will be choosing a rank which *does* communicate with some of those that have already been mapped. Second, it represents a communication that uses larger messages. This is similar to the rationale used by Hoefler and Snir [45] in their greedy mapping heuristic for the generic case of topology-aware mapping. However, our proposed heuristic is different in the sense that we extract the communication pattern in a closed-form fashion without building any process topology graph. In other words, we will systematically find the heaviest edges and their corresponding reference cores at each step.

Alg. 4.5 illustrates the details of our mapping heuristic (BGMH) for the binomial gather communication pattern. Note that without loss of generality, we assume rank 0 is the root of gather. Line 12 along with the loop in line 5 determine the way that the new process is chosen with respect to each reference core. Therefore, they correspond to Step 4 in Alg. 4.1. The reference core is updated in each iteration of the loop in line 6 and is representative of Step 7 in Alg. 4.1. For each value of i, BGMH iterates over all the potential reference cores in $\mathcal{V}$ and maps a new process as close as possible to each. In addition, every newly mapped rank is added to the set of potential reference cores in line 10.

Complexity All the proposed algorithms have the same complexity. They perform one iteration per process and in each iteration, the main cost is for finding the closest core to the reference core. This can be done in $\mathcal{O}(p)$ with a linear search, where p denotes the total number of processes. Thus, the total complexity will be $\mathcal{O}(p^2)$.

Algorithm 4.5: BGMH—Mapping heuristic for the binomial gather communication pattern

Input : Number of processes p, Physical topology distance matrix D
Output: Mapping array M representing the new rank for each process

```
1  M[0] = 0 ; // Fix rank 0 on its current core
2  Initialize V ; // The set of potential reference cores
3  V ← 0 ; // Start with rank 0
4  i = p / 2;
5  while i > 0 do
6      for ref_rank ∈ V and ref_rank + i < p do
7          new_rank = ref_rank + i;
8          target_core = find_closest_to(ref_rank, D);
9          M[new_rank] = target_core;
10         V ← new_rank ; // Add the new rank to the list of potential
               reference cores
11     end
12     i = i / 2;
13 end
```

4.2.3 Preserving the Correct Order of the Output Buffer

At the end of `MPI_Allgather`, each process will have an output vector that holds the individual messages corresponding to every other process. The elements of this vector should appear in a correct order, i.e., in the order of the process ranks to which they initially belonged. Each underlying allgather algorithm is designed so as to preserve such a correct order of the output buffer. However, rank reordering can disrupt the desired order because as soon as we change the rank of a given process from i to j, it will act as the process with rank j while actually having the input vector corresponding to rank i. In the following, we explain two different approaches for addressing this issue. Sack and Gropp [106] also use similar techniques in their distance-halving recursive doubling algorithm.

Extra initial communications

In the first approach, we use extra send/receive communications to move the data between the processes with respect to the reordered ranks. This is done before the allgather algorithm

issues any communications. For instance, if the process with rank 0 is going to be reordered to rank 1, we will have rank 1 send its input vector to rank 0. In addition, rank 0 will send its input vector to the process that is going to be reordered to 0. This way, the input vector of all the processes will be in correspondence to their new ranks and hence, the output vector will be in correct order.

Memory shuffling at the end

In the second approach, we do not exchange the input buffers and let the allgather operation proceed as usual. However, we shuffle the output buffer elements at the end based on how the process ranks have been reordered. For instance, if rank 0 has been changed to rank 1, then we know that the element at index 1 in the output buffer actually belongs to rank 0. Thus, it should be moved to the head of the output buffer.

It should be noted that in practice we use the above mechanisms with recursive doubling and binomial gather only. The output vector order does not apply to a binomial broadcast as the output buffer is not a vector any more. For the ring algorithm, we resolve the issue from within the algorithm itself by storing each incoming message at the correct offset of the output vector. This can easily be done with no additional overheads for the ring because every process receives only one individual message at each stage of the algorithm. We can easily figure out the correct offset in the output vector based on the mapping array. On the contrary, recursive doubling and binomial gather communicate aggregate messages that will require costly packing/unpacking from/to non-contiguous memory regions. For recursive doubling, for example, the processes receive aggregate messages that encapsulate multiple individual messages, each of which belong to a particular rank. For instance, in the second stage, rank 0 receives an aggregate message from rank 2 that contains the contributions from both rank 2 and rank 3. Upon reception, rank 0 simply concatenates such an aggregate message to the elements it already has. Thus, it is required that the individual elements in each aggregate message belong to a particular consecutive set of ranks. Reordering the

ranks will disrupt this requirement because the elements in each aggregate message can correspond to an arbitrary set of ranks.

4.3 Experimental Results and Analysis

For our experiments that are discussed in this chapter, we use the same InfiniBand cluster (GPC) explained in Section 3.4 of Chapter 3. Moreover, we use MVAPICH2-2.0, Scotch-6.0.0, and hwloc-1.8.1.

4.3.1 Microbenchmark Results

We use the OSU Microbenchmarks [93] to measure the latency of `MPI_Allgather` with and without rank reordering. The results report the percentage of performance improvement over the default algorithms used in MVAPICH2. We also compare the performance of our proposed mapping heuristics against Scotch [95] as an alternative mapping algorithm to our proposed heuristics. With 4096 processes, we only report the results for a maximum message size of 256KB due to memory restrictions on each node. In all figures, 'initComm' and 'endShfl' respectively refer to extra initial communications and memory shuffling at the end for preserving the correct order of the output buffer. Moreover, 'Hrstc' represents the results for our proposed heuristics.

We perform the experiments for four different initial mappings of processes: *block-bunch*, *block-scatter*, *cyclic-bunch*, and *cyclic-scatter*. We choose these four so as to characterize the impacts of our heuristics under four different (and well known) initial mappings. Note that the gain in performance by our heuristics (as well as any other rank-reordering scheme) depends on how far the initial mapping is from the ideal mapping for each collective communication pattern. In the block mapping, adjacent ranks are mapped onto the same node as much as possible before moving to any other node. Similarly, a bunch mapping binds the adjacent ranks to the cores of the same socket inside each node. On the contrary, cyclic

(a) block-bunch

(b) block-scatter

(c) cyclic-bunch

(d) cyclic-scatter

Figure 4.1: Microbenchmark performance improvements for the non-hierarchical topology-aware allgather with four different initial mappings—4096 processes.

mapping distributes adjacent ranks across the nodes in a round-robin fashion. The scatter mapping uses a similar round-robin scheme to scatter ranks across all the sockets within each node.

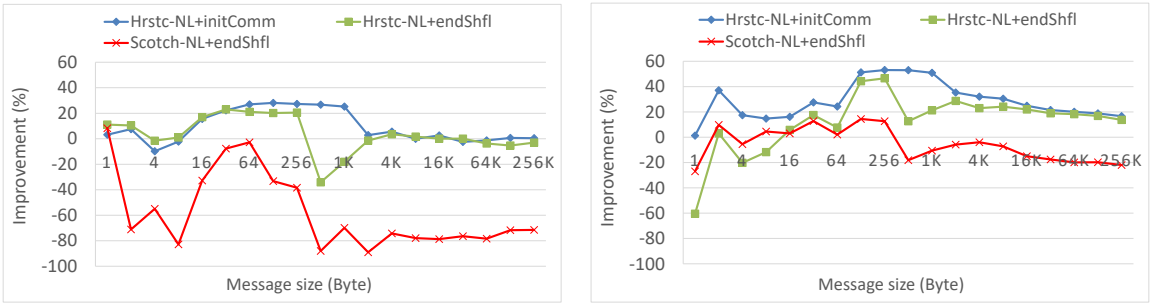
Non-hierarchical allgather

Fig 4.1 shows the results for the non-hierarchical allgather approach. As shown in Fig. 4.1(a), we can achieve up to about 67% improvement for messages smaller than 1KB with our proposed mapping heuristics. The improvements correspond to RDMH as MVAPICH2 uses recursive doubling in this range of message sizes. We also see that Scotch performs quite poorly in this interval. Moreover, for RDMH and up to 1KB message size, the improvement is increasing with message size. This is because larger messages are affected more by the

adverse effects of congestion. Another observation is the better performance achieved by extra initial communications compared to memory shuffling.

Fig. 4.1(a) also shows that for messages larger than 1KB we cannot achieve any performance improvement. This is because MVAPICH2 uses the ring algorithm in this range of message sizes and the initial block-bunch mapping is already the best match. However, unlike Scotch, our heuristic (RMH) does not cause any performance degradation. Also, Fig. 4.1(b) shows that with a block-scatter initial mapping, RHM can provide about 50% improvement for message sizes above 1KB. This is because the intra-node scatter mapping is not a good match for the ring algorithm. Again, we see that Scotch performs poorly. We can also see that for messages above 1KB, initComm and endShfl result in the same performance as we do not actually use/need any of these mechanisms for the ring algorithm (see Section 4.2.3).

Fig. 4.1(c) and 4.1(d) show that with an initial cyclic-bunch or cyclic-scatter mapping, our heuristic can provide up to 50% improvement for message sizes below 1KB. In addition, we can achieve 78% improvement for messages larger than 1KB. We also see that memory shuffling overheads can become quite costly for 512B and 1KB message sizes. Another observation is the higher improvement for messages above 1KB compared to Fig. 4.1(a) and 4.1(b). This is mainly because an initial cyclic/scatter mapping along with the underlying ring algorithm result in a higher congestion across the network/QPI links. At the same time, we see a relatively lower improvement for message sizes below 1KB in Fig. 4.1(d) compared to Fig. 4.1(a) and 4.1(b). This is because an initial cyclic (scatter) mapping is better than block (bunch) for the recursive doubling algorithm. Thus, we see that a poor initial mapping for one algorithm can be relatively better for another. This is an encouraging observation for employing runtime rank reordering for collective communications.



(a) block-bunch, non-linear

(b) block-scatter, non-linear

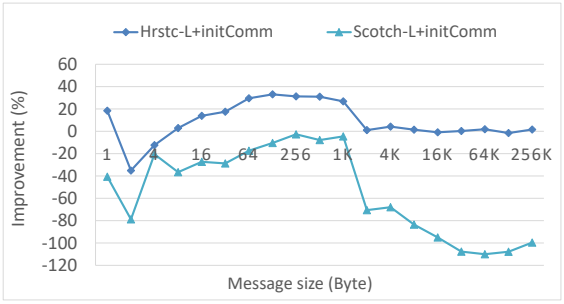
Figure 4.2: Microbenchmark performance improvements for the hierarchical topology-aware allgather with a non-linear intra-node broadcast/gather and two different initial mappings—4096 processes.

Hierarchical allgather

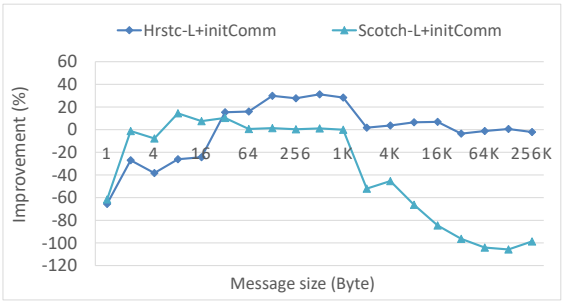
With a hierarchical allgather, we only consider the two block-bunch and block-scatter initial mappings. There are two reasons for this approach. First, hierarchical allgather is not supported with a cyclic mapping in MVAPICH. Second, the cyclic mapping results in a similar layout of the processes to the block mapping at both of the intra- and inter-node communicators. In our result charts, we use the 'L' and 'NL' suffixes to respectively refer to the linear (direct) and non-linear (indirect algorithmic) intra-node broadcast/gather phases of the hierarchical allgather algorithm.

Fig. 4.2 and Fig. 4.3 show that the improvements are generally lower for the hierarchical algorithms. This is because a hierarchical approach provides a level of topology awareness by restricting the inter-node communications to node-leaders only. Moreover, with a hierarchical approach, rank reordering is used at a smaller scale as it is applied to node-leaders and local processes separately. We can also see that Scotch performs relatively better with the hierarchical approach compared to the non-hierarchical one.

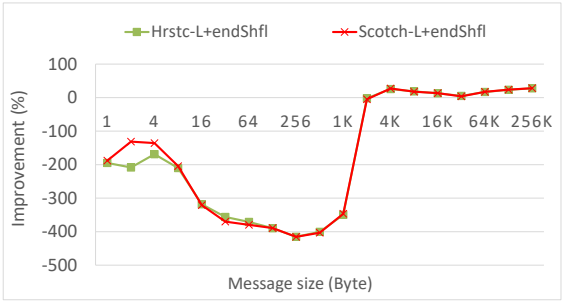
Fig. 4.2 show the results for the hierarchical allgather with a non-linear intra-node broadcast/gather. In particular, Fig. 4.2(b) shows that for messages above 1KB, we can attain up to 30% improvement over a block-scatter mapping. This improvement comes



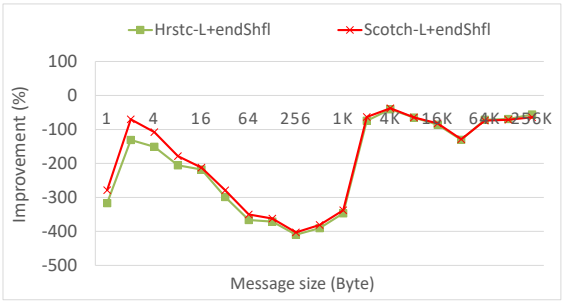
(a) block-bunch, linear, initial communications



(b) block-scatter, linear, initial communications



(c) block-bunch, linear, memory shuffling at the end



(d) block-scatter, linear, memory shuffling at the end

Figure 4.3: Microbenchmark performance improvements for the hierarchical topology-aware allgather with a linear intra-node broadcast/gather and two different initial mappings—4096 processes.

from the gather phase within each node. The BGMH algorithm reorders the ranks so that large-message communications of the intra-node binomial gather fall within a single socket, and thus benefit from a higher bandwidth. However, we see a decreasing improvement with increase in the message size which is due to the overheads induced by the extra communications or memory shuffling. We believe the intra-node impacts of our algorithms can be better studied in a system with a higher number of cores per node.

Fig. 4.3 show the results for the hierarchical allgather with a linear intra-node broadcast/gather. Note that in this case, we cannot have any rank reordering at the intra-node level as there is no particular pattern for which to optimize the mapping; all the processes directly communicate with the root process. Thus, there is less room for improvements. However, as shown by Fig. 4.3(a), we can still improve the performance by more than 30%

for message sizes below 1KB (RDMH). As expected, we cannot achieve any improvement for the messages larger than 1KB because the initial block mapping is already a good match for the underlying ring algorithm. Fig. 4.3(c) and 4.3(d) show that for messages below 1KB, the performance is quite poor for the endShfl version of both Scotch and our heuristics. The main reason is because in this case, the overheads of memory shuffling are too high that nullify the potential benefits of a better mapping. Note that memory shuffling in this case is done over combined (larger) messages that are gathered from all the ranks within a node. Another observation is the improvement for the messages larger than 1KB. We do not know where exactly this improvement comes from and need to investigate it further.

4.3.2 Application Results

In this section, we evaluate the efficiency of our topology-aware rank reordering for Nbody [107] as a real application. We measure the execution time of Nbody with and without our rank-reordered versions of `MPI_Allgather`. Our profiling results with 1024 processes show that Nbody makes more than 1,000 calls to `MPI_Allgather` which makes it a potentially good candidate for our experimental analysis with a real application. We report the results for 1,024 processes as we could not scale the application to a larger number of ranks due to memory restrictions. Moreover, we only use extra initial communications for preserving the correct order of the output buffer as it was shown to outperform memory shuffling in the microbenchmark section.

Fig. 4.4 shows the execution time of Nbody with different mapping approaches and the non-hierarchical `MPI_Allgather` used in MVAPICH2. As shown by the figure, for the block-bunch mapping, our heuristics result in the same execution time as the default mapping. Again, this is because the initial mapping is already a good match for the underlying communication patterns in this case. Scotch, on the other hand causes an almost 2-fold increase in the execution time. In fact, it can be seen that in all cases shown in Fig. 4.4, Scotch exacerbates the execution time of the application. With a block-scatter mapping, we

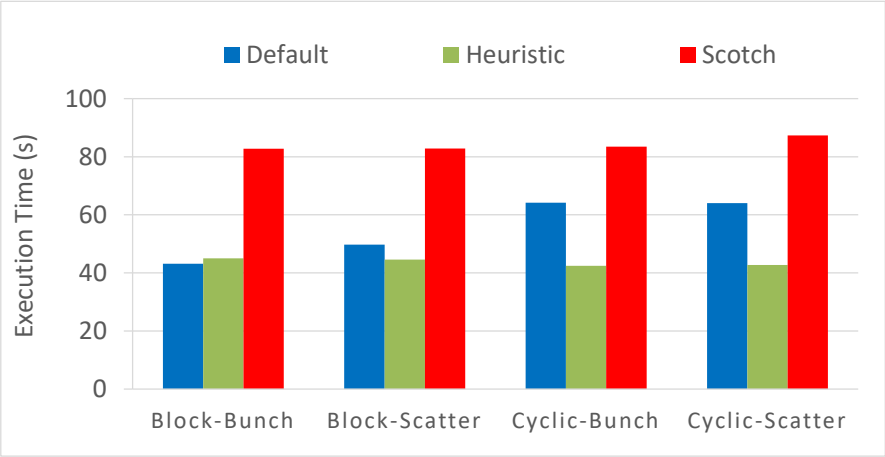
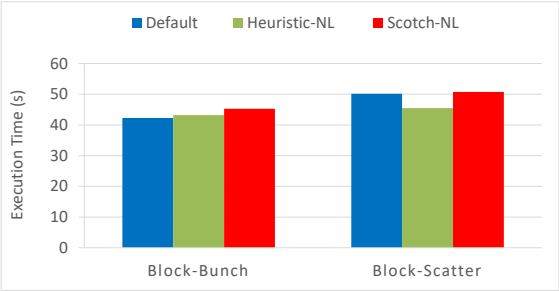
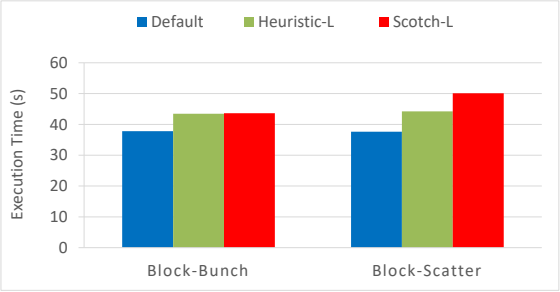


Figure 4.4: Execution time of the Nbody application with the non-hierarchical allgather and different mapping approaches—1024 processes.



(a) Non-linear intra-node communications



(b) Linear intra-node communications

Figure 4.5: Execution time of the Nbody application with the hierarchical allgather and different mapping approaches—1024 processes.

can achieve about 10% reduction in the execution time with our proposed heuristics. The highest improvement is seen with the cyclic-bunch and cyclic-scatter initial mappings where we can achieve about 30% reduction in execution time with our proposed rank reordering algorithms.

Fig. 4.5 shows the results with respect to the hierarchical allgather approach. In particular, Fig. 4.5(a) shows that with a non-linear intra-node pattern, we cannot see any improvement over the block-bunch mapping. For block-scatter however, we could achieve

about 10% improvement with our heuristics. In addition, Fig. 4.5(b) shows that rank reordering could not improve execution time of the application when a linear pattern is used for the intra-node broadcast/gather. This is an expected behavior as the combination of a block mapping at the inter-node layer and the linear intra-node patterns highly restrict the opportunity to benefit from rank reordering. We even see a slight increase in the execution time which is not expected and requires further investigations to find out the root cause of it.

4.3.3 Overhead Analysis

In this section, we evaluate the total overhead of topology-aware rank reordering with our mapping heuristics, and compare it to Scotch. The two main components of the total overhead are the time required for finding the physical distances, and the time spent by the mapping algorithm. Finding the physical distances is a one-time overhead, whereas we will have the mapping overhead once for each communication pattern. Our measurements show that our heuristics have almost the same amount of overhead. Therefore, we avoid reporting the overhead results for each one of them separately.

Fig. 4.6 shows the overhead of finding the physical distances for three different number of processes. The trend is mainly dominated by the overhead of gathering distance information from all ranks at rank 0. We can see a linear scaling as the number of processes increase. With 4096 ranks, it takes about 3.3 seconds to extract all distances. This overhead is similarly applied to both Scotch and our proposed heuristics. However, it is a one-time overhead which is *not* repeated every time that we perform rank reordering for a new pattern.

Fig. 4.7 shows the time spent by the mapping algorithms after the distances have been extracted. As shown by Fig. 4.7(a), our proposed heuristics impose a significantly lower overhead compared to Scotch which is shown in Fig. 4.7(b). We can also see a better scaling trend for our heuristics compared to Scotch. The lower overhead of our heuristics is rooted

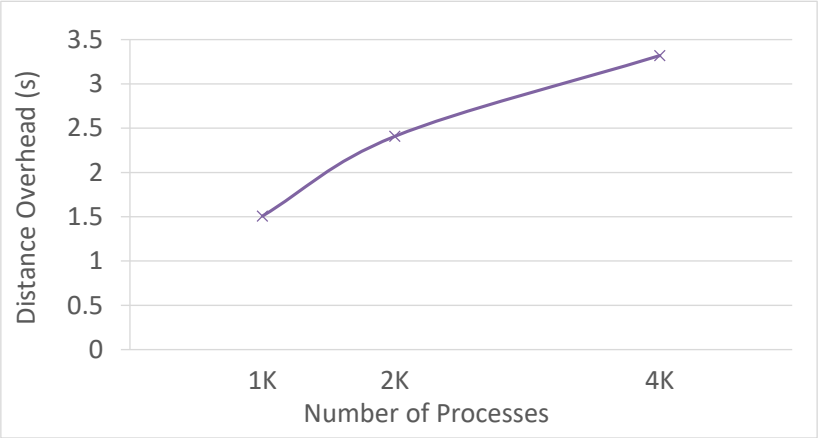
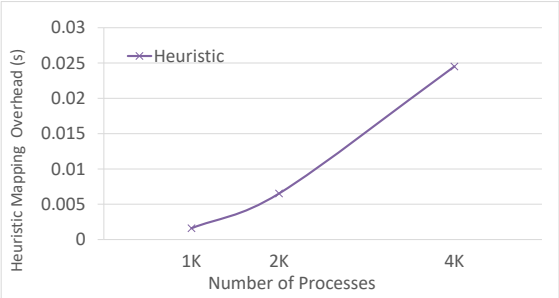
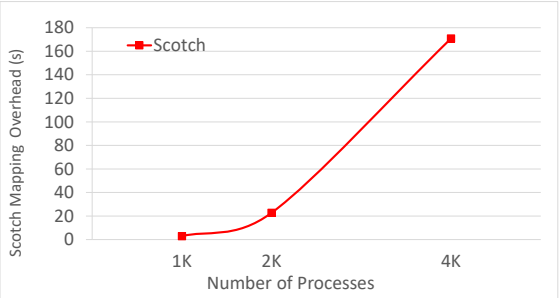


Figure 4.6: Overhead of extracting the physical distances for different number of cores.



(a) Our heuristics



(b) Scotch

Figure 4.7: The overhead of the mapping algorithms with different number of processes.

in the simpler design and the fact that unlike Scotch, we do not need to build a process topology graph for any of our mapping heuristics. With 4096 processes, the overhead of our heuristics is only about 25 ms, whereas the same number for Scotch is more than 160 seconds. In addition, with 1024 ranks, the total overhead including the physical distance extraction is less than 1.6 seconds which represents less than 4% of the total execution time of Nbody with the same number of processes.

4.4 Related Work

Various attempts have been made to bring topology awareness to collective communication design. Almási et al. [2] propose machine-optimized versions of MPICH2 collective communications for IBM Blue Gene/L systems. They observe that the MPICH2 collective algorithms tend to map poorly onto the 3D torus network used in IBM Blue Gene/L systems, ending up to an inefficient usage of the limited bisection bandwidth of the network. Chan et al. [19] propose algorithms for multiport broadcast and allgather in order to take advantage of simultaneous communications by exploiting multiple links connected to each node in multidimensional grids. Faraj et al. [27] presented optimized collective algorithms for the IBM Blue Gene/P systems that heavily exploit the hardware features such as the high performance DMA engine which frees the processing cores from packet management and is capable of keeping all six links connected to each node busy. Kumar et al. [66] optimize MPI collective communications for the IBM Blue Gene/Q systems by exploiting various specific features that are provided by such systems.

Bhatelé et al. [10] study the impacts of mapping for collective communications over sub-communicators. Accordingly, they propose Rubik which provides a simple tool for systematic generation of a wide variety of mappings. Zahavi et al. [125] study the interactions among collective communication patterns, routing algorithm, and congestion in fat-tree networks. They inspect the communication pattern of common collective communications and present an algorithm for generating routing tables that can avoid congestion for global collective communications in fat-trees with full bisection bandwidth. Prisacari et al. [99] show that unlike the common belief that the alltoall collective communication pattern requires a full bisection bandwidth, alltoall exchanges can be done free of congestion with only half bisection bandwidth. They propose an algorithm for extracting an optimized exchange pattern ensuring that in each stage of the algorithm, the number of messages traversing each node does not exceed the aggregate bandwidth available at that node.

Sack and Gropp [106] present non-minimal versions of the recursive-doubling and bucket algorithms to decrease the amount of congestion, and exploit multiport communications within fat-tree and 3D torus networks. They propose a recursive-doubling, distance-halving algorithm which is essentially a modification of the original recursive-doubling algorithm used for the `MPI_Allgather` operation. The processes first start communicating with their farthest peers, and the distance between the communicating pairs is halved in every stage of the algorithm. Unlike our work, Sack and Gropp [106] still ignore how the processes have been mapped onto the system. Moreover, they only consider the network topology and assume there is only one process on each node.

Kandalla et al. [59] design topology-aware collective communication algorithms for InfiniBand clusters by using leader processes that correspond to different hierarchies within the system topology. Subramoni et al. [113] use rank reordering to design a network-aware broadcast algorithm. However, they only consider the network topology for improving the inter-node component of a hierarchical broadcast algorithm, and hence do not take into account the intra-node topology of the target system. In our design, we consider both the hierarchical and non-hierarchical allgather approaches, and in each case take into account the system topology at both the intra- and inter-node layers. In addition, we propose fine-tuned mapping heuristics for four of the major communication patterns that are commonly used in `MPI_Allgather`, whereas Subramoni et al. [113] use the classic DFT/BFT algorithms for the binomial tree communication pattern only. In another work, Subramoni et al. [112] propose network topology-aware communication schedules to reduce congestion for alltoall FFT operations in InfiniBand clusters.

Li et al. [70] propose three shared-memory NUMA-aware algorithms for the intra-node component of `MPI_Allreduce`. The algorithms are targeted for a thread-based implementation of MPI ranks [33] where all MPI ranks that reside on a node run within the same address space and hence require only one memory copy to communicate with each other. The proposed algorithms attempt to minimize the inter-socket data transfers within each

node. In a similar study, Ma et al. [78] focus on optimization of the intra-node component of collective communications based on different levels of hierarchies within each nodes. they propose a framework for measuring the distance between each and every pair of processes based on the memory and physical hierarchies of the target system. Two algorithms are proposed for building broadcast trees and allgather logical rings based on the physical distances among processes. The algorithms have also been used in another work [77] for the optimization of hierarchical collectives. However, the main focus in that work is to increase the overlap between the intra- and inter-node phases of hierarchical collectives by taking advantage of single-copy mechanisms in the OS kernel.

Faraj et al. [28] propose an algorithm for building contention-free allgather rings among the nodes in switched clusters. The ring is built by a depth-first traversal of a spanning tree within the network topology. However, the intra-node topology is considered to be flat. Kumar et al. [67] design tree-based collective communication algorithms with a focus on fat-tree topologies. The proposed algorithms use multiple k-ary trees to transmit data across multiple network links so as to achieve higher throughput.

4.5 Summary

Collective communications constitute a major portion of the total communications that are used in many parallel applications. Various algorithms have been designed to implement collective communications by means of a set of point-to-point communications with a certain pattern. These algorithms have been traditionally designed based on the MPI ranks of the processes only. We discussed how the specific mapping of the processes along with the physical topology of the system affects the potential performance and benefits of a collective communication algorithm. Accordingly, we proposed four fine-tuned mapping heuristics for various communication patterns used in an MPI allgather operation. The heuristics provide a fast mechanism for finding a better match between the specific communication

pattern of the collective algorithm and the physical topology of the target system. Our experimental results showed that using our heuristics we can improve the overall latency of MPI allgather. It was also shown that the proposed heuristics deliver a higher performance and a significantly lower overhead compared to a general mapping library such as Scotch.

Chapter 5

Topology-Aware Communications in Heterogeneous GPU Clusters

Over the past decade, the HPC landscape has changed significantly, particularly due to the emergence of accelerators. In particular, Graphics Processing Unit (GPU) accelerators have established themselves in modern heterogeneous HPC clusters by offering high performance and energy efficiency. At the time of writing this document, 12% of the top-500 supercomputers in the world are equipped with GPUs [117]. Heterogeneous GPU clusters have become the platform of choice for various HPC applications [68, 34]. To further increase the computational power and address larger problems, such systems provide various processing elements (PEs) including multicore general-purpose processors as well as multiple GPU devices. Hereinafter, we use the term Central Processing Unit (CPU) to refer to the general-purpose processors in contrast to GPUs. However, we note that such general-purpose processors are no longer central as there exist many of them in a parallel system.

In the previous chapters, we discussed how topology awareness can lead to a more efficient utilization of communication channels in traditional CPU-only systems. In this chapter, we show that similar issues exist for GPU-to-GPU communications as well, and discuss how topology awareness can be deployed in heterogeneous GPU clusters. In particular, we study the joint problem of process-to-core(node) mapping and GPU-to-process

assignment in GPU clusters that consist of multiple multi-GPU nodes [84, 31, 32]. To the best of our knowledge, our work in this chapter is the first work that considers topology awareness for GPU clusters.

5.1 GPU-Aware MPI

In a GPU cluster, processes offload certain parts of their computations to the GPUs so as to accelerate their progress. Therefore, they will require support from the MPI library to communicate data which might reside in the GPU memory. To perform such communications efficiently, MPI must be tuned and become GPU-aware. In MPI applications, both the intra- and inter-node GPU communications are considered to be a major performance bottleneck. In fact, the overhead of GPU communications in applications with high data interdependency may even nullify the potential benefits of GPU offloading. In this regard, various attempts have been made to improve the performance of GPU communications by introducing GPU awareness into MPI point-to-point and collective operations [30, 57, 97].

Accordingly, support for GPU communications has been added to well known MPI implementations such as MVAPICH2 and Open MPI. Such support may follow a general approach which involves staging GPU data into the host buffer and leveraging the traditional CPU-based MPI routines. Such support may also involve further tuning by pipelining transfers and using specialized algorithms. In addition, some hardware-related features such as CUDA IPC and GPUDirect RDMA can provide direct GPU-to-GPU communications.

5.2 Motivation

Each compute node in a GPU cluster may consist of multiple GPU devices. Such nodes are commonly known as multi-GPU nodes. MPI processes running on a multi-GPU node can potentially select any of the GPU devices available on that node. However, depending on the selected GPU, the inter-process GPU-to-GPU communications may have to traverse

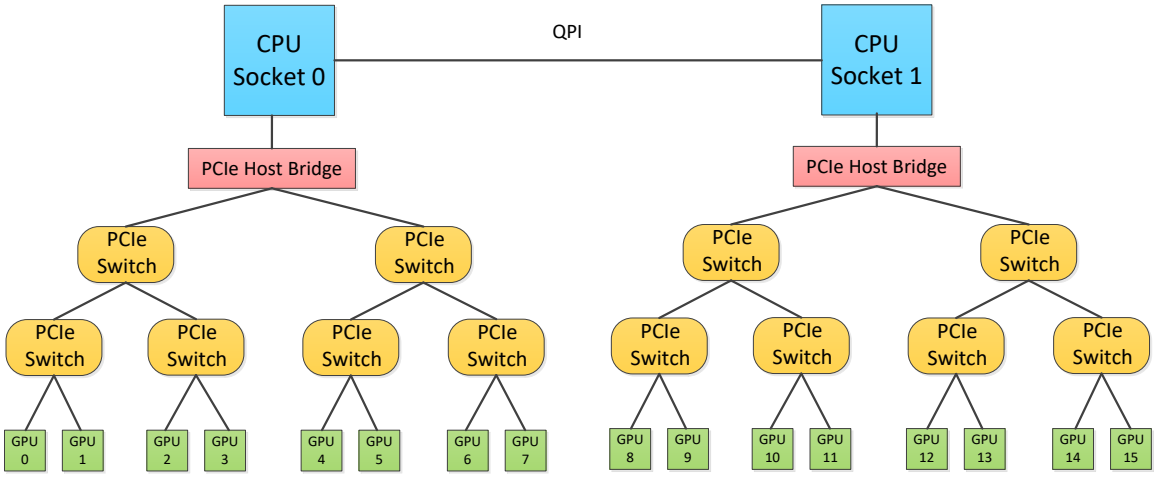


Figure 5.1: The topology of a multi-GPU node with 16 GPUs connected to each other by a multi-level PCIe interconnect.

different paths. The paths may consist of different communication channels with different performance characteristics.

Fig. 5.1 shows the PCIe topology of a multi-GPU node consisting of 16 GPUs and 2 CPU sockets. We can see multiple levels and types of communication channels among different GPUs. For instance, communication between GPU0 and GPU1 only passes though one PCIe switch, whereas communication between GPU0 and GPU8 should traverse multiple PCIe switches, as well as the QPI link. Fig. 5.2 shows the actual bandwidth of these channels at different levels of the topology. As shown by the figure, for message sizes larger than 8KB, the topology level has a considerable impact on the communication bandwidth among the GPUs. We can see better performance (higher bandwidth) for the lower levels of the topology. It is also shown that as the message size increases, the performance gap between different topology levels becomes wider. These results motivate us to consider the physical topology of GPUs when it comes to improving communication performance through topology awareness in GPU clusters.

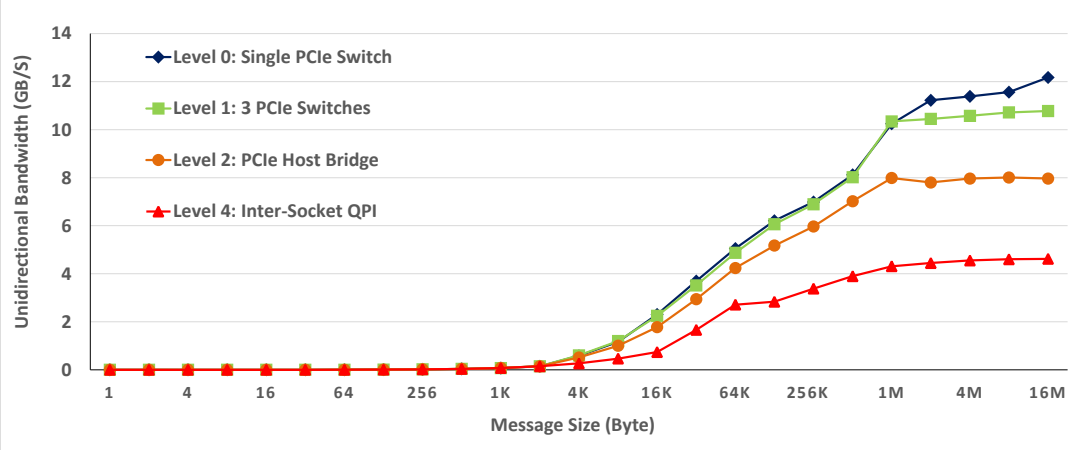


Figure 5.2: GPU communication bandwidth at different levels of the fabric topology that interconnects GPUs within a multi-GPU node.

5.3 MAGC: Unified Mapping Approach for GPU Clusters

GPU topology information can be used to find a more efficient assignment of GPUs to the processes of an application. In particular, Faraji et al. [31, 32] model the GPU assignment problem as a topology-aware mapping problem to design a topology-aware GPU selection scheme for multi-GPU nodes. They show the effectiveness of topology-aware GPU selection for improving GPU-to-GPU communication performance in a multi-GPU node. On the other hand, in Chapter 3, we studied the importance of topology-aware process mapping for CPU-to-CPU communications. Thus, in this section, we consider these two problems in a combined manner for GPU clusters, where we will delve into the joint problem of process-to-core(node) mapping and GPU-to-process assignment.

More specifically, we seek to answer the following question: Given an application that uses both CPU and GPU processing elements, how should such PEs be assigned to application processes so that the ultimate communication performance is optimized? In essence, the question is which core/node should host each process, and which GPU should be assigned to it. This is a challenging problem because optimizing for one communication

pattern (e.g., CPU-to-CPU) might conflict with optimization for another (i.e., GPU-to-GPU). Moreover, the CPUs and GPUs can have different communication patterns, as well as different intra-node physical topologies.

We first elaborate on how process-to-CPU mapping and GPU-to-process assignment become related to each other (and impact each other) in GPU clusters, and then discuss how a unified approach can be used to address both problems. In particular, we propose MAGC, a Mapping Approach for GPU Clusters. MAGC attempts to improve the total communication performance by considering both CPU-to-CPU and GPU-to-GPU communications of an application[1], as well as the CPU and GPU physical topologies of the system.

5.3.1 The Joint Problem of Process Mapping and GPU Assignment

In heterogeneous GPU clusters, topology-aware mapping is a joint problem of process-to-core(node) mapping and GPU-to-process assignment. Fig. 5.3 illustrates a schematic diagram of this problem where we have the CPU and GPU communication patterns, as well as the CPU and GPU topologies as input, and would like to find an efficient process-to-core mapping and GPU-to-process assignment.

One way to address the joint problem (shown in Fig. 5.3) is to independently figure out process-to-core mappings and GPU-to-process assignments in two separate steps. For instance, one could first bind processes to CPU cores based on the CPU communication pattern and CPU physical topology of the system, and then assign GPUs to processes with respect to the GPU communication pattern and the physical topology of the GPUs. At each step, a mapping algorithm/tool (e.g., Scotch) can be used to find an optimized assignment. As discussed in Section 6.5, the first step of such an approach (i.e., process-to-core mapping) has extensively been studied before. Moreover, in other work [31, 32], we studied topology-aware GPU assignment within a single node. However, in the following, we will discuss the

[1]We interchangeably use the terms CPU-to-CPU and CPU communications to refer to the communications among the CPU cores of a cluster. Same thing applies to GPU-to-GPU and GPU communications for GPU devices.

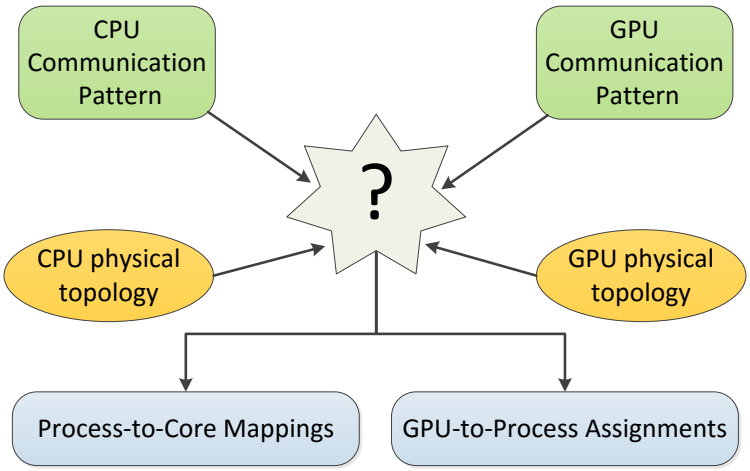


Figure 5.3: Schematic diagram of the joint process-to-core mapping and GPU-to-process assignment problem.

shortcomings of such a two-step approach for addressing the problem shown in Fig. 5.3, and will motivate the need for a unified topology-aware mapping framework.

Deriving process-to-core mapping and GPU assignment independently could easily lead to a poor result because a particular process-to-core mapping could limit the choices for efficient assignment of GPUs. In fact, the two steps could lead to conflicting process placement strategies. This is specially true when we consider GPU assignment across multiple nodes. For instance, consider two processes with zero (or low) CPU communications between them. Topology-aware mapping strategies would typically map such two processes on nodes that are far from each other. However, the same two processes could have a high amount of GPU communications with each other which requires their assignment to two GPUs that are physically close to each other. This could potentially lead to a situation where a process is mapped onto a CPU on one node, and assigned a GPU residing on another remote (potentially distant) node. This is undesirable for multiple reasons. First, the only visible GPU devices to a process are those that fall within the same node onto which the process has been mapped. Thus, accessing a remote GPU would require a middleware

software framework (such as rCUDA [25]) to allow remote GPU virtualization. Second, all data movements between the process and its remote GPU would have to pass across the network. Third, some GPU-aware MPI designs [108] leverage CPU-assisted mechanisms, in which the GPU-to-GPU communication is pipelined by staging the data portions in the CPU memory. Such communications would be adversely affected with remote GPU assignments.

A potential workaround for remote GPU assignment problem is to move (remap) the processes accordingly. That is, process mappings will be revised with respect to GPU assignments so that no process is assigned a remote GPU; each process will be moved to the same node as its assigned GPU. However, if we simply move the processes with respect to GPU assignments, it might increase CPU communication costs by placing highly communicating processes far from each other. Thus, it is required to consider CPU-to-CPU and GPU-to-GPU communications jointly to derive process mappings and GPU assignments in a unified framework.

5.3.2 Unified Framework

We propose the unified framework shown in Fig. 5.4 to tackle the topology-aware process mapping and GPU assignment problem shown in Fig. 5.3. The framework consists of three main phases that address the problem at different physical topology layers. The first phase mainly focuses on the inter-node network layer, whereas the second and third phases deal with the problem at the intra-node layer.

First phase

In order to avoid the problem of remote GPU assignment mentioned in Section 5.3.1, we limit GPU assignments to the boundaries of the node hosting each process. Thus, we first figure out the node on which each process should be mapped. However, we do this with respect to a combined communication pattern as the input to the mapping algorithm in the

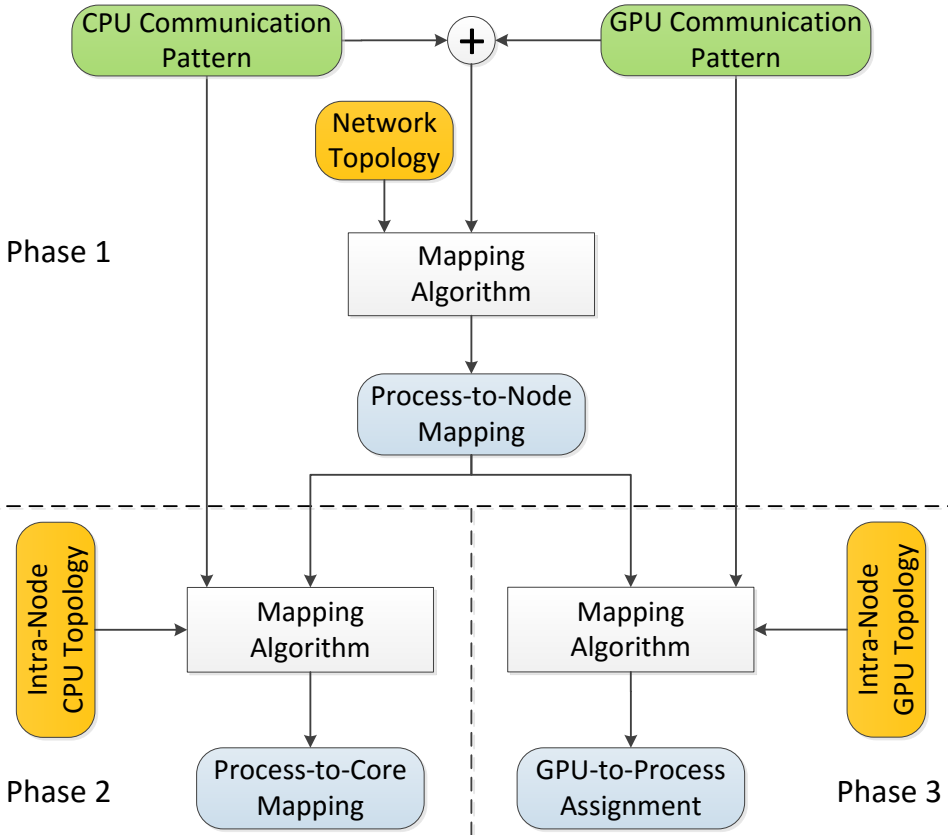


Figure 5.4: Schematic diagram of MAGC; a three-phase approach for process mapping and GPU assignment in heterogeneous GPU clusters.

first phase. The combined pattern is built by adding the CPU and GPU communication patterns together. For the physical topology, we only need to consider the network topology at this phase because it will be the representative of the physical topology for both GPUs and CPUs at the inter-node layer of the system. In fact, this is the feature that enables us to use the combined CPU and GPU communication pattern in the first phase to conduct the mapping with a joint CPU-GPU awareness.

The output of the first phase will be a process-to-node mapping. This mapping will also imply an implicit GPU assignment as it confines each process with the set of GPUs that reside on its hosting node. With an appropriate mapping algorithm, the mapping

resulting from the first phase will have the following desired characteristics. The set of processes mapped on the same node will have a relatively higher combined CPU and GPU communications. This will potentially improve performance as such communications will benefit from stronger intra-node (versus inter-node network) communication channels. It will also decrease the potential need to have any remote GPU assignments in future steps. Furthermore, by using a network-aware mapping algorithm, we will gain similar benefits across the network as well; higher-communicating processes will fall into the nodes that are relatively closer to each other within the network.

Second phase

The second phase determines process-to-core bindings within each individual node. As shown by Fig. 5.4, at this phase, we only use the CPU communication pattern along with the intra-node CPU topology to calculate the desired mappings. The mapping result from the first phase is also fed to the algorithm to build the corresponding intra-node CPU communication pattern for each node. This is done by considering only the portion of the global CPU communication pattern matrix that corresponds to the set of processes mapped onto each node in Phase 1.

Note that there is no need to consider GPU-to-GPU communications at this stage as a process can be assigned any of the GPUs residing on its hosting node regardless of the core to which it is bound. In other words, we do not have the problem of remote GPUs within the boundaries of each single node. Moreover, at the intra-node level, CPU cores and GPUs do not necessarily have the same physical topology. Therefore, we leave GPU communications to the third phase where we attempt to optimize GPU assignments within each node based on the GPU communications and GPU physical topologies.

Third phase

In the third phase, we perform explicit assignment of GPUs to the processes within each node. At this stage, all processes have already been bound to a particular CPU core. As shown in Phase 3 of Fig. 5.4, GPU assignment is done with respect to the GPU communications of an application and the intra-node GPU topology. Again, the mapping result from the first phase is used to build the intra-node GPU communication pattern for each node. The explicit assignment at this phase is complementary to the implicit GPU assignment that is enforced by the process mapping mentioned in the first phase. Here, the GPUs are assigned so as to further improve intra-node GPU communications within each node.

5.3.3 Mapping Algorithms

As shown in Fig. 5.4, MAGC uses a mapping algorithm to derive the output at each phase. In general, any mapping algorithm can be used for this purpose. The algorithms used at different phases can be the same or different from each other. By modeling the communication patterns and physical topologies in terms of graphs, one can use a generic graph mapping tool such as Scotch to perform the mapping. Alternatively, one can use other fine-tuned mapping algorithms with respect to the particular physical topology of the system.

We will use the mapping approach discussed in Chapter 3 for the first phase of MAGC. For the specific systems used in our experiments in this chapter, such an approach will be simplified to its initial partitioning step because all nodes are connected to a single switch. For the second and third phases, we consider two design alternatives. In one case, we use Scotch as the mapping algorithm, whereas in another case, we use a greedy heuristic that attempts to directly optimize maximum congestion. We use the heuristic mainly to see if explicit modeling and optimization of congestion at the intra-node level can provide better

results compared to Scotch. As the heuristic is used in both the second and third phase of MAGC, we will use the general identifier of PE rather than CPU core or GPU device in the following.

Congestion-based mapping heuristic

As a general graph mapping tool, Scotch models the physical topology by a weighted graph. This makes it possible to apply the library for various systems with different architectures. However, such a generic graph model flattens the hardware architecture and does not represent the hierarchy in communication channels. Therefore, it cannot take congestion characteristics into account. Moreover, the specific edge weight values used for the graphs can highly affect the quality of Scotch results [56]. Finding an appropriate set of values can be nontrivial, specially as they can only be integers.

In our proposed heuristic, we use a tree to model the physical topology of the PEs. The tree will provide the actual hierarchy and structure of connections among the PEs. We use the hwloc and NVML [90] libraries to extract the topology of CPU cores and GPU devices within each node. Using the topology tree, we keep track of the congestion imposed on each individual link of the physical topology, and use it as the main metric to figure out the desired mapping. To this end, we take into account the actual bandwidth of each link, and define congestion as the total traffic load that is passed through each link divided by its bandwidth.

Alg. 5.1 shows the heuristic details. It takes the set of processes $\mathcal{P}$, the set of PEs, the communication pattern matrix, and the topology tree as input, and returns a mapping from the set of processes to the set of PEs. The algorithm iteratively chooses a new process and maps it to a desired PE in a greedy fashion. More specifically, at each iteration of the main loop in line 7, the new process for mapping is chosen based on the metric δ. The `for` loop in lines 8 to 10 calculates the value of δ for each unmapped process. The process with the highest value of δ is selected as the new process for mapping in line 11. As shown in

line 9, δ captures the difference between two terms. For each unmapped process, these two terms respectively calculate the average of communication volume with the set of mapped and unmapped neighbors. Accordingly, matrix $\boldsymbol{N}$ in line 6 captures the neighborhood relationships among the processes. Such a metric δ allows us to choose a process whose communications are more associated with the set of mapped processes than the unmapped ones.

The new processing element for hosting the chosen process is selected by lines 12 to 18 of Alg. 5.1. In particular, the `for` loop in lines 12 to 17 evaluates each unoccupied PE for hosting the new process, and assigns a cost to it in line 16. The cost corresponding to each unoccupied PE is equal to the maximum congestion that will result from mapping the new process onto that PE. Accordingly, as the processes are mapped to a PE, we update link congestions across the corresponding physical topology tree. This is done with respect to the communications of the newly mapped process with the set of already-mapped processes. More specifically, for each mapped neighbor n of p_{new}, we first find the links along the route from the PE that will potentially host p_{new} to the PE that hosts n. We then update the congestion of all such links with respect to the communication volume between p_{new} and n, which is given by the communication pattern matrix. Among all possible target PEs for p_{new}, we greedily choose the one that will lead to the lowest maximum congestion across all links within the physical topology tree (line 18). The new process is then mapped onto the new PE in line 19, and the topology tree is updated with the corresponding congestion values in line 20.

Complexity The main loop of Alg. 5.1 performs one iteration per process. In each iteration, In each iteration, finding the value of δ takes $\mathcal{O}(pd)$, where p and d denote the number of processes (on a single node) and the degree of the the communication pattern graph, respectively. The loop in lines 12 to 17 runs for a maximum of p iterations with a cost of $\mathcal{O}(dl)$ per iteration, where l denotes the number of links that connect two nodes

Algorithm 5.1: Congestion-based greedy heuristic for the intra-node mappings in Phase 2 and Phase 3 of MAGC

Input : Set of processes $\mathcal{P}$, set of processing elements $\mathcal{PE}$, communication pattern matrix $\boldsymbol{C} = [c_{ij}]$, topology tree T

Output: The mapping $\tau : \mathcal{P} \rightarrow \mathcal{PE}$

1 $\mathcal{P}_M = \emptyset$; `// Set of all mapped processes`

2 $\mathcal{P}_M^c = \mathcal{P}$; `// Set of all unmapped processes`

3 $\mathcal{PE}_M = \emptyset$; `// Set of occupied PEs`

4 $\mathcal{PE}_M^c = \mathcal{PE}$; `// Set of unoccupied PEs`

5 initZero(T); `// Initialize link congestions to 0`

6 $\boldsymbol{N} = [n_{ij}]_{|P|\times|P|}$, $n_{ij} = \begin{cases} 1 & c_{ij} > 0 \\ 0 & \text{otherwise} \end{cases}$;

7 **while** $\mathcal{P}_M^c \neq \emptyset$ **do**

8 **for** $q \in P_M^c$ **do**

9 $\delta = \frac{\sum_{r \in \mathcal{P}_M} (c_{qr} + c_{rq})}{\sum_{r \in \mathcal{P}_M} n_{qr}} - \frac{\sum_{s \in \mathcal{P}_M^c} (c_{qs} + c_{sq})}{\sum_{s \in \mathcal{P}_M^c} n_{qs}}$;

10 **end**

11 $p_{new} = q_{max}$; `// Choose the process with the highest value of` δ `as the new process to map`

12 **for** $pe \in \mathcal{PE}_M^c$ **do**

13 Temporarily assign p_{new} onto pe ;

14 calcCongestion(T);

15 $maxCong$ = findMaxCongestion(T) ;

16 $cost_{pe} = maxCong$;

17 **end**

18 $pe_{new} = pe_{min}$; `// Choose PE with lowest cost`

19 $\tau(p_{new}) = pe_{new}$; `// Map new process to new PE`

20 updateCongestion(T) ; `// Update the topology tree`

21 $\mathcal{P}_M = \mathcal{P}_M \cup \{p_{new}\}$, $\mathcal{P}_M^c = \mathcal{P}_M^c \setminus \{p_{new}\}$;

22 $\mathcal{PE}_M = \mathcal{PE}_M \cup \{pe_{new}\}$, $\mathcal{PE}_M^c = \mathcal{PE}_M^c \setminus \{pe_{new}\}$;

23 **end**

($l \propto \log p$ for tree topologies). Finally, updating congestions in line 20 of the algorithm takes another $\mathcal{O}(dl)$. Thus, the total complexity of Alg. 5.1 will be $\mathcal{O}(p(pd+pdl+dl)) = \mathcal{O}(p^2dl)$. It is worth noting that Scotch has a complexity of $\mathcal{O}(pd \log p + p^3) = \mathcal{O}(p^3)$[45].

5.4 Experimental Results and Analysis

5.4.1 Experimental Setup

We conduct our experiments on two GPU clusters: Cluster A and Cluster B. These clusters are part of the K80 and K20 Helios supercomputers provided by Compute Canada and installed at the University of Laval. Both clusters are equipped with multi-GPU nodes. Cluster A is composed of 4 nodes each having 16 K80 GPUs, two 12-core Intel Xeon Ivy Bridge processors operating at 2.7 GHz (total of 24 cores per node), and 256 GB of memory. Cluster B consists of 8 nodes each having 8 K20 GPUs, two 10-core Intel Xeon Ivy Bridge processors operating at 2.5 GHz (total of 20 cores per node), and 128 GB of memory. Both clusters run CentOS 6.7 and use Mellanox QDR InfiniBand as the interconnect. We use a total of 64 MPI processes in all our experiments with each MPI rank assigned to a single CPU core and GPU device. Also, we use Open MPI 1.10.2, CUDA 7.5, and Scotch 6.0.

5.4.2 Microbenchmark Results

For our microbenchmark analysis, we have developed a microbenchmark suite that models various communication patterns among the CPU cores and among the GPU devices of a cluster. The current version consists of three main microbenchmarks: 1) 2D 5-point Stencil (2D), 2) 3D 7-point Stencil (3D), and 3) Sub-communicator collective (COL)[2]. For the 2D 5-point and 3D 7-point microbenchmarks we consider two cases: a) *non-weighted* and b) *weighted*. In the former, we use the same message size for the communications along all dimensions, whereas in the latter, larger messages are used along the first dimension

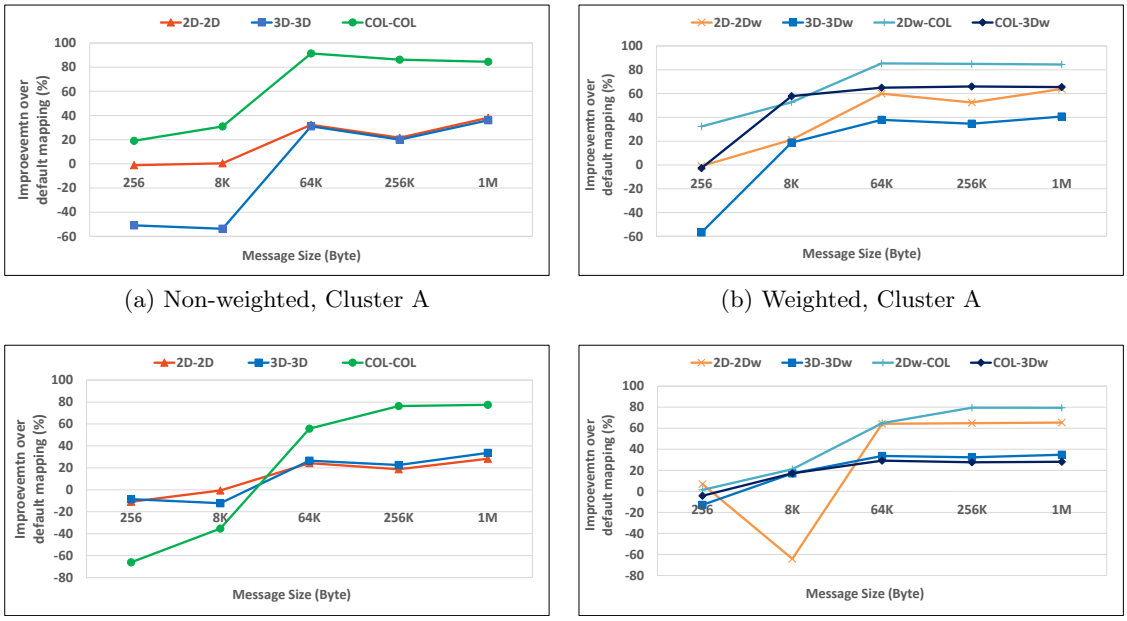
[2]See Section 3.4.1 of Chapter 3 for more details.

(3 times larger). In the sub-communicator collective microbenchmark, the processes are organized into a 3-dimensional grid with a sub-communicator created for each group of processes falling along the first dimension. An MPI collective (`MPI_Alltoall` in our tests) is called over each sub-communicator.

Each of the above-mentioned microbenchmarks can be independently used as the communication pattern among CPU cores and among GPU devices. We consider all possible combinations of such microbenchmarks (9 in total) to model a wide variety of communication patterns. We represent each combination as an X-Y pair, where X and Y respectively denote the microbenchmark of choice for CPU-to-CPU and GPU-to-GPU communications. For instance, '2D-COL' represents the case where we use the 2D pattern for CPU communications, and the sub-communicator collective pattern for GPU communications. For the weighted versions of the 2D/3D microbenchmarks, we use an additional 'w'.

Fig. 5.5 and Fig. 5.6 show the corresponding results. The figures show the communication time improvements achieved by using MAGC over the default process mappings and GPU assignments. In particular, Fig. 5.5 shows the improvement trend across 5 different message sizes. For the sake of clarity, the results are only shown for a subset of all combinations of our microbenchmarks, and Scotch is used as the mapping algorithm at the intra-node layer (Phase 2 and Phase 3). The trend is similar for the cases not shown. In Fig. 5.6, however, we show the results for all cases and a message size equal to 1MB.

From Fig. 5.5, it can be seen that for both clusters, improvements increase with message size. This is an expected behavior as MAGC is targeted for message volume and bandwidth optimizations, whereas smaller massages are more sensitive to message counts and startup latencies. In fact, we can see that for message sizes below 64KB, in some cases MAGC could result in performance degradation. However, for 64KB message size and above, MAGC can consistently improve performance for all microbenchmarks. For the COL-COL case in particular, we see more than 80% improvement. Also, we can see better results with the weighted versions of the 2D/3D microbenchmarks shown in Fig. 5.5(b) and Fig. 5.5(d).



(a) Non-weighted, Cluster A

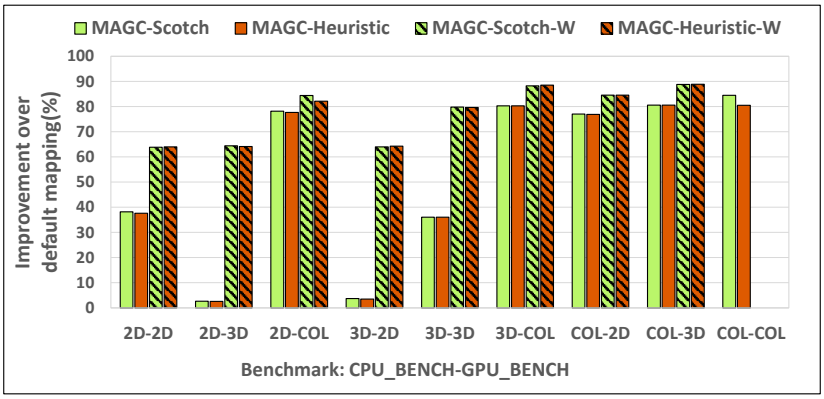
(b) Weighted, Cluster A

(c) Non-weighted, Cluster B

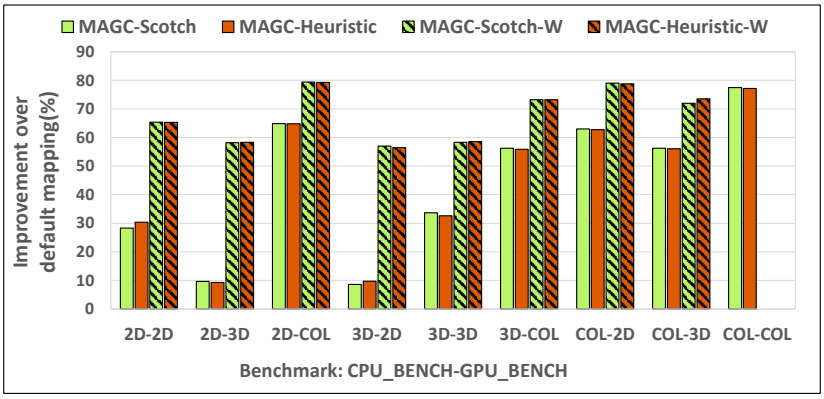
(d) Weighted, Cluster B

Figure 5.5: Microbenchmark runtime improvements for different message sizes using MAGC-Scotch on Cluster A and Cluster B—64 processes.

Fig. 5.6(a) and Fig. 5.6(b) show the improvements with 1MB message size for all microbenchmarks. The figures show the results with both Scotch and the congestion-based heuristic as the mapping algorithms used in the second and third phases of MAGC. 'MAGC-Scotch-W' and 'MAGC-Heuristic-W' refer to the weighted versions of the 2D and 3D microbenchmarks. As shown, on both clusters, MAGC can successfully improve performance for all microbenchmark combinations. The highest improvements are achieved in cases that involve the sub-communicator all-to-all benchmark. For the COL-COL microbenchmark, we can achieve 91.4% and 79.4% improvement on Cluster A and Cluster B respectively. Also, improvements are generally higher for the weighted versions of microbenchmarks. This is because the larger messages communicated along one of the grid dimensions in the weighted microbenchmarks provide more room for optimizations. Whereas the default mapping and GPU assignment is oblivious to the communication volume and bandwidth among



(a) 1MB message size and all microbenchmarks - Cluster A



(b) 1MB message size for all microbenchmarks - Cluster B

Figure 5.6: Microbenchmark runtime improvements for 1MB message size using MAGC on Cluster A and Cluster B—64 processes.

different processes, our design takes advantage of such information, and improves performance by mapping intense communications on higher-bandwidth channels. The weighted microbenchmarks also have a relatively less symmetric communication pattern, which again increases the chance for optimizations.

Another observation is that MAGC results in a similar performance with either Scotch or the heuristic. To understand the reason, we have evaluated both algorithms in terms of their impact on maximum congestion. Fig. 5.7 shows maximum congestion improvements achieved by MAGC with Scotch and the heuristic for all microbenchmarks on Cluster A.

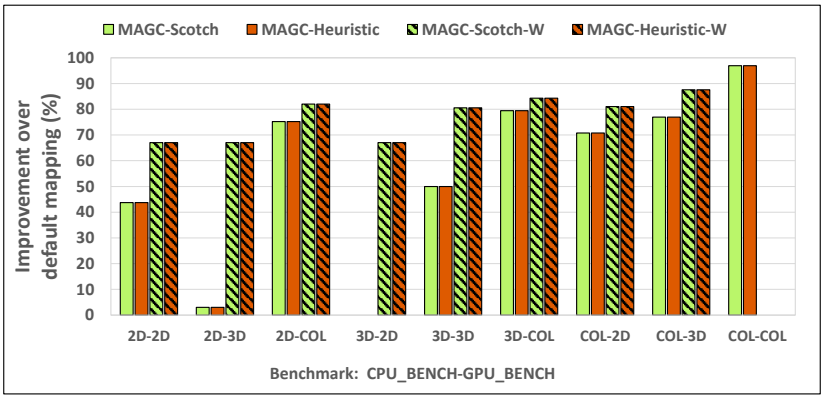


Figure 5.7: Maximum congestion improvements achieved by MAGC with Scotch and the heuristic on Cluster A—64 processes.

The results show the combined improvements with respect to both CPU and GPU communications. The trend is similar for Cluster B and hence, we refrain from repeating it here. The first observation is that in almost all cases, MAGC can successfully improve (decrease) maximum congestion; the only exception is the non-weighted 3D-2D for which we do not see any improvements. The second observation is that in all cases, Scotch and the heuristic have led to the same amount of improvement in maximum congestion.

There are two reasons for such similar performance. First, MAGC uses the same fixed mapping algorithm in the first phase, regardless of the choice (Scotch or heuristic) for the second and third phases. Thus, the impact scope of the mapping algorithms used at the second and third phases is rather limited and bound to the intra-node layer. Second, the problem size at the intra-node layer is relatively small (16 for Cluster A, 8 for Cluster B) for which both Scotch and the heuristic result in the same performance. The differences between the two can be better studied on systems with higher numbers of CPU cores and GPU devices per node. Finally, it is worth noting that a good correlation is seen between the results in Fig. 5.7 for maximum congestion and those in Fig. 5.5 for runtime improvements.

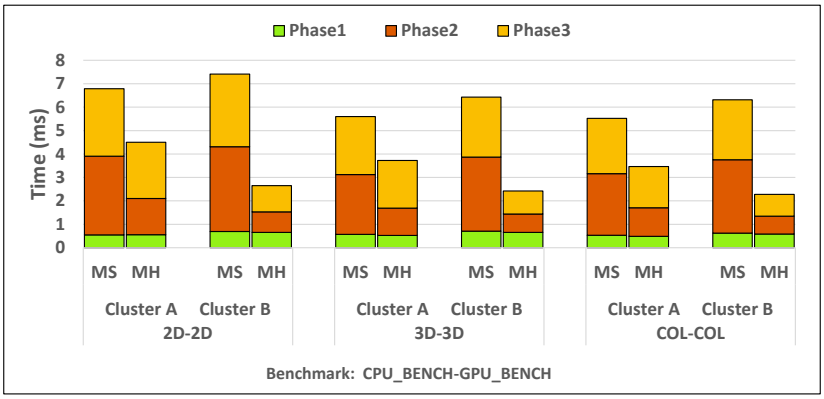


Figure 5.8: MAGC's overheads with Scotch (MS) and heuristic (MH) on Cluster A and Cluster B—64 processes.

Overheads

We also evaluate MAGC in terms of the overheads. Fig. 5.8 shows the total time spent by MAGC with Scotch (MS) and the heuristic (MH). It also shows the time breakdown across each of the three phases. For Phase 2 and Phase 3, the results represent the aggregate time across all nodes. For the sake of clarity, we only present the results for three of the microbenchmarks. The trend is similar for other microbenchmarks as well. We can see from the figure that MAGC imposes a low overhead which is less than 8 ms in all cases. We can also see that the overheads are lower with the heuristic compared to Scotch. This is specially true for Cluster B where we see considerably lower overheads with the heuristic in Phase 2 and Phase 3. This is mainly due to the fact that there are fewer CPU cores and GPU devices on Cluster B compared to Cluster A. Consequently, the heuristic will be dealing with a smaller topology tree for which the overheads of various steps in Alg. 5.1 (such as updating congestion values across the tree) will be lower. For Scotch, however, the overheads are almost the same on both clusters and we see a slightly higher overhead for Phase 2 on Cluster B.

5.4.3 Application Results

In this section, we evaluate MAGC with a real application. To this end, we use HOOMD-blue [5, 36] which is a general-purpose particle simulation toolkit shown to scale from a single CPU core to thousands of GPUs. We run the application with 64 processes on Cluster A and Cluster B, with and without MAGC. Moreover, we consider two versions of the application: single-precision and double-precision. For the input, we use the classic Lennard-Jones (LJ) liquid benchmark with two different number of particles: 512,000 and 2,000,000.

Fig. 5.9 shows the corresponding results in terms of the TPS (number of application time steps per second) improvements achieved from using MAGC in comparison with a default run of the application. We can see up to 6.5% and 8% improvements for Cluster A and Cluster B respectively. Moreover, for both clusters, the highest improvements are achieved with 512K number of particles. For Cluster A, the improvements are higher for the single-precision case, whereas on Cluster B, we see the highest improvement for the double-precision case. We also see degradations in improvement with increase in particle size on both clusters. We expected to see an opposite trend as larger number of particles would increase the total volume of messages exchanged among processes. However, a larger number of particles will also increase the total computation load of the application which can mask the impacts of communication improvements.

Our profiling results for HOOMD-blue show that the majority of communicated messages fall below 32KB on our platforms and with the workloads in use. Therefore, the messages are not large enough to consistently make the application bandwidth-bounded. Also, the communication pattern resembles a non-weighted 3-dimensional stencil with wraparounds, which makes the pattern quite symmetric. These are the main reasons for which we do not see greater performance enhancements for HOOMD-blue as they limit the room for potential optimizations through topology-aware mappings. We expect to see higher improvements

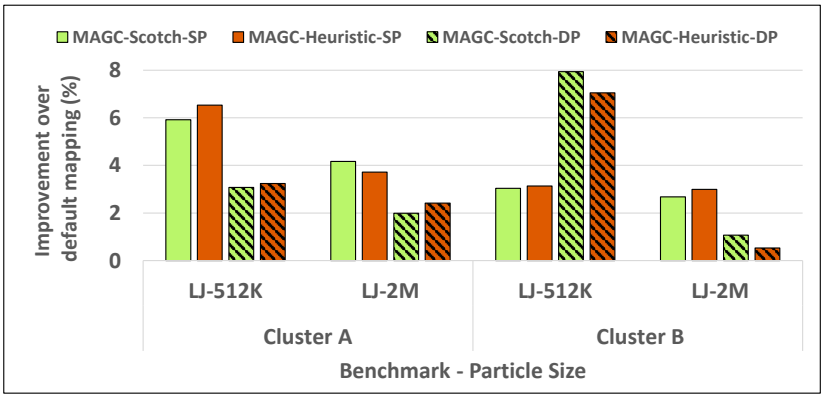


Figure 5.9: HOOMD-blue TPS (number of application time steps per second) improvements with MAGC—64 processes.

for applications that use larger messages and/or employ irregular communication patterns.

5.4.4 Comparison with Straightforward Strategies

In another set of experiments, we compare MAGC against two other strategies to further evaluate the importance of considering GPU communications and using a unified framework.

Strategy 1 In the first strategy, we use the same three phases as MAGC, but ignore GPU communications in the first phase. Thus, the processes are 1) mapped to the nodes based on the network topology and the CPU communication pattern only, 2) locally bound to specific cores within each node based on the intra-node CPU topology and CPU communication pattern, and 3) locally assigned specific GPU devices within each node based on the intra-node GPU topology and GPU communication pattern.

Strategy 2 The second strategy ignores the GPUs completely, and performs a traditional CPU-only process mapping. Thus, it will consist of two phases only where the processes are 1) mapped to the nodes based on the network topology and the CPU communication pattern only, and 2) locally bound to specific cores within each node based on the intra-node CPU topology and CPU communication pattern. For the GPU assignment, the default approach

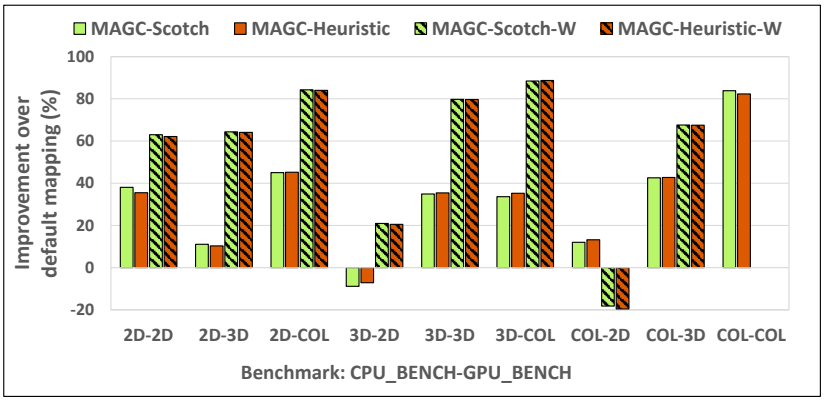
is used where each process is simply assigned the GPU that has the same ID as the process rank.

Fig. 5.10 shows the microbenchmark results for the two strategies. We only present the results for Cluster A; the trend is similar for Cluster B. By comparing the results shown in Fig. 5.10 to those shown before in Fig. 5.6(a), we can see that MAGC achieves higher performance improvements than Strategy 1 and Strategy 2 for most of the microbenchmarks. In some cases (3D-2D non-weighted and COL-2D weighted), Strategy 1 and Strategy 2 even cause up to 20% degradation in performance, whereas MAGC can provide more than 80% improvement.

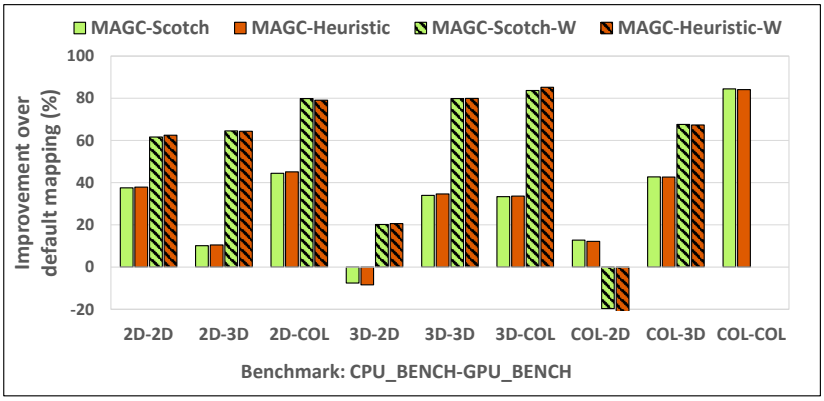
We also see similar results for the two strategies in Fig. 5.10(a) and Fig. 5.10(b), implying that the third phase of Strategy 1 (i.e., local GPU assignment within each node) does not have major impacts on the performance. This, along with the better performance achieved by MAGC, shows the importance of considering GPU communications from the first phase of the mapping methodology for GPU clusters.

Another observation is the lower consistency in performance improvements provided by Strategy 1 and Strategy 2 compared to MAGC. We see more stable improvements for MAGC across different microbenchmarks that are not affected by the type of the processing element (CPU or GPU) that runs each microbenchmark. This is demonstrated by the results for the 2D-COL versus COL-2D microbenchmarks. Whereas MAGC provides a consistent improvement (about 80%) in both cases, the two other strategies lead to different results. MAGC achieves such consistency due to its joint consideration of both CPU and GPU communications.

A closer examination of the results in Fig. 5.10 reveals that Strategy 1 and Strategy 2 can achieve a similar improvement to MAGC in cases where the same microbenchmark is used on both the CPUs and the GPUs (e.g., 2D-2D). This is expected because in such cases, optimizing for the CPU communications will also implicitly optimize the GPU communications as the two use the exact same pattern. Note that in our microbenchmark experiments,



(a) Strategy 1—1MB message size



(b) Strategy 2—1MB message size

Figure 5.10: Microbenchmark runtime improvements achieved by Strategy 1 and Strategy 2 on Cluster A—64 processes.

we use the same message size for both the CPU and the GPU communications. Therefore, with the same microbenchmarks, the communication pattern will be exactly the same on the CPUs and GPUs. However, if different message sizes were used, then we would potentially see higher improvements for MAGC even in the cases where the same microbenchmark is used on both the CPUs and GPUs.

In addition, when different microbenchmarks are used for the CPU and GPU communication patterns, Strategy 1 and Strategy 2 can perform comparably to MAGC only when we have a weighted microbenchmark on the CPUs accompanied by a non-weighted

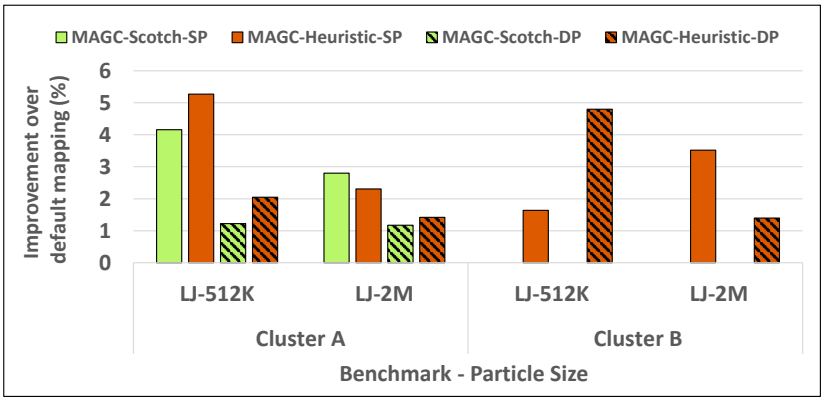
microbenchmark on the GPUs. For instance consider the 2D-COL case. We see more than 80% improvement achieved by MAGC and the other two strategies. This is because in such cases, we will have larger messages communicated among the CPUs compared to GPUs, which makes CPU communications the dominant factor in performance. However, when this is not the case (e.g., 3D-2D, COL-2D, COL-3D), we can clearly see MAGC's superior performance compared to the other two strategies. For instance, in the weighted 3D-2D case, Strategy 1 and Strategy 2 only provide 20% improvement, whereas MAGC provides more than 60%. For non-weighted 3D-2D and weighted COL-2D, we even see performance degradation with Strategy 1 and Strategy 2, whereas MAGC achieves more than 80% improvement for the weighted COL-2D case.

Finally, Fig. 5.11 shows the application improvements achieved by Strategy 1 and Strategy 2 on Cluster A and Cluster B. We can see that the improvements are lower than those achieved by MAGC in Fig. 5.9. This is specially true for Strategy 2 where we can even see performance degradation in some cases. We see relatively better results for Strategy 1 which could be due to local topology-aware GPU assignments done in its third phase. It is worth noting that unfortunately, we could not get the results for Strategy 1 with Scotch on Cluster B (missing bars in Fig. 5.11(a)) due to a failure in the corresponding experiment.

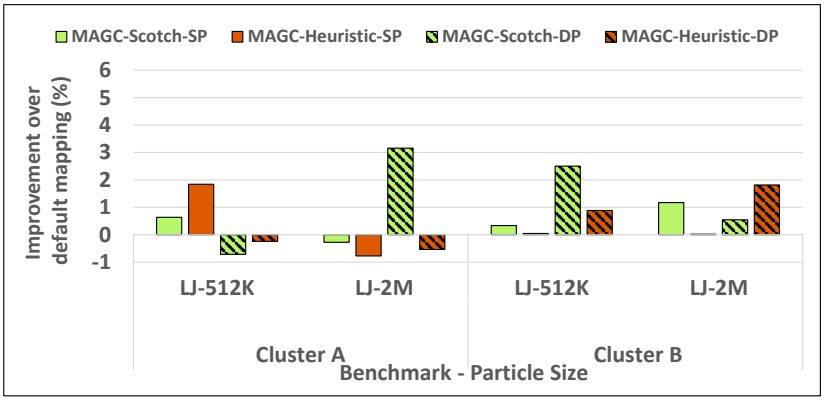
5.5 Related Work

In GPU clusters, researchers have studied various GPU-aware point-to-point and collective operations to improve the GPU communication performance [30, 57, 97]. Faraji and Afsahi [30] propose an intranode GPU-aware MPI_Allreduce in which CUDA IPC is used to gather the pertinent data into the shared GPU buffer, followed by an in-GPU reduction. The CUDA IPC copy type has also been utilized by other researchers [57, 97] to improve one-sided and point-to-point communications.

Martinasso et al. [79] provide a detailed analysis of the congestion behavior associated



(a) Strategy 1



(b) Strategy 2

Figure 5.11: HOOMD-blue TPS improvements achieved by Strategy 1 and Strategy 2—64 processes.

with the PCIe fabric that is used to connect the GPUs in a multi-GPU node. Accordingly, a congestion-aware performance model is proposed that can be used to predict the communication times in presence of congestion on a given PCIe topology. The proposed model can help to design more efficient algorithms for intra-node GPU communications. Lutz et al. [76] propose an auto-tuning framework for distribution of stencil computations across multiple GPUs. They show that various PCIe layouts can have adverse effects on the performance, thereby utilizing all GPUs might not be necessarily a better choice in all cases.

5.6 Summary

In this chapter, we focused on topology awareness in the context of heterogeneous GPU clusters. We discussed the physical topology of GPU communication channels and how it can impact the performance of GPU communications. Next, we discussed the joint problem of process-to-core mapping and GPU-to-process assignment in GPU clusters. Accordingly, we proposed a unified mapping approach called MAGC that takes into account both the CPU and GPU communication patterns of an application, as well as the CPU and GPU physical topologies of the system. MAGC exploits a three-phase approach that first assigns processes to the nodes across the network, and then performs CPU core bindings and GPU assignments within each node.

We studied MAGC's benefits with two different algorithms used for finding the desired core bindings and GPU assignments within each node. The first algorithm used the Scotch library, whereas for the second one we designed a congestion-based heuristic. The heuristic uses the maximum congestion imposed on the intra-node communication channels as the metric to find the desired mappings. Our experimental results showed that using MAGC, we can achieve considerable performance improvements by a more efficient utilization of both the CPU and GPU communication channels within a GPU cluster. The results showed similar improvements with Scotch and the congestion-based heuristic. However, the overheads were shown to be lower for the latter. We also compared MAGC's performance with two straightforward strategies and showed that it can outperform them considerably, which provides further indication of the importance of using a unified framework for topology-aware process mapping and GPU assignment in GPU clusters.

Chapter 6

Neighborhood Collective Communications Optimization

As discussed in Chapter 2, neighborhood collectives are the relatively new communication type added to MPI to address certain demands and shortcomings. By abstracting application communications into neighborhood collectives, MPI provides users with yet another opportunity to optimize performance. The topology information associated with a neighborhood collective can be exploited by the MPI library to derive an optimized communication pattern. Currently, well-known MPI libraries such as MPICH [86], MVAPICH [87], and OpenMPI [92] use a naïve approach for performing neighborhood collectives. Every process simply issues a send (receive) operation to (from) each of its outgoing (incoming) neighbors. These send/receive operations are usually issued all at once in a nonblocking fashion and are treated just as a set of individual point-to-point communications. Thus, no specific pattern is used to govern the communications in order to deliver better performance.

In this chapter, we discuss how to improve the performance of neighborhood collectives in MPI through designing nontrivial communication schedules. We use no prior knowledge about the neighborhood topology, and hence target the distributed graph topologies in MPI. Distributed graph topologies provide the most flexible and generic way for modeling process topologies in MPI, but they make it more challenging to optimize neighborhood collectives on top of them. We show, however, that useful information still can be extracted from distributed graph topologies for communication optimizations. In particular, we show how

to discover and exploit *common neighborhoods* in such topologies in order to optimize the performance of neighborhood collectives through message combining.

More specifically, we propose a distributed algorithm that can be used to extract message-combining communication patterns and schedules for neighborhood collectives. We show that part of the problem falls within the scope of maximum matching in weighted graphs, where we seek to find a mutual pairing of the processes that have neighbors in common. To the best of our knowledge, this is the first work that shows how common neighborhoods and message combining can be used to optimize the performance of neighborhood collectives over the generic distributed graph topology interface of MPI.

6.1 Designing Nontrivial Algorithms for Neighborhood Collectives

Collective communications optimization can generally be divided into two branches: optimization for small messages and optimization for large messages. The optimization objective is different for small messages as compared to the objective for large messages. For small messages the algorithms mainly seek to reduce the number of communication stages (start-up latencies), whereas for large messages it is the total volume of transferred messages, as well as the congestion induced by the underlying algorithm, that matters.

6.1.1 Design Principles

In this chapter, we focus mainly on small-message communications. Therefore, we wish to determine how to potentially decrease the number of communication stages in a neighborhood collective. Accordingly, we identify the following two principles that can be used to decrease the number of communication stages in a certain collective communication:

1. increasing the number of senders in each stage,
2. increasing the number of messages transferred to a destination in each send operation (in each stage).

These two principles can also be found at the core of well-known algorithms used for conventional collectives [116]. Two good examples are the binomial tree and the recursive doubling algorithms used for `MPI_Bcast` and `MPI_Allgather`, respectively. Binomial broadcast tries to increase the number of simultaneous senders by having every process contribute as a source for message transmission as soon as it receives the broadcast message. For `MPI_Allgather` however, every process is already contributing as a source from the beginning of the collective call. Therefore, the recursive doubling algorithm for `MPI_Allgather` tries to decrease the number of communication stages by increasing the number of messages that are transferred in each individual send operation through message combining. However, optimizing neighborhood collectives is more challenging. The reason is that conventional collectives describe a *global* communication among *all* the processes. This provides global knowledge of the desired collective communication, locally at each process. With neighborhood collectives, however, the communication pattern is described by a graph that is distributed among the processes. Thus, each process has only a local view of the entire collective pattern.

For a neighborhood collective, the first principle mentioned above can help improve performance when the processes have an *unbalanced* number of neighbors in the underlying topology graph. More specifically, we can increase the number of senders in each stage by delegating a portion of communications from the processes that have a high number of outgoing neighbors to those having lower outdegrees. This strategy helps balance the number of messages that should be sent out by each process, which in turn helps avoid situations where one process is overloaded with many messages to send while some other processes are idle. The tree transformation algorithm discussed by Hoefler and Schneider [43] is an instance of such an optimization.

In this chapter, we discuss how the second principle mentioned above can be used to improve the performance of neighborhood collectives. More specifically, we are interested in designing a nontrivial communication algorithm that exploits message combining in order

to decrease the number of communication stages of a neighborhood collective. Although such an optimization can be used in conjunction with the neighbor-balancing approach, it becomes particularly useful in cases where neighbor balancing fails to provide any benefits—for instance, where process topologies are already balanced and the number of outgoing neighbors of all processes is approximately the same. Such cases, combined with the fact that most applications expose a balanced topology graph, add to the importance of our proposed design.

6.1.2 Common Neighborhoods

We apply message combining to neighborhood collectives by exploiting the potential *common neighborhoods* that might exist in the underlying process topology graph. We define the common neighborhood of p_1 and p_2 as the set of all processes that are an outgoing neighbor of both p_1 and p_2. The existence of common neighborhoods among the processes in a topology graph provides an opportunity for decreasing the number of individual messages that should be sent out by each process.

To clarify this point, we consider the sample topology graph shown in Fig. 6.1, where p_1 and p_2 share the processes $n_1, n_2, \ldots, n_k$ as a part of their outgoing neighbors. In a trivial design, p_1 and p_2 each send one message to each of $n_1, n_2, \ldots, n_k$ neighbor processes, which adds up to k communication stages.[1] However, we can use message combining to evenly divide the common neighbors between p_1 and p_2 so that they each require communicating with only half of their neighbors that fall within the common neighborhood. More specifically, p_1 and p_2 can first communicate with each other to exchange their messages and build a combined message consisting of both p_1 and p_2 messages. Next, using the combined message, p_1 and p_2 respectively communicate with $n_1, n_2, \ldots, n_{\frac{k}{2}}$ and $n_{\frac{k}{2}+1}, n_{\frac{k}{2}+2}, \ldots, n_k$. This step transfers the message of both p_1 and p_2 to the processes in their common neighborhood in $\frac{k}{2} + 1$ communication stages.

[1] Of course more stages are required to cover other non-common neighbors.

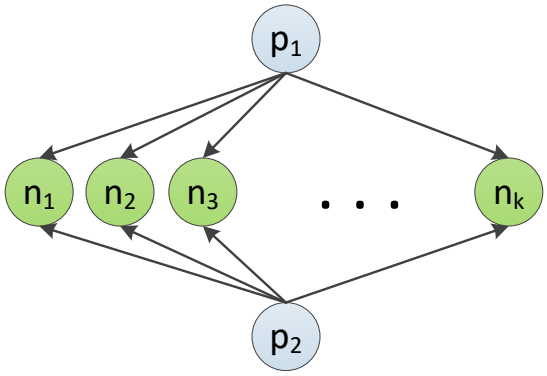


Figure 6.1: Sample process topology graph showing a common neighborhood for p_1 and p_2 consisting of k processes $n_1, n_2, \ldots, n_k$.

Accordingly, we propose an approach that finds the common neighborhoods among the processes in an MPI distributed graph topology and exploits them to design more efficient communication schedules for neighborhood collectives. We distinguish the following two phases in our design:

1. Phase 1 for building a communication *pattern* based on the given topology graph,
2. Phase 2 for building a communication *schedule* based on the derived pattern and a specific neighborhood collective

In the following sections, we discuss each of these phases in more detail.

6.2 Communication Pattern Design

The first phase is the main part of our design, in which we build a nontrivial communication pattern among the processes based on pairwise message combining between the processes that share a common neighborhood. The resulting communication pattern consists of a series of message-combining steps for each process. We note that this phase depends only on the given topology. Thus, it needs to be done *only once for each given process topology*

graph. Moreover, the result from this phase can be used for different types of neighborhood collectives. For instance, it can be used to build a communication schedule for both neighborhood allgather and neighborhood alltoall collectives.

6.2.1 Distributed Design

A major challenge is that we commit ourself to a fully distributed algorithm that runs on each process and builds a communication pattern that locally describes the specific communications of each process. A distributed design potentially has lower overheads and better scalability. Moreover, we want to maintain the distributed representation of the underlying topology provided by the MPI distributed graph topology functions. This is important: the MPI distributed topology functions were added to the standard because the old non-distributed versions of such functions were well known to be one of the most non-scalable structures in MPI [6]. Thus, we intend to avoid any central solution that requires building the complete topology graph at one or more processes.

Our proposed approach works with both the adjacent and non-adjacent distributed graph topology interfaces of MPI. The adjacent API has the advantage that the information about the outgoing/incoming neighbors of each process is available at no additional cost, whereas extracting such information from the non-adjacent API might require additional communications. However, this will be a one-time cost only and the standard already includes APIs to query the outgoing/incoming neighbors of each process. In addition, the adjacent API provides a more natural way for programmers to describe the topology graph and has been shown to have a slightly better performance for neighborhood collectives [119].

6.2.2 Message-Combining Algorithm

In order to describe our proposed message-combining algorithm, we first establish the following definition.

Friend processes: Two processes are defined as a *friend* of each other if they have

a certain number of outgoing neighbors in common. The two processes are said to be in a *friendship relationship*. This relationship is defined with respect to a given threshold Θ specifying the minimum number of common neighbors that two processes should have in order to be considered friends. Note that two friend processes may or may not be neighbors themselves.

Algorithm 6.1 shows a high-level abstraction of the main steps in our proposed algorithm. Each process p in the topology graph runs Alg. 6.1 to build its local portion of the desired communication pattern. First, we extract information about the common neighborhoods of p and save it into a matrix M (line 2). We explain this step in more detail in a separate section. Next, from M, we find all friends of p with respect to an input friendship threshold Θ (line 6). We also initialize O_a and I_a (line 4), which denote the list of *active* outgoing and incoming neighbors of p, respectively. Active neighbors represent a subset of neighbors that have not yet been addressed. After that, in the main loop (lines 5 to 25) we iteratively perform three main tasks: (1) pair p with one of its friends for the purpose of message combining, (2) divide the corresponding common neighbors between p and its paired friend, and (3) update topology information and the output pattern.

More specifically, p first attempts to find a target friend such as f to pair with (line 6). This is a major step and we discuss it in a separate section. Having found f, we extract the set of outgoing neighbors that p and f have in common (line 8) and divide it evenly between p and f (lines 10 to 17). More specifically, let CN represent the set of common neighbors between p and f. Then, we divide CN into two balanced subsets CN_{on} and CN_{off}, which respectively denote the neighbors assigned (onloaded) to p and the neighbors offloaded to f.

A key point here is that we have to make sure the division is consistent at p and f. In other words, CN_{on} and CN_{off} should conform to the following conditions:

1. $CN_{on} \cap CN_{off} = \emptyset$

Algorithm 6.1: Distributed message combining for neighborhood collectives

Input : Set of outgoing neighbors O, set of incoming neighbors I, friendship threshold Θ

Output: The communication pattern $\mathcal{T}$

```
1  p = this process;
2  M = Build_common_neighborhood_matrix(O, I) ;
3  F = Find_friends(M, Θ) ;
4  O_a = O, I_a = I;
5  while |F| > 0 do
6  |   f = Find_friend_to_pair(M, F);
7  |   if found f then
8  |   |   CN = Find_common_neighbors(M, f);
9  |   |   sort(CN);
10 |   |   if p < f then
11 |   |   |   Keep (onload) the first half of CN;
12 |   |   |   Offload the second half of CN to f;
13 |   |   end
14 |   |   else
15 |   |   |   Keep (onload) the second half of CN;
16 |   |   |   Offload the first half of CN to f;
17 |   |   end
18 |   |   Add f and CN_on to T ;
19 |   end
20 |   Notify each neighbor in O_a whether it was onloaded/offloaded ;
21 |   Update I_a and T based on notifications from neighbors in I_a ;
22 |   O_a = O_a − CN ;
23 |   Update_common_neighborhood_matrix(M, O_a, I_a);
24 |   F = Find_friends(M, Θ) ;
25 end
26 Notify each neighbor in O_a that p is done;
27 Add O_a to T;
28 while I_a ≠ ∅ do
29 |   Update I_a and T based on notifications from neighbors in I_a;
30 end
```

2. $CN_{on} \cup CN_{off} = CN$

3. CN_{on} at $p = CN_{off}$ at f

4. CN_{off} at $p = CN_{on}$ at f

To ensure these conditions are met, we first sort the list of common neighbors with respect to the MPI rank of the corresponding processes (line 9). The list is then divided in half, and we decide which half to assign to p based on the MPI rank of p and f. If the rank of p is less than f, all the neighbors in the first half of the sorted CN are assigned to p, and vice versa.

At this point, we have built one major stage of our target communication pattern for p (and also f), which implies p exchanging messages with f, building a combined message, and sending it to all the processes in CN_{on}, namely the common outgoing neighbors assigned (onloaded) to p. Thus, we save f and CN_{on} into the output pattern $\mathcal{T}$ (line 18).

At the end of each iteration, we notify each active outgoing neighbor of p about the outcome of the current iteration. This notification (line 20) is used for three purposes:

1. informing offloaded neighbors of p that they will not receive any message from p in the final communication pattern,

2. informing onloaded neighbors of p that they will receive a combined message in the final communication pattern, which includes the data from p and f,

3. updating the list of active incoming neighbors of each process (line 21), which is necessary for updating neighborhood information for the next iteration.

We also update the list of active outgoing neighbors to remove those covered in the current iteration (line 22). Then we update the common neighborhood matrix based on the remaining active outgoing/incoming neighbors of each process (line 23), and we recompile the list of remaining friends accordingly (line 24). The details of how we update the common

neighborhood matrix are discussed in a separate section. As shown by the condition in line 5, the main loop of the algorithm finishes when there are no more remaining friends for the process to consider for pairing. The process p then notifies its remaining active outgoing neighbors that it is done (line 26) and adds them to $\mathcal{T}$ (line 27). Next, it remains active to receive the notifications sent to it from the set of its active incoming neighbors (line 29). The notifications are used to update both the pattern ($\mathcal{T}$) and the set of active incoming neighbors (I_a) itself.

Finding common neighborhoods

The first step to building our desired pattern is finding the common neighborhoods in the topology graph (line 2 of Alg. 6.1). More specifically, each process has to find the set of all other processes with which it has some of its outgoing neighbors in common. Fortunately, this can be accomplished in an MPI distributed graph topology by a set of neighborhood communications. To this end, each process queries each of its *outgoing* neighbors about their *incoming* neighbors. For each outgoing neighbor n_{out} of a given process p, the incoming neighbors of n_{out} represent the set of all processes with which p has n_{out} as a common neighbor. By querying all its outgoing neighbors, p builds a common neighborhood matrix M. Each row of M corresponds to one of the outgoing neighbors of p such as n_i, and it lists the set of all incoming neighbors of n_i. Therefore, each element in row n_i represents the rank of a process with which p has n_i in common as an outgoing neighbor. By exploring all the elements in matrix M, we can find all the processes with which p has at least one neighbor in common. Also, the number of times a process rank appears within M designates the number of neighbors that p and the given process have in common (the common neighborhood size). Hence, we can find all friends of p with respect to a given friendship threshold Θ (line 3 of Alg. 6.1).

Each process already has the list of all its outgoing and incoming neighbors, which makes the extraction of M easier. Every process in the topology needs only to send its own list of

(already known) incoming neighbors to each of its incoming neighbors and receive one such list from each of its outgoing neighbors. We note that M requires $O(d^2)$ memory space, where d denotes the average outdegree of the processes in the topology graph.

Pairing friend processes

In each iteration of the loop in Alg. 6.1, each process p attempts to pair with one (and only one) of its friend processes (line 6). Among all possible friends, we want p to choose a friend with which it has the highest number of common neighbors because that provides more opportunity for performance optimization. However, the friend selection must be mutual between the two processes involved. If p chooses f as its target friend to pair with, then we must ensure that f has also chosen p mutually as its target friend.

The friend-pairing problem can be modeled as a distributed maximum matching problem in weighted graphs. The corresponding graph is the *friendship graph* of processes $G(V, E)$. For each process p, we have a vertex $v_p \in V$, and two vertices v_p, v_q are adjacent ($(v_p, v_q) \in E$) if and only if p and q are friends. The edge weight represents the number of common neighbors between p and q.

Finding a maximum matching in the friendship graph G pairs each process with at most one other friend and does this pairing so that the total number of common neighbors covered in the main topology graph is maximized. However, note that G is inherently distributed among the processes, because each process extracts information only about its own neighborhood and friends (lines 2 and 3 of Alg. 6.1). Thus, we have a distributed maximum weighted matching (MWM) problem.

Various algorithms with different complexities and approximation ratios have been proposed for the distributed MWM problem [122, 47, 73, 74, 63]. In this chapter, we use an algorithm similar to that proposed by Hoepman [47]. A recent study [50] shows that Hoepman's algorithm outperforms other major alternatives for distributed MWM. Algorithm 6.2 shows the details of our algorithm for distributed matching of friend processes. The

main differences between Alg. 6.2 and that designed by Hoepman are in the information that is communicated among the processes and the way we choose the candidates in each iteration. In Hoepmans algorithm, a node u sends a request signal to its chosen candidate only, whereas in our algorithm, u informs all its neighbors (in the friendship graph) about its chosen candidate. Moreover, whereas Hoepman always chooses the locally heaviest edge, we use the procedure outlined in lines 4 to 23 of Alg. 6.2. The rationale for our approach is described below.

Algorithm 6.2 is an iterative algorithm where each process first chooses a potential friend for pairing and then checks to see whether its choice is mutual. If it is, then we have a successful pairing, and this step is completed. Otherwise, the process goes to the next iteration to try another (or the same) friend again. A given process p runs the main loop in lines 3 to 30 until it reaches either a *paired* or *terminal* state. The former represents the case where p can successfully find a mutual friend, whereas the latter represents the case where p fails to find a mutual friend and gives up the search. Friend selections occur in lines 4 to 23. The remaining steps are used for communicating the choices to check their mutuality.

In the first iteration of the loop, each process p greedily chooses the friend with which it has the maximum number of common neighbors. In all other iterations, we consider two main cases: (1) p maintains its previously chosen friend f_{old} (lines 8 to 10), or (2) p tries one of its other still-active friends (lines 11 to 22). As shown by the condition in line 8, p maintains its previously chosen friend f_{old} only if it finds that f_{old} has also failed to pair with its previously chosen friend. The rationale is that in such a case, f_{old} will still be looking for a friend to pair with and it might choose p this time. Thus, it makes sense for p to maintain f_{old}. However, p does so only if two other conditions are also met: (1) f_{old} has not become terminal and hence is still looking for a friend, and (2) the rank of p is lower than the rank of f_{old}. This second condition is needed to avoid deadlock situations where the set of friends chosen by a group of processes creates a circular dependency. Without the

Algorithm 6.2: Distributed and mutual friend pairing/matching.

Input : Common neighborhood matrix M, Set of friends F
Output: A friend process f to pair with for message combining

```
1  p = this process;
2  paired = terminal = False;
3  while !(paired OR terminal) do
4      if first iteration then
5          f = friend with maximum number of common neighbors;
6      end
7      else
8          if f_old is not terminal AND is not paired AND p < f_old then
9              f = f_old; // Retry the previous choice
10         end
11         else
12             f = g ∈ F : g < p and g chose p;
13             if not found g then
14                 F = F − f_old ;
15                 if F is empty then
16                     terminal = True ; // failed to pair with anyone
17                 end
18                 else
19                     f = friend with maximum number of common neighbors;
20                 end
21             end
22         end
23     end
24     Notify all friends in F about f;
25     if p is not terminal AND f chose p then
26         paired = True ; // Successfully paired with f
27     end
28     Notify all friends in F about paired state;
29     Remove from F all the paired and terminal friends;
30 end
```

second condition, such processes will all maintain their previously chosen friend, causing them all to fail indefinitely.

In choosing a new friend, we first search for a friend g such that rank of g is lower than the rank of p, and g has chosen p as its potential friend (line 12). The reason is that we know from the first case above that g will choose p again as its potential friend and, hence, p and g could successfully pair. If g does not exist, we attempt to choose a new friend with which p has the highest number of common neighbors (line 19). However, we first discard the previously chosen friend from the list of friends in line 14, and we put the process in the *terminal* state if there are no more friends left to try. Doing so guarantees a bounded number of iterations when a process fails to mutually pair with a friend.

At the end of each iteration, we have two communication steps. In the first one (line 24), p sends (receives) the value of f to (from) all its remaining friends. Next (line 25), we check to see whether the friend selections are mutual between p and f, and we set the *paired* state accordingly. The selection is mutual if p is not terminal (i.e., has actually chosen a friend) and the target friend selected by its chosen friend (f) is equal to p. In the second communication step (line 28), p informs its friends about its success/failure in mutual pairing based on the information received in the first communication step. Next, we update the list of friends to remove all the paired and terminal ones. Such friends will not be active in the next iteration of the algorithm.

Updating neighborhood information

At the end of each iteration of Alg. 6.1, we update the neighborhood information before moving to the next iteration (line 23 of Alg. 6.1). This step is necessary because the neighbor offloading/onloading changes the effective neighborhoods of a process. More specifically, when a process such as p offloads some of its outgoing neighbors to a friend process f, it no longer is communicating (directly) with those neighbors, thus effectively canceling those processes as outgoing neighbors of p. Also, in our current design, each outgoing neighbor

of a process is considered for at most one message-combining stage; that is, the onloaded neighbors of a process are not considered for message combining with other friends in further iterations of the algorithm. Thus, before choosing another friend for message combining, we first need to update the common neighborhood matrix with respect to such changes in effective outgoing neighbors.

We can do this update in a way similar to that used for building the common neighborhood matrix in the first place. Each process queries each of its outgoing neighbors about their remaining active incoming neighbors. To this end, each process sends its own list of active incoming neighbors (I_a) to each of its active incoming neighbors and receives one such list from each of its active outgoing neighbors. Next, we compare each received list with its corresponding row in the common neighborhood matrix. This is the row of the matrix that corresponds to the outgoing neighbor from which the list has been received. Then, we mask from the matrix those elements that are no longer in the list.

Complexities

The complexity of Alg. 6.2 can be given by $\mathcal{O}(rf)$, where r denotes the number of iterations of the algorithm and f denotes the number of friends per process (node degrees in the friendship graph). This leads to a *worst-case* complexity of $\mathcal{O}(p^2)$ for Alg. 6.2, where p denotes the number of processes. The reason is that in worst case, $r = p - 1$ due to lines 8 to 10 of Alg. 6.2, and $f = p - 1$ for a complete friendship graph.

The complexity of Alg. 6.1 can be given by $\mathcal{O}(t(rf + n^2))$, where t denotes the number of iterations of the algorithm and n denotes the average number of neighbors per process (node degrees in the topology graph). The n^2 term is for traversing the neighborhood matrix to find the friends of a process. Assuming[2] an upper bound of f for t, the *worst-case* complexity of Alg. 6.1 will be $\mathcal{O}(p(p^2 + p^2)) = \mathcal{O}(p^3)$ with a complete topology graph.

We note that these are the worst-case complexities. In Section 6.4.3, we show that in

[2]We do not have a formal proof for it.

practical use cases, Alg. 6.1 overhead increases with $\sqrt{f}(f + n^2)$. Moreover, n is expected to be considerably lower than p for the topology graphs used with neighborhood collectives. Thus, a complete process topology graph is not a realistic use case as it would actually represent a conventional global allgather/alltoall communication.

6.3 Communication Schedule Design

So far, we have explained how to design an optimized communication *pattern* based on the underlying process topology graph. In this section, we explain how to build a communication *schedule* from the derived pattern for a given neighborhood collective function. The communication schedule precisely specifies the send, receive, and memory copying operations that a process should perform in order to implement the derived communication pattern. This means it has the buffer addresses associated with it. Moreover, it specifies the exact order of such operations, providing a fine-grained description of the temporal characteristics of the desired communication pattern. Therefore, unlike the pattern that is built once for each topology graph, the schedule should be built *once for each specific call to a neighborhood collective*.

6.3.1 Generic Scheme

Algorithm 6.3 shows the steps involved in building the communication schedule. Again, each process p runs the algorithm to build the desired schedule. As the input, it takes the message-combining pattern $\mathcal{T}$ and all the parameters corresponding to a specific neighborhood collective call, including send/receive buffer addresses and message sizes (counts). The output is a communication schedule $\mathcal{S}$ that is used to conduct the given neighborhood collective. The $+=$ symbol is used to represent the ordered addition of operations to the schedule. From a generic perspective, Alg. 6.3 builds a communication schedule that consists of several blocking communication steps. Each step mainly captures a series of

nonblocking point-to-point send/receive operations that are issued by each process.

Algorithm 6.3: Communication schedule extraction from the message-combining pattern.

Input : Communication pattern $\mathcal{T}$, send/receive buffers, send/receive sizes, send/receive datatypes, MPI communicator
Output: Communication schedule $\mathcal{S}$

```
1  for each step t in T do
2      if have a combining friend f then
3          S += send operation to f;
4          S += receive operation from f;
5          S += wait on issued operations;
6          S += build the combined message;
7          for each onloaded neighbor n_o do
8              S += send operation to n_o (combined message);
9          end
10     end
11     for each incoming neighbor n_i tagged with t do
12         S += receive operation from n_i to get a message (possibly combined);
13     end
14     S += wait on issued operations;
15     for each incoming neighbor n_i tagged with t do
16         S += copy operation to move the message contents to the final buffers;
17     end
18     S += wait on issued operations;
19 end
20 for each remaining outgoing neighbor n_o do
21     S += send operation to n_o;
22 end
23 Repeat lines 11 to 17 for the remaining incoming neighbors;
```

Accordingly, the loop in lines 1 to 19 processes the pattern $\mathcal{T}$ one step at a time. Each step of $\mathcal{T}$ contains the results of the corresponding iterations of Alg. 6.1, which can include two items: (1) a friend process f to combine messages with and a corresponding set of onloaded outgoing neighbors and (2) a set of incoming neighbors to receive messages from. If $\mathcal{T}$ contains a message-combining friend f at step t, we add to the schedule a send (receive) operation to (from) f for exchanging messages between p and f (lines 3 and 4). Next, we add a directive to the schedule (line 5) to note that the succeeding operations should not be

started until all the previous ones are completed. Then, we add an operation to build the desired combined message, which in our current implementation translates into a memory copy operation. After that, we add a send operation (lines 7 to 9) to transfer the combined message from p to each of its onloaded outgoing neighbors.

Lines 11 to 17 of Alg. 6.3 process the list of incoming neighbors from which p should receive a message at step t. For each of such incoming neighbors, we add a receive operation to $\mathcal{S}$ to receive a message into an intermediate buffer (line 12). In line 16 we add a memory copy operation to $\mathcal{S}$ to move data into the final application buffers. First, however, we add a directive to $\mathcal{S}$ (line 14) to ensure that such data movements are not started before the messages arrive in the intermediate buffers. Before processing the next step t, we add another directive to $\mathcal{S}$ (line 18) to block any further operations until the current step completes. At the end, we add a send (receive) operation to $\mathcal{S}$ for each remaining outgoing (incoming) neighbor that has not been covered in any of the previous steps of $\mathcal{T}$ (lines 20 to 23). These operations constitute a final logical step of communications in the schedule and correspond to lines 27 to 30 of Alg. 6.1. We note that Alg. 6.3 has a complexity of $\mathcal{O}(n)$, where n denotes the number of neighbors per process in the topology graph.

6.3.2 Specific Designs

We have currently devised two specific versions of Alg. 6.3 to support the schedule design for two of the MPI neighborhood collectives, namely, neighbor allgather and neighbor alltoallv[3]. We have chosen these two functions because they have important differences in terms of their corresponding schedule design.

The design for alltoallv is more involved than that of allgather because of two main reasons. First, alltoallv/alltoall involves more complicated data exchange and combined-message build-up stages (lines 3, 4 and 6 of Alg. 6.3). The reason is that in alltoallv/alltoall we will have a different message for each (onloaded) neighbor, whereas in allgather, the same

[3]The design for neighbor alltoall easily follows from alltoallv.

exchanged/combined message is used for all onloaded neighbors at each stage.

Second, for allgather, all the required arguments for the send/receive operations that are added to the schedule are locally available at each process. This is mainly because with allgather, we know that the incoming neighbors of a process p will all send the same amount of data to p. However, in alltoallv, each process could send (receive) different amount of data to (from) each of its outgoing (incoming) neighbors. The `sendcounts` (`recvcounts`) argument of the function determines the specific amount of data for each neighbor. As a result, two paired friend processes could have different `sendcounts` values for the common neighbors that are onloaded/offloaded between them. Thus, for alltoallv, Alg. 6.3 will involve communication between the paired friends so as to exchange certain parts of the `sendcounts` argument. We note that unlike the send/receive operations listed in Alg. 6.3, these communications are not part of the final schedule.

6.4 Experimental Results and Analysis

In this section, we evaluate the performance of our design for various neighborhood topologies. We use a microbenchmark to measure the latency of neighborhood collectives with and without our message-combining approach. In our microbenchmark, we first build a desired process topology graph and then call the neighborhood collective function on top of it. The function is called 1,000 times and the average latency per call is reported as the output[4]. All the experiments are conducted on the GPC cluster at SciNet (see Section 3.4 for details) and we use MVAPICH2-2.2.

We evaluate the performance for two flavors of our design and the default naïve approach: *persistent* and *nonpersistent*. The persistent approach is useful in cases where the neighborhood collective function parameters (such as buffer addresses, counts, etc.) are known to remain unchanged. Thus, in the persistent approach, the neighborhood collective schedule is built *once* in the first call to the collective (or in an "`init`" API) and is reused

[4]The first 100 calls are ignored for the sake of warm-up.

for all succeeding calls. In the nonpersistent approach, the schedule is built every time that the neighborhood collective is called. Note that the communication pattern is built only once in both cases; only the schedule is built for every call to the collective function in the nonpersistent case. In the following charts, we report the results for four cases:

1. Def-Nonpersist: The default naïve approach in MVAPICH
2. Opt-Nonpersist: Our proposed message-combining design
3. Def-Persist: The persistent version of the default naïve approach in MVAPICH
4. Opt-Persist: The persistent version of our proposed message-combining design

It is also worth noting that we use a friendship threshold of $\theta = 4$ in all the experiments as it represents the minimum number of common neighbors that could potentially lead to performance improvement through message combining. If two processes have fewer than 4 neighbors in common, then we cannot decrease the number of communication stages through message combining due to the additional communication needed to exchange messages between the friend processes.

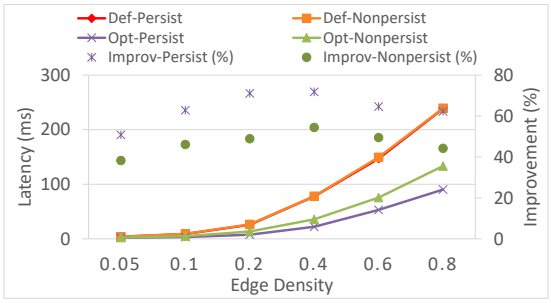
6.4.1 Random Sparse Graph

In our first set of experiments, we build the neighborhood topology based on a random sparse digraph $G(V, E)$. The set of vertices V represents the set of MPI processes, and an edge $(i, j) \in E$ represents an outgoing neighbor from process i to process j. The edges of the graph are created randomly with respect to a density factor $0 < p \leq 1$, which is provided as an input. Higher values of p result in a denser graph with a higher number of outgoing/incoming neighbors per process, whereas lower values of p produce a sparser graph with fewer number of neighbors per process. We note that random sparse graphs have also been used by Hoefler and Schneider [43] to evaluate neighborhood collective optimizations.

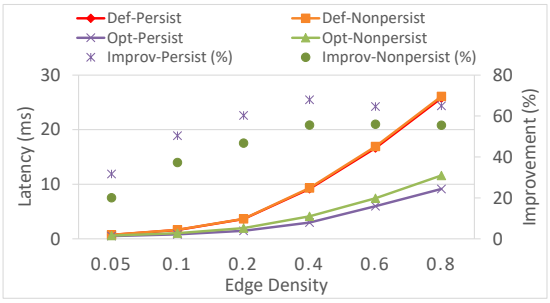
We conduct our experiments for different values of the density factor p that allow us to evaluate our proposed design with neighborhood topologies of random shapes and different sizes. For each specific value of p, we run our microbenchmark 5 times to account for the randomness of the topology graph and report the average.

Figure 6.2 shows the results for neighborhood allgather and neighborhood alltoallv with a 4-byte message size. It shows the absolute latency values (left axis) as well as the improvement percentages (right axis). We can see that our proposed design results in lower latencies compared with those with the naïve approach used in MVAPICH. In general, we see greater improvements for denser graphs as expected, because denser graphs provide a larger number of neighbors per process, which in turn provide more room for improvements through our proposed approach. Fig. 6.2 shows that even at a low edge density of 0.05, we can still achieve about 50% and 30% improvement for neighborhood allgather and neighborhood alltoallv, respectively. According to the figure, the improvements can be as high as 70%. Another observation is that although the persistent feature does not impact the default approach noticeably, it can improve the performance of our design. This is because building a schedule out of the naïve pattern is much easier (with almost no overhead) than building it from the message-combining pattern. Thus, the persistent feature has higher impacts for the latter. However, as we discuss in more detail in Section 6.4.3, the schedule overhead is still low for our design, and hence both the persistent and nonpersistent versions of our design outperform the default approach.

In Fig. 6.3 and Fig. 6.4 we show the results across various message sizes and three edge densities for neighborhood allgather and neighborhood alltoallv, respectively. We can see lower latencies achieved by our proposed design across all message sizes up to 1 KB for both neighborhood allgather and neighborhood alltoallv. The improvements vary from 36% to 71% based on the specific message size and edge density, but in most cases we can achieve about 50% reduction in latency. The improvement gap starts to shrink after 1 KB message size because of bandwidth effects. For clarity, we do not show the results for message sizes

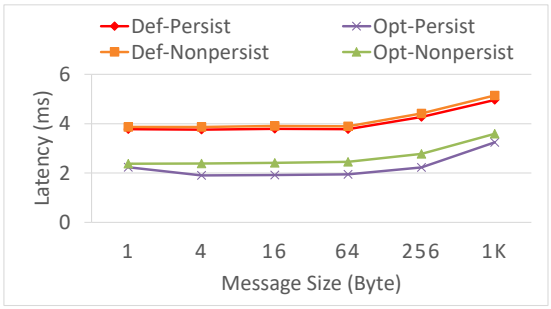


(a) NBR_Allgather—4,096 MPI processes

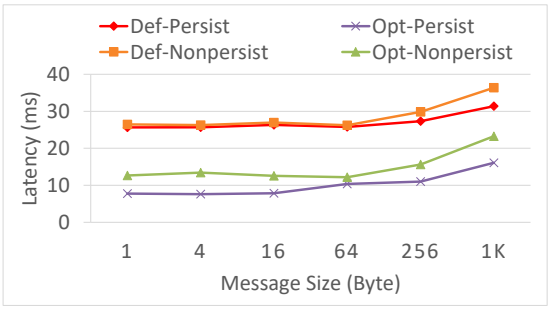


(b) NBR_Alltoallv—1,024 MPI processes

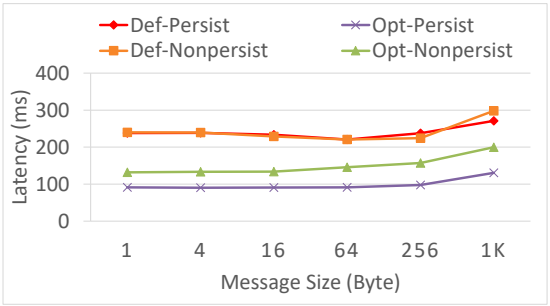
Figure 6.2: Average latencies for the random sparse graph topology—4-byte message size and different edge densities (NBR = Neighborhood).



(a) Edge density $p = 0.05$

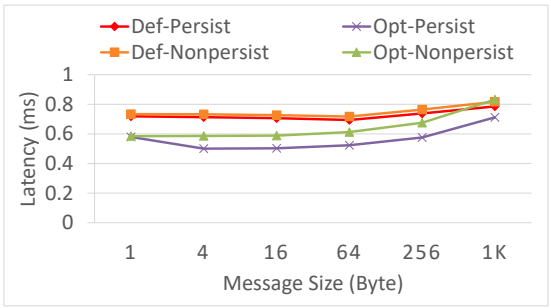


(b) Edge density $p = 0.2$

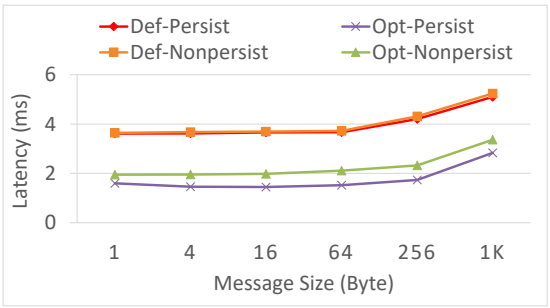


(c) Edge density $p = 0.8$

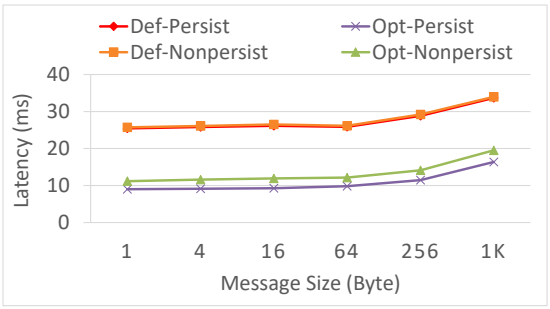
Figure 6.3: Neighborhood allgather average latencies for the random sparse graph topology—4,096 MPI processes, various message sizes and three topology graph edge densities $p = 0.05, 0.2, 0.8$.



(a) Edge density $p = 0.05$



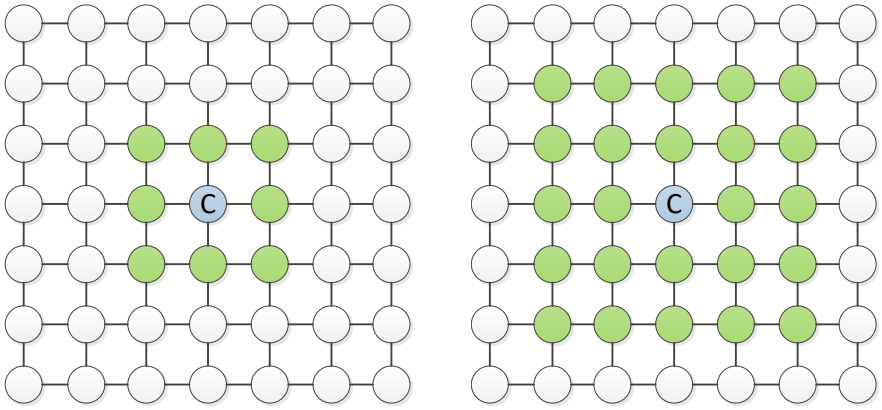
(b) Edge density $p = 0.2$



(c) Edge density $p = 0.8$

Figure 6.4: Neighborhood alltoallv average latencies for the random sparse graph topology—1,024 MPI processes, various message sizes and three topology graph edge densities $p = 0.05, 0.2, 0.8$.

above 1 KB here. Those results show that all four approaches converge more or less to the same latency for message sizes of approximately 4 KB. The exact convergence point varies somewhat for different edge densities and occurs at larger message sizes (near 16 KB) for denser graphs. After that, the naïve approach leads to lower latencies. This result is expected because message combining does not improve the bandwidth terms of the underlying communication pattern. Thus, it does not have much benefit for large messages, which are sensitive mainly to bandwidth/congestion characteristics of communication patterns (recall that our goal from the beginning was to improve the performance for small messages). Another observation is that for our design, the persistent approach provides lower latencies than the nonpersistent approach across all message sizes. Even the nonpersistent version, however, still outperforms the naïve approach. We can also see that the gap between the



(a) $d = 2, r = 1$ (b) $d = 2, r = 2$

Figure 6.5: A sample Moore neighborhood with $d = 2$ and $r = 1, 2$. The neighbors are shown in green for the 'C' node.

persistent and nonpersistent approaches becomes wider as we move from sparse graphs (Fig. 6.3(a)) to denser ones (Fig. 6.3(c)). This is due to the higher number of neighbors and friend processes in denser graphs, which increases the number of operations performed by Alg. 6.3.

6.4.2 Moore Neighborhoods

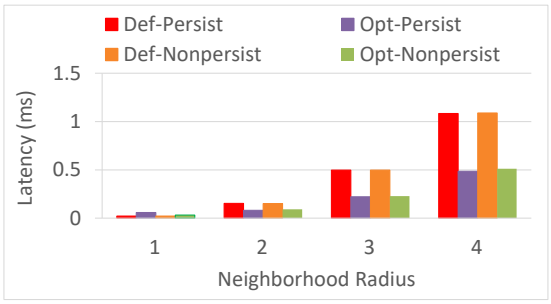
For our second set of experiments, we use a Moore neighborhood to model the process topology graph. A Moore neighborhood is defined with two parameters: dimension (d) and radius (r). The former represents the number of grid dimensions that the nodes (MPI processes) are organized into, and the latter represents the absolute value of the maximum distance at which other nodes are considered a neighbor of a given node. Thus, for each pair of d and r values, the number of neighbors of each node will be equal to $(2r + 1)^d - 1$. Note that Moore neighborhoods are symmetric, in the sense that each neighbor will be both an outgoing and an incoming neighbor of a given MPI process. Fig. 6.5 shows a Moore neighborhood with $d = 2$ and $r = 1, 2$.

Moore neighborhoods allow us to study the benefits of our design with more regular

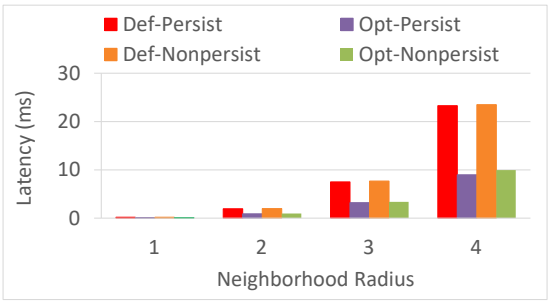
types of topologies. In some sense, they can be considered as a generalization of the stencil patterns such as 2D 5-point or 3D 7-point. However, Moore neighborhoods provide a relatively higher number of neighbors per process. They also allow modeling of a wider variety of neighborhood shapes and sizes through different values of d and r. In addition, simpler patterns such as 2D 5-point and 3D 7-point are more suited to the Cartesian process topology interface of MPI, not the general distributed graph interface that is the focus of our work in this paper. We note that Moore neighborhoods have also been used by Träff et al. [119] as examples of isomorphic neighborhoods in MPI.

We conduct our experiments for different values of d, r. Figure 6.6 and Fig. 6.7 respectively show the results for neighborhood allgather and neighborhood alltoallv with a 4-byte message size. In almost all cases, our approach successfully decreases the latency for different values of d and r. The only exception is the case with $d = 2, r = 1$, where we can see a slight increase in the latency. The are two reasons for this increase. First, in this case, each process has a small number of neighbors (8 to be precise), for which the naïve approach will be good enough. Second, the number of common neighbors between any two processes is at most 4. Considering the additional communication required to exchange messages between any two friend processes, we cannot achieve any considerable decrease in the number of communications with only four common neighbors.

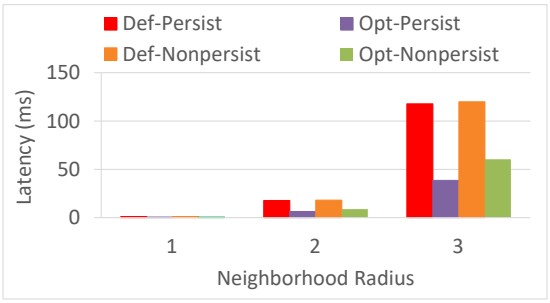
Another observation is that we obtain greater improvements with the increase in either d or r. The reason is that higher values of d and/or r result in a higher number of neighbors per process, providing more opportunity for optimizations through message combining. Moreover, we mostly see similar results for both the persistent and nonpersistent versions of our design for neighborhood allgather, implying a low overhead in Phase 2 of our design. This is confirmed by the results shown in Section 6.4.3. Note that we do not have any results for $d = 4, r = 4$ in Fig. 6.6(c). The reason is that with 8,192 processes and $d = 4$, we will have 8 processes along some of the dimensions, which is not large enough to support a neighborhood radius of 4 without duplicate neighbors. For similar reasons, we do not have



(a) $d = 2,\ 128 \times 64$ MPI processes



(b) $d = 3,\ 32 \times 16 \times 16$ MPI processes

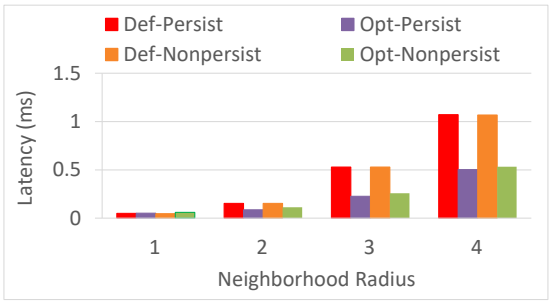


(c) $d = 4,\ 16 \times 8 \times 8 \times 8$ MPI processes

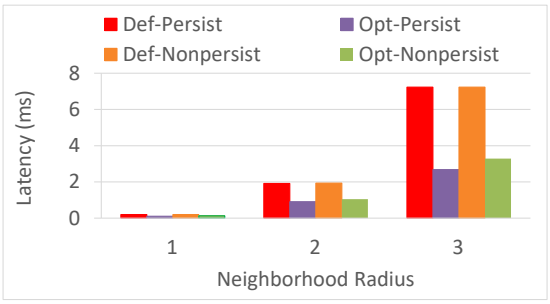
Figure 6.6: Neighborhood allgather average latencies for the Moore neighborhood graph topology—8,192 MPI processes, 4-byte message size, $d = 2, 3, 4$, and $r = 1, 2, 3, 4$.

the results for all combinations of d, r values with the neighborhood alltoallv in Fig. 6.7.

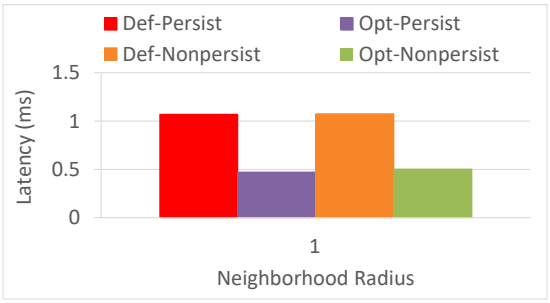
Figure 6.8 and Fig. 6.9 show the results across different message sizes and for three pairs of d, r values. We can see a consistent lower latency achieved by our proposed approach across all message sizes up to 1 KB. The improvements are mostly around 40% for $d = 2, r = 2$ and increase to more than 65% for higher values of d. Similar to the case with the random sparse graph, our results (not presented here) show that all approaches have almost the same performance at 4-KB message size (the exact point depends on d, r), after which the naïve approach outperforms message combining. Also, as shown in Fig. 6.6, for neighborhood allgather we do not see a considerable difference between the persistent and nonpersistent approaches with lower values of d, r. The impacts become more visible starting from $d = 4, r = 2$ as shown by Fig. 6.8(c). For neighborhood alltoallv, we see a



(a) $d = 2,\ 32 \times 32$ MPI processes



(b) $d = 3,\ 16 \times 8 \times 8$ MPI processes



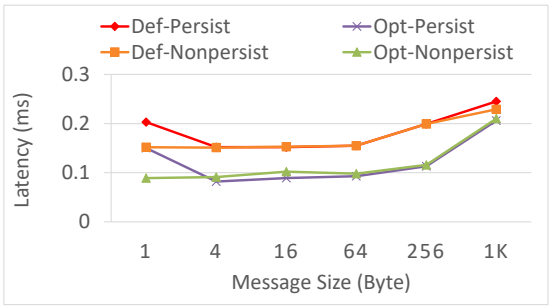
(c) $d = 4,\ 8 \times 8 \times 4 \times 4$ MPI processes

Figure 6.7: Neighborhood alltoallv average latencies for the Moore neighborhood graph topology—1,024 MPI processes, 4-byte message size, $d = 2, 3, 4$, and $r = 1, 2, 3, 4$.

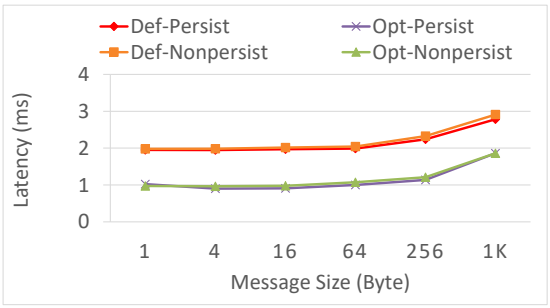
considerably lower latency in Fig. 6.9(a) for the persistent approach due to the additional overheads associated with the alltoallv schedule design.

6.4.3 Overhead Analysis

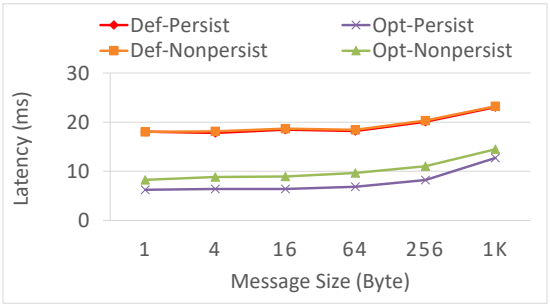
In this section we analyze the overheads associated with our message-combining approach. The overheads can be divided into two main parts: (1) Phase 1 overhead, representing the time spent by Alg. 6.1 to extract the communication pattern and (2) Phase 2 overhead, representing the time spent by Alg. 6.3 to build the communication schedule from the pattern. We note that Phase 1 overheads are encountered only once per given topology, whereas Phase 2 overheads are encountered each time a neighborhood collective function is called. With a persistent approach, however, Phase 2 overheads will be imposed only once for each specific instance of a neighborhood collective.



(a) $d = 2, r = 2$ 128×64 MPI processes



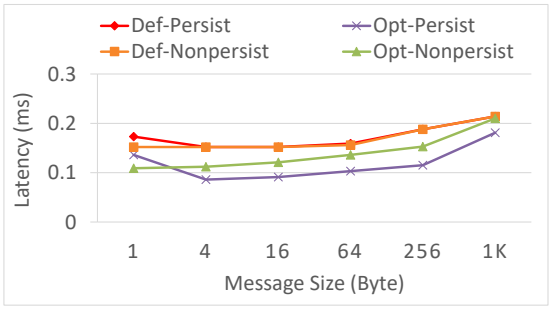
(b) $d = 3, r = 2$ $32 \times 16 \times 16$ MPI processes



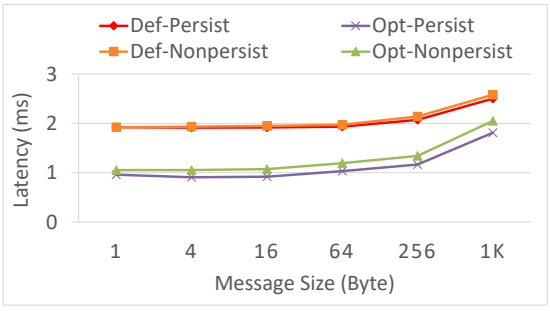
(c) $d = 4, r = 2$ $16 \times 8 \times 8 \times 8$ MPI processes

Figure 6.8: Neighborhood allgather average latencies for the Moore neighborhood graph topology—8,192 MPI processes, various message size, selective d, r values.

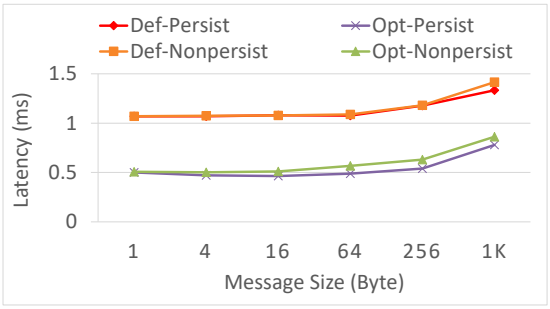
Table 6.1 and Table 6.2 respectively show the time spent at each phase of our design for the sparse random graph and the Moore neighborhood. As we can see, the main overhead belongs to Phase 1, where we build the message-combining pattern (Alg. 6.1). Phase 2 overheads (Alg. 6.3) are four orders of magnitude lower and remain well below 1 second in all cases. Thus, we do not see significant differences between the persistent and nonpersistent flavors of our design in the results shown in Section 6.4.1 and 6.4.2. This is particularly true with the Moore neighborhood for which we observe low Phase 2 overheads in Table 6.2.



(a) $d = 2, r = 2$ 32×32 MPI processes



(b) $d = 3, r = 2$ $16 \times 8 \times 8$ MPI processes



(c) $d = 4, r = 1$ $8 \times 8 \times 4 \times 4$ MPI processes

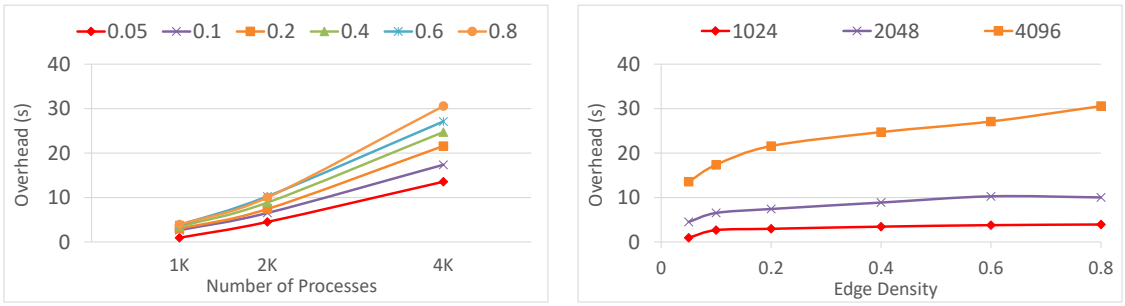
Figure 6.9: Neighborhood alltoallv average latencies for the Moore neighborhood graph topology—1,024 MPI processes, various message size, selective d, r values.

Table 6.1: Random sparse graph. Time spent in Phase 1 and Phase 2—4K processes

Edge Density	$p = 0.05$	$p = 0.1$	$p = 0.2$	$p = 0.4$	$p = 0.8$
Phase 1	13.54 s	17.36 s	21.58 s	24.71 s	30.56 s
Phase 2	0.0005 s	0.001 s	0.005 s	0.019 s	0.07 s

Table 6.2: Moore neighborhood. Time spent in Phase 1 and Phase 2—8K processes

(d, r)	$(2, 2)$	$(2, 3)$	$(2, 4)$
Phase 1	0.31 s	0.62 s	0.96 s
Phase 2	0.000008 s	0.000017 s	0.000047 s



(a) Overhead versus number of processes

(b) Overhead versus edge density

Figure 6.10: Phase 1 overheads for the sparse random graph topology across different number of processes and edge densities.

Next, we evaluate how the overheads are affected by the total number of processes as well as the number of neighbors per process[5]. Figures 6.10 shows the overhead of Phase 1 for the sparse random graph topology across various numbers of processes and edge densities. The overhead increases with the total number of processes as well as edge densities (i.e., number of neighbors). An important observation is that the increase in the total number of processes has a much higher impact on overhead than does the increase in the number of neighbors. For instance, from Fig. 6.10(b) we can see that the overhead with 2K processes and 0.8 edge density ($\approx 1,600$ neighbors) is lower than the overhead with 4K processes and 0.05 edge density (≈ 200 neighbors). In fact, 6.10(a) shows a superlinear increase in the overhead with the total number of processes, whereas we see a sublinear trend with the edge densities in Fig. 6.10(b).

This behavior has its root in the fact that the overhead of Phase 1 is dominated by the point-to-point communications used in Alg. 6.2 to notify the friend processes (lines 24 and 28 of Alg. 6.2). On the other hand, our experimental results show that for the random sparse graph topology, the number of iterations of Alg. 6.1 increases with the square root of f ($t \propto \sqrt{f}$), which leads to an overhead complexity of $\mathcal{O}(\sqrt{f}f) = \mathcal{O}(f^{1.5})$ for Phase 1.

Moreover, Fig. 6.11 shows how the number of friends per process increases in the sparse

[5]We discuss the trends for Phase 1 only as Phase 2 overheads are negligible compared to it.

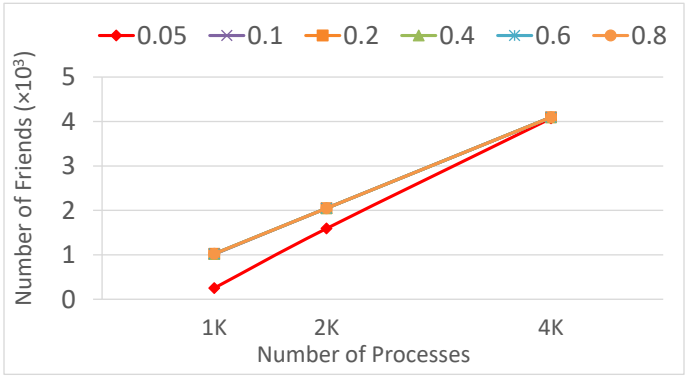
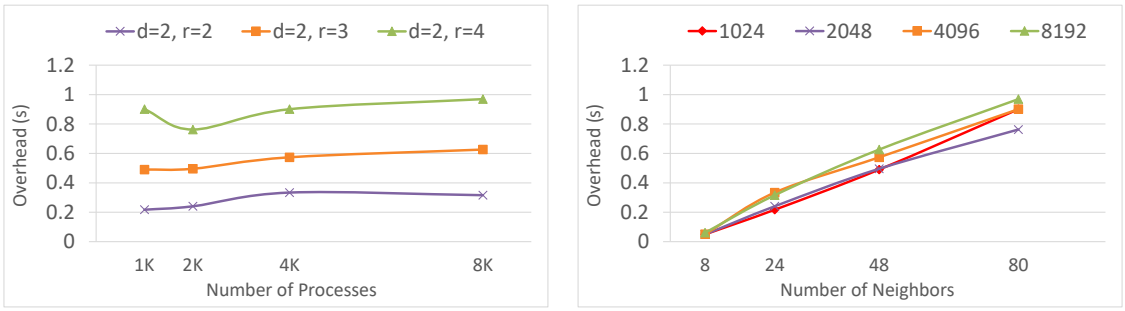


Figure 6.11: Increase in the number of friends per process in the random sparse graph topology ($\Theta = 4$).

random graph topology. We can see that it increases linearly with the number of processes but remains unchanged with an increase in the edge densities. The reason is that in the random sparse graph topology (and $\Theta = 4$), each process tends to be a friend with almost every other process, even at low edge densities. In fact, starting from 0.1 edge density, each process becomes a friend with all other processes. Thus, Phase 1 overhead increases with $\mathcal{O}(p^{1.5})$ for the random sparse graph topology.

Figure 6.12 shows the overhead trends of Phase 1 for the Moore neighborhood topology. An important observation is that we no longer see a superlinear increase in the overhead. In fact, the overhead remains mostly unchanged as we increase the number of processes (Fig. 6.12(a)), and it increases linearly only with the number of neighbors (Fig. 6.12(b)). The reason is that Moore neighborhoods model a more structured and localized neighborhood topology compared with that of random sparse graphs. Therefore, as shown by Fig. 6.13, the number of friends per process is not affected by the increase in the total number of processes. It increases linearly only with the number of neighbors per process. In addition, our experimental results show that for the Moore neighborhood, the number of iterations of Alg. 6.1 is independent of f. Moreover, the number of neighbors per process (n) is independent of the total number of processes, resulting in an overhead complexity of $\mathcal{O}(n)$.



(a) Overhead versus number of processes (b) Overhead versus neighborhood radius

Figure 6.12: Phase 1 overheads for the Moore neighborhood topology across different number of processes and neighbors.

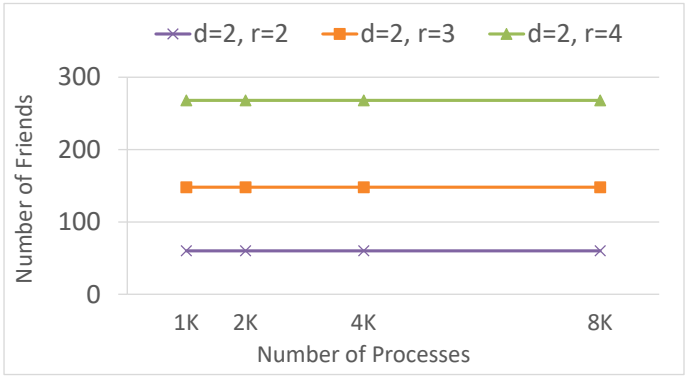


Figure 6.13: Increase in the number of friends per process in the Moore neighborhood topology ($\Theta = 4$).

This is quite encouraging as many real applications tend to use such structured and localized neighborhood topologies.

6.5 Related Work

Hoefler et al. [42] discuss enhancements to the process topology interface of MPI, and in particular the distributed graph topology interface that significantly improves the scalability, informativeness, and user friendliness of the older topology interface. Ovcharenko et al. [94] highlight the existence of sparse neighbor communications in many parallel applications and

propose a communication package on top of MPI to support them. Hoefler and Träff [46] discuss the importance of having sparse collective operations in MPI and propose three such operations. The proposed operations form the basis for what was ultimately added to MPI as the neighborhood collectives. In another work, Hoefler et al. [41] show the benefits of using sparse collective operations for quantum mechanical simulations. Kandalla et al. [58] use nonblocking neighborhood collectives to redesign the BFS algorithm in the Combinatorial BLAS [17] library. They use neighborhood collectives as a building block to compose conventional (global) collectives in order to achieve better communication-computation overlap.

Kumar et al. [65] show how the multisend interface of the IBM Deep Computing Message Framework (DCMF) [64] can be used to optimize applications that exhibit a neighborhood collective communication pattern. The multisend interface allows multiple send operations to be encapsulated into a single call by providing a list of destinations to the direct memory access engine. Unlike our design, this approach does not attempt to derive an optimized communication pattern; each node still sends a message to each of its neighbors directly. Our optimization is orthogonal to the multisend interface and can exploit its corresponding benefits to further optimize the performance of the derived pattern.

Hoefler and Schneider [43] discuss a number of optimization principles for neighborhood collectives. They show how graph coloring can be used to design a communication schedule that avoids creation of hotspots at the end nodes. They also propose an algorithm for balancing the communications from high-outdegree processes with those having lower out-degrees. These optimizations are orthogonal to our proposed design. Moreover, balancing the communication tree is beneficial for unbalanced neighborhood topologies only, whereas our proposed optimization applies to both balanced and unbalanced neighborhoods. This is important because many real applications use balanced neighborhood topologies.

Träff et al. [120] discuss *isomorphic* neighborhoods in which all the processes have the same neighborhood structure. They propose new interfaces for defining such neighborhoods

in MPI that impose lower overheads compared with those of the generic graph interface. In another work [119], Träff et al. propose message-combining algorithms for isomorphic neighborhoods and show the benefits for improving the performance of neighborhood collectives. However, the proposed algorithms are limited to isomorphic neighborhoods that are defined over Cartesian topologies. Furthermore, they depend on new interfaces that are not yet available in MPI. We design our proposed approach over the distributed graph topology interface of MPI, which provides the most generic and flexible way for defining neighborhood topologies.

6.6 Summary

Neighborhood collectives were added to MPI to provide better support for the sparse neighbor communication patterns used in many parallel applications. They also address some of the main shortcomings of the more traditional collective communication in MPI. In particular, neighborhood collectives enable programmers to define their own collective communication patterns through the process topology interface of MPI. They also promote better scalability by avoiding global communication patterns.

Accordingly, in this chapter we focused on the performance optimization of neighborhood collectives. We discussed how the process topology information associated with a neighborhood collective can be used to design nontrivial communication schedules for various neighborhood collective operations. More specifically, we explained how common neighborhoods among the processes can be exploited to improve the performance of neighborhood collectives. We proposed a fully distributed algorithm to find the common neighborhoods in an MPI distributed graph topology and exploit them to build a message-combining communication pattern and schedule for neighborhood collectives. The core idea was to decrease the number of individual communications that are required to send/receive data to/from neighbors. We achieved this by evenly dividing the common neighbors of two processes into

two disjoint sets, and assigning each set to only one of the two processes. Our experimental results showed that using our optimized schedules, we can achieve considerable reduction in the latency of neighbor allgather and neighbor alltoallv for neighborhood topologies of various shapes and sizes.

Chapter 7

Conclusions and Future Work

7.1 Conclusion

Communication continues to be the performance bottleneck in HPC systems and hence, communication optimization represents one of the most important paradigms for increasing the overall performance of HPC systems. This is particularly important when we consider the fast pace at which the number of processors within large-scale systems continues to grow. In this dissertation we studied the benefits of topology awareness for improving communication performance in HPC systems. We presented new algorithms and approaches for topology-aware assignment of the processes of a parallel application onto the processing elements of a target system. The proposed algorithms improve the communication performance by providing a better match between the communication characteristics of the application and the heterogeneous set of communication channels of the target system. In the remainder of this section, we point out the highlights of the research described in this dissertation.

In Chapter 3, we proposed a new greedy heuristic for topology-aware mapping of processes. We put our main focus on decreasing the amount of congestion that is imposed on each link within the network. To this end, we incorporated two main features in our heuristic design: (1) routing information, and (2) a hybrid metric. For each candidate

mapping, the routing information allowed us to have a precise evaluation of the traffic load that would pass across each link of the network. The hybrid metric enabled us to evaluate each candidate mapping from different aspects and hence, make better decisions as we gradually build the desired mapping through each iteration of the greedy heuristic. Our experimental results on an InfiniBand cluster showed that we can achieve considerable improvements in congestion by using our proposed heuristic. Another highlight is the parallel design of our proposed algorithms. The parallel design allowed us to explore a wider scope of candidate mappings from the entire search space and at the same time achieve a lower overhead. In this regard, we consider our design as a first attempt towards building parallel topology-aware mapping algorithms for large-scale HPC systems.

In Chapter 4, we showed how topology-aware mapping can be used to achieve the potential benefits of various collective communication algorithms under different mappings of processes. We proposed topology-aware mapping heuristics that were specifically designed to improve the performance of certain collective communication patterns. The main highlight of the chapter is that we can exploit the knowledge of the collective communication algorithms at the level of the MPI library to design low-overhead mapping heuristics that are fine-tuned for each specific collective communication pattern.

In Chapter 5, we showed that topology awareness is also important in the GPU communications domain. Using GPUs to accelerate certain computations of an application is becoming an intrinsic part of parallel computing. Consequently, the performance of GPU communications will play a key role in the overall performance delivered to users. We showed that the physical topology of the GPU communication channels is a determining factor in GPU communications performance. Accordingly, we discussed how topology-aware mapping can be used in a heterogeneous GPU cluster to jointly improve the performance of both CPU and GPU communications. We proposed a unified framework for topology-aware process-to-core mapping and GPU-to-process assignment, and showed its benefits through experimental evaluations on two multicore multi-GPU clusters.

In Chapter 6, we focused on performance optimization of neighborhood collective communications in MPI. Neighborhood collectives represent a relatively new type of communication in MPI that greatly increase the importance and employment of MPI process topologies. We showed that useful information can be extracted from a given process topology to optimize the performance of neighborhood collectives. In particular, we explained how the existence of common neighborhoods in the process topology graph can be used as an opportunity to improve the performance of neighborhood communications. We showed how we can design an efficient distributed algorithm to find the common neighborhoods in a generic MPI distributed graph topology and use it to build message-combining communication schedules for neighborhood collectives. Through experimental evaluations we showed that such optimized schedules can successfully decrease the latency of neighborhood communications.

7.2 Future Work

Parallel Topology- and Routing-Aware Process Mapping

We intend to study more sophisticated combinations of the individual metrics for building the hybrid metric. In our current design, we used a simple linear combination with equal weights for each individual metric. Further analytical and experimental studies could help to determine an optimized combination of the metrics. This can be done with respect to the performance characteristics of each application and system. For instance, one could assign a higher weight to the distance-based metrics (hop-bytes) for applications whose communications are dominated by small messages. On the other hand, for applications that communicate many large messages, a higher weight can be assigned to the congestion-based metrics. Discovery of other metrics and adding them to the hybrid metric is another direction of future work.

In addition, we intend to improve the scalability of communications among node-leaders

by making such communications hierarchical. Node-leaders could be clustered into multiple disjoint groups. All members within a group communicate with each other, whereas inter-group communications are limited to group-leaders only. Next, we would like to improve the performance of our heuristics further by parallelizing the computations performed by each node-leader across all the cores within each node. We also seek to extend our approach to also cover the mapping at the intra-node layer and eliminate the initial graph partitioning stage.

Future work can also extend our proposed approach to systems with other types of high-performance interconnects. In general, our proposed heuristics can be used with any system that provides some APIs for querying information about the underlying routing algorithm and the precise layout of the network links. In particular, we are interested in extending our approach to systems that use multilevel direct network topologies such as Dragonfly. Extending our approach towards systems with adaptive routing is another direction for future research. One way to use PTRAM in the presence of adaptive routing is to use an approximation of the routing algorithm to represent the routing information. For instance, we can use the default shortest paths that are commonly used in adaptive routing algorithms. In this case, our approach will attempt to map the processes in such a way that congestion on the default shortest paths used by the underlying adaptive algorithm is minimized. This can potentially decrease the need for using alternative paths (that might not be optimal) by the adaptive routing.

Another direction for future work is to add the temporal characteristics of communication patterns to the mapping heuristics. Current representations of process topologies provide only a bulk representation of message exchanges among each pair of communicating processes. They do not provide any knowledge about how such communications are distributed throughout the lifetime of an application. Knowledge about the temporal properties of communications will allow us to have a more realistic measure of runtime

congestion. To this end, we must develop an efficient mechanism for extraction and representation of the temporal characteristics of communication patterns. One possible solution is to partition the runtime of an application into several time slots t_i each with a length equal to δ_i. The time slots can then be used as a set of *temporal tags* to label the edges in the process topology graph. This can be done at the same profiling stage used to gather the communication pattern of an application. Having labeled the edges, the mapping heuristic will increase the congestion value of the network links only if the overlapping flows belong to the same time slot, i.e., have the same temporal tag in the process topology graph. This way, we will have a more realistic congestion value for each link at every iteration of the mapping heuristic.

Topology-Aware Mapping Heuristics for Collective Communications

With respect to topology-aware collective communications, we first plan to extend the set of our heuristics to other allgather algorithms such as the Bruck algorithm [116], as well as the algorithms used for other collective communications such as alltoall and allreduce. We also seek to evaluate the performance of our heuristics on systems that have a more complicated intra-node topology with a larger number of cores per node. Another interesting direction for future work is to devise an adaptive version of our proposed approach where a runtime component is used to decide whether to use the reordered communicator for a given collective or not. This can be done based on the potential performance improvements that each heuristic can provide for various message sizes and/or collective communication algorithms.

Another direction for future work is to use the physical topology and the mapping of processes to modify the collective communication algorithm so as to minimize the congestion that is imposed across the different channels. We can build a tree representation of the given MPI communicator by recursively splitting the set of processes into two disjoint partitions. Using the knowledge of the physical topology and the specific placement of the processes,

the partitioning is done such that communication between two processes in one partition will not interfere with communication between two processes in another partition. Then, for a broadcast operation as an example, we can build the desired communication pattern by traversing the tree in a top-down manner and schedule a message transmission between the sibling partitions in the tree.

Topology-Aware Communications in Heterogeneous GPU Clusters

We intend to evaluate MAGC with other HPC applications designed to exploit hybrid CPU-GPU systems. We are specifically interested in evaluating the effects of MAGC on applications that exhibit irregular communication patterns with heavy communications among the CPU cores as well as among the GPU devices. We speculate that these types of applications will benefit the most from MAGC. We also intend to conduct our experiments at larger scales with systems consisting of a larger number of multi-GPU nodes. Another direction of interest is to extend MAGC to newer GPU interconnects such as NVLink [89]. Future work can also consider the computation load of the GPUs as an additional metric for finding optimized GPU assignments. This will help to balance the kernels that are meant to be offloaded for GPU acceleration across all the available GPUs.

Providing support for small-message communications is another important direction for future research. As shown by the experimental results in Chapter 5, our current design is mainly beneficial for larger messages. The reason is that we put our focus on the communication volume and bandwidth. For small message sizes, we need to design mapping algorithms that are tuned based on the zero-byte latency characteristics of the underlying physical topology. In addition, the heuristics should be based more on the frequency of communications rather than the volume. Thus, we believe an ideal mapping heuristic should have flexibility to use different criteria with respect to the size of the communicated messages. For instance, if an application is bound by the performance of small messages, the mapping should be optimized based on the distance and frequency of communications;

otherwise, congestion and communication volume should be used to find an optimized mapping.

Neighborhood Collective Communications Optimization

We first plan to evaluate the benefits of our design for real applications. A major challenge is that we must port current applications so as to change their point-to-point neighbor communications with appropriate calls to neighborhood collectives. This is because neighborhood collectives are a relatively new feature added to MPI and hence they are not yet used in HPC applications. Moreover, we intend to extend our message-combining algorithm design with *cumulative combining*. In our current design, each process considers any of its outgoing neighbors in at most one message-combining round. The reason is that at the end of each iteration of Alg. 6.1, we ignore both the offloaded and onloaded neighbors of the processes that successfully paired with a friend process. In cumulative combining, however, we will only ignore the offloaded neighbors of each process. Thus, each process reconsiders the set of its onloaded neighbors for further message combining opportunities with other friend processes. Such a nested combining will enable us to further decrease the number of individual communications that are performed by each process.

Another direction for future work is to consider a group of processes for message combining in each iteration of our proposed algorithm. Currently, we only exploit the common neighborhoods between two friend processes at each message-combining round. We can extend this by attempting to find the largest common neighborhood among k friend processes. A larger value of k will allow us to divide the set of common neighbors among a larger number of friend processes, which in turn leaves each of them with fewer neighbors to communicate with. However, larger values of k will also increase the number of message exchanges required among the friend processes. Therefore, an interesting research direction is to find the optimal value of k based on the common neighborhood size and the number of friend processes involved. We also note that while the pairwise combining relates to a

maximum weighted matching problem in the friendship graph, the k-ary extension will fall into the scope of finding maximum cliques in a distributed graph.

Finding the best value for the friendship threshold Θ is another direction for future work. Higher values of Θ lead to a lower number of friends per process which will decrease the communication overheads of Alg. 6.2. However, it will also lower the opportunities for message combining. Thus, it provides a trade-off between the quality and overhead of the algorithm. A metric that can be used to guide the appropriate selection of the Θ value is the density and structure of the topology graph. For instance, it might be better to use a higher value of Θ for denser topology graphs. The reason is that a dense topology graph will result in a dense friendship graph where each node will have a high chance of being matched along a heavy-weight edge. Therefore, we can safely ignore the friends that correspond to smaller-size common neighborhoods.

Future work can also improve the overheads of our design. In particular, we can update Alg. 6.2 to exploit the remote memory access (RMA) operations of MPI. This feature could help avoid a large fraction of the point-to-point communications that are performed between a process and its friends. We are also interested in evaluating performance and overhead of our design when it is used with other distributed maximum matching algorithms. MPI datatypes and/or interconnect features can also be exploited to decrease overheads through design and implementation of zero-copy schedules.

Bibliography

[1] A. H. Abdel-Gawad, M. Thottethodi, and A. Bhatelé. RAHTM: routing algorithm aware hierarchical task mapping. In *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 325–335, 2014.

[2] G. Almási, P. Heidelberger, C. J. Archer, X. Martorell, C. C. Erway, J. E. . Moreira, B. Steinmacher-Burow, and Y. Zheng. Optimization of MPI collective communication on BlueGene/L systems. In *Proc. International Conference on Supercomputing (ICS)*, pages 253–262, 2005.

[3] B. Alverson, E. Froese, L. Kaplan, and D. Roweth. Cray XC series network. Cray Inc., White Paper WP-Aries01-1112. 2012.

[4] An introduction to the Intel Quickpath Interconnect. White Paper, 2009.

[5] J. A. Anderson, C. D. Lorenz, and A. Travesset. General purpose molecular dynamics simulations fully implemented on graphics processing units. *Journal of Computational Physics*, 227(10):5342–5359, 2008.

[6] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, T. Hoefler, S. Kumar, E. Lusk, R. Thakur, and J. L. Träff. MPI on millions of cores. *Parallel Processing Letters*, 21(01):45–60, 2011.

[7] P. Balaji, R. Gupta, A. Vishnu, and P. Beckman. Mapping communication layouts to network hardware characteristics on massive-scale blue gene systems. *Computer Science-Research and Development*, 26(3-4):247–256, 2011.

[8] R. F. Barrett, C. T. Vaughan, and M. A. Heroux. MiniGhost: a miniapp for exploring boundary exchange strategies using stencil computations in scientific parallel computing. Sandia National Laboratories, Tech. Rep. SAND5294832. 2011.

[9] G. Berti and J. L. Träff. What MPI could (and cannot) do for mesh-partitioning on non-homogeneous networks. In *Proc. European PVM/MPI User's Group Conference on Recent Advances in Parallel Virtual Machine and Message Passing Interface (EuroPVM/MPI)*, pages 293–302, 2006.

[10] A. Bhatelé, T. Gamblin, S. H. Langer, P.-T. Bremer, E. W. Draeger, B. Hamann, K. E. Isaacs, A. G. Landge, J. A. Levine, V. Pascucci, M. Schulz, and C. H. Still. Mapping

applications with collectives over sub-communicators on torus networks. In *Proc. International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 97:1–97:11, 2012.

[11] A. Bhatelé, G. R. Gupta, L. V. Kalé, and I.-H. Chung. Automated mapping of regular communication graphs on mesh interconnects. In *Proc. International Conference on High Performance Computing (HiPC)*, pages 1–10, 2010.

[12] A. Bhatelé, N. Jain, K. E. Isaacs, R. Buch, T. Gamblin, S. H. Langer, and L. V. Kalé. Optimizing the performance of parallel applications on a 5D torus via task mapping. In *International Conference on High Performance Computing (HiPC)*, pages 1–10, 2014.

[13] A. Bhatelé, N. Jain, Y. Livnat, V. Pascucci, and P.-T. Bremer. Analyzing network health and congestion in Dragonfly-based supercomputers. In *Proc. International Parallel and Distributed Processing Symposium (IPDPS)*, pages 93–102, 2016.

[14] A. Bhatelé and L. V. Kalé. An evaluative study on the effect of contention on message latencies in large supercomputers. In *International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–8, 2009.

[15] A. Bhatelé and L. V. Kalé. Heuristic-based techniques for mapping irregular communication graphs to mesh topologies. In *International Conference on High Performance Computing and Communications (HPCC)*, pages 765–771, 2011.

[16] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. hwloc: A generic framework for managing hardware affinities in HPC applications. In *Proc. Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 180–186, 2010.

[17] A. Buluç and J. R. Gilbert. The combinatorial BLAS: Design, implementation, and applications. *The International Journal of High Performance Computing Applications*, 25(4):496–509, 2011.

[18] A. Castro, H. Appel, M. Oliveira, C. A. Rozzi, X. Andrade, F. Lorenzen, M. A. L. Marques, E. K. U. Gross, and A. Rubio. octopus: a tool for the application of time-dependent density functional theory. *Physica Status Solidi (b)*, 243(11):2465–2488, 2006.

[19] E. Chan, R. van de Geijn, W. Gropp, and R. Thakur. Collective communication on architectures that support simultaneous communication over multiple links. In *Proc. Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 2–11. ACM, 2006.

[20] I.-H. Chung, C.-R. Lee, J. Zhou, and Y.-C. Chung. Hierarchical mapping for HPC applications. In *Proc. International Parallel and Distributed Processing Workshops (IPDPSW)*, pages 1815–1823, 2011.

[21] N. Dandapanthula, H. Subramoni, J. Vienne, K. Kandalla, S. Sur, D. K. Panda, and R. Brightwell. INAM—a scalable InfiniBand network analysis and monitoring tool. In *Proc. European Conference on Parallel Processing (Euro-Par)*, pages 166–177, 2011.

[22] M. Deveci, K. Kaya, B. Ucar, and U. V. Catalyurek. Fast and high quality topology-aware task mapping. In *Proc. International Parallel and Distributed Processing Symposium (IPDPS)*, pages 197–206, 2015.

[23] M. Deveci, S. Rajamanickam, V. J. Leung, K. Pedretti, S. L. Olivier, D. P. Bunde, U. V. Catalyurek, and K. Devine. Exploiting geometric partitioning in task mapping for parallel computers. In *Proc. International Parallel and Distributed Processing Symposium (IPDPS)*, pages 27–36, 2014.

[24] J. Dinan, P. Balaji, E. Lusk, P. Sadayappan, and R. Thakur. Hybrid parallel programming with MPI and unified parallel C. In *Proc. International Conference on Computing Frontiers*, pages 177–186, 2010.

[25] J. Duato, A. J. Pena, F. Silla, R. Mayo, and E. S. Quintana-Ortí. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *Proc. International Conference on High Performance Computing and Simulation (HPCS)*, pages 224–231, 2010.

[26] G. Faanes, A. Bataineh, D. Roweth, E. Froese, B. Alverson, T. Johnson, J. Kopnick, M. Higgins, and J. Reinhard. Cray cascade: a scalable HPC system based on a Dragonfly network. In *Proc. International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 103:1–103:9, 2012.

[27] A. Faraj, S. Kumar, B. Smith, A. Mamidala, J. Gunnels, and P. Heidelberger. MPI collective communications on the Blue Gene/P supercomputer: algorithms and optimizations. In *Proc. International Conference on Supercomputing (ICS)*, pages 489–490, 2009.

[28] A. Faraj, P. Patarasuk, and X. Yuan. Bandwidth efficient all-to-all broadcast on switched clusters. *International Journal of Parallel Programming*, 36(4):426–453, 2008.

[29] A. Faraj and X. Yuan. Automatic generation and tuning of MPI collective communication routines. In *Proc. International Conference on Supercomputing (ICS)*, pages 393–402, 2005.

[30] I. Faraji and A. Afsahi. GPU-aware intranode MPI allreduce. In *Proc. European MPI Users' Group Meeting (EuroMPI/ASIA)*, pages 45–50, 2014.

[31] I. Faraji, S. H. Mirsadeghi, and A. Afsahi. Topology-aware GPU selection on multi-GPU nodes. In *Proc. International Parallel and Distributed Processing Symposium Workshops (IPDPSW/AsHES)*, pages 712–720, 2016.

[32] I. Faraji, S. H. Mirsadeghi, and A. Afsahi. Exploiting heterogeneity of communication channels for efficient GPU selection on multi-GPU nodes. *Accepted for publication in the Journal of Parallel Computing*, 2017.

[33] A. Friedley, T. Hoefler, G. Bronevetsky, A. Lumsdaine, and C.-C. Ma. Ownership passing: efficient distributed memory programming on multi-core systems. In *Proc. Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 177–186, 2013.

[34] O. Fuhrer, C. Osuna, X. Lapillonne, T. Gysi, B. Cumming, M. Bianco, A. Arteaga, and T. C. Schulthess. Towards a performance portable, architecture agnostic implementation strategy for weather and climate models. *Supercomputing Frontiers and Innovations*, 1(1):45–62, 2014.

[35] Gadget-3 application code, http://http://www.prace-ri.eu/ueabs/#GADGET, last accessed 2017/05/30.

[36] J. Glaser, T. D. Nguyen, J. A. Anderson, P. Lui, F. Spiga, J. A. Millan, D. C. Morse, and S. C. Glotzer. Strong scaling of general-purpose molecular dynamics simulations on GPUs. *Journal of Computational Physics*, 192:97–107, 2015.

[37] R. L. Graham, S. Poole, P. Shamis, G. Bloch, N. Bloch, H. Chapman, M. Kagan, A. Shahar, I. Rabinovitz, and G. Shainer. Overlapping computation and communication: Barrier algorithms and ConnectX-2 CORE-Direct capabilities. In *International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1–8. IEEE, 2010.

[38] P. Grun. Introduction to InfiniBand for end users. White paper, InfiniBand Trade Association. 2010.

[39] F. Gygi. Large-scale first-principles molecular dynamics: moving from terascale to petascale computing. *Journal of Physics: Conference Series*, 46(1):268–277, 2006.

[40] R. W. Hockney. The communication challenge for MPP: Intel Paragon and Meiko CS-2. *Parallel computing*, 20(3):389–398, 1994.

[41] T. Hoefler, F. Lorenzen, and A. Lumsdaine. Sparse non-blocking collectives in quantum mechanical calculations. In *Proc. European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting (EuroPVM/MPI)*, pages 55–63, 2008.

[42] T. Hoefler, R. Rabenseifner, H. Ritzdorf, B. R. de Supinski, R. Thakur, and J. L. Träff. The scalable process topology interface of MPI 2.2. *Concurrency and Computation: Practice and Experience*, 23(4):293–310, 2011.

[43] T. Hoefler and T. Schneider. Optimization principles for collective neighborhood communications. In *Proc. International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 98:1–98:10, 2012.

[44] T. Hoefler, T. Schneider, and A. Lumsdaine. Multistage switches are not crossbars: Effects of static routing in high-performance networks. In *Proc. International Conference on Cluster Computing*, pages 116–125, 2008.

[45] T. Hoefler and M. Snir. Generic topology mapping strategies for large-scale parallel architectures. In *Proc. International Conference on Supercomputing (ICS)*, pages 75–84, 2011.

[46] T. Hoefler and J. L. Traff. Sparse collective operations for MPI. In *Proc. International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–8, 2009.

[47] J.-H. Hoepman. Simple distributed weighted matchings. arXiv preprint cs/0410047. 2004.

[48] The Hydra process management framework, http://wiki.mpich.org/mpich/index.php, last accessed 2017/05/30.

[49] HyperTransport Technology Consortium, http://www.hypertransport.org/, last accessed 2017/05/30.

[50] C. U. Ileri and O. Dagdeviren. Performance evaluation of distributed maximum weighted matching algorithms. In *Proc. International Conference on Digital Information and Communication Technology and its Applications (DICTAP)*, pages 103–108, 2016.

[51] InfiniBand Trade Association. InfiniBand Architecture Specification, http://www.infinibandta.org/, last accessed 2017/05/30.

[52] S. Ito, K. Goto, and K. Ono. Automatically optimized core mapping to subdomains of domain decomposition method on multicore parallel environments. *Computers & Fluids*, 80(0):88–93, 2013.

[53] N. Jain, A. Bhatelé, X. Ni, N. J. Wright, and L. V. Kalé. Maximizing throughput on a Dragonfly network. In *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 336–347, 2014.

[54] N. Jain, A. Bhatelé, M. P. Robson, T. Gamblin, and L. V. Kalé. Predicting application performance using supervised learning on communication features. In *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2013.

[55] E. Jeannot and G. Mercier. Near-optimal placement of MPI processes on hierarchical NUMA architectures. In *Proc. European Conference on Parallel Processing (Euro-Par): Part II*, 2010.

[56] E. Jeannot, G. Mercier, and F. Tessier. Process placement in multicore clusters: Algorithmic issues and practical techniques. *IEEE Tran. Parallel and Distributed Systems*, 25(4):993–1002, 2013.

[57] F. Ji, A. M. Aji, J. Dinan, D. Buntinas, P. Balaji, R. Thakur, W.-c. Feng, and X. Ma. DMA-assisted, intranode communication in GPU accelerated systems. In *Proc. International Conference on High Performance Computing and Communication & International Conference on Embedded Software and Systems (HPCC-ICESS)*, pages 461–468, 2012.

[58] K. Kandalla, A. Buluç, H. Subramoni, K. Tomko, J. Vienne, L. Oliker, and D. K. Panda. Can network-offload based non-blocking neighborhood MPI collectives improve communication overheads of irregular graph algorithms? In *Proc. International Conference on Cluster Computing Workshops*, pages 222–230, 2012.

[59] K. Kandalla, H. Subramoni, A. Vishnu, and D. K. Panda. Designing topology-aware collective communication algorithms for large scale InfiniBand clusters: Case studies with scatter and gather. In *Proc. International Parallel and Distributed Processing Workshops Symposium (IPDPSW)*, pages 1–8, 2010.

[60] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[61] G. Karypis and V. Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *Journal of Parallel and Distributed Computing*, 48(1):71–95, 1998.

[62] J. Kim, W. J. Dally, S. Scott, and D. Abts. Technology-driven, highly-scalable Dragonfly topology. *Computer Architecture News*, 36(3):77–88, 2008.

[63] C. Koufogiannakis and N. E. Young. Distributed algorithms for covering, packing and maximum weighted matching. *Distributed Computing*, 24(1):45–63, 2011.

[64] S. Kumar, G. Dozsa, G. Almasi, P. Heidelberger, D. Chen, M. E. Giampapa, M. Blocksome, A. Faraj, J. Parker, J. Ratterman, B. Smith, and C. J. Archer. The Deep Computing Messaging Framework: Generalized scalable message passing on the Blue Gene/P supercomputer. In *Proc. International Conference on Supercomputing (ICS)*, pages 94–103, 2008.

[65] S. Kumar, P. Heidelberger, D. Chen, and M. Hines. Optimization of applications with non-blocking neighborhood collectives via multisends on the Blue Gene/P supercomputer. In *Proc. International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–11, 2010.

[66] S. Kumar, A. Mamidala, P. Heidelberger, D. Chen, and D. Faraj. Optimization of MPI collective operations on the IBM Blue Gene/Q supercomputer. *The International Journal of High Performance Computing Applications*, 28(4):450–464, 2014.

[67] S. Kumar, S. S. Sharkawi, and K. A. N. Jan. Optimization and analysis of MPI collective communication on fat-tree networks. In *Proc. International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1031–1040, 2016.

[68] X. Lapillonne and O. Fuhrer. Using compiler directives to port large scientific applications to GPUs: An example from atmospheric science. *Parallel Processing Letters*, 24(01):1450003:1–1450003:18, 2014.

[69] D. Li, Y. Wang, and W. Zhu. Topology-aware process mapping on clusters featuring NUMA and hierarchical network. In *International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 74–81, 2013.

[70] S. Li, T. Hoefler, and M. Snir. NUMA-aware shared-memory collective communication for MPI. In *Proc. International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, pages 85–96, 2013.

[71] E. M. Loiola, N. M. M. de Abreu, P. O. Boaventura-Netto, P. Hahn, and T. Querido. A survey for the quadratic assignment problem. *European Journal of Operational Research*, 176(2):657–690, 2007.

[72] C. Loken, D. Gruner, L. Groer, R. Peltier, N. Bunn, M. Craig, T. Henriques, J. Dempsey, C.-H. Yu, J. Chen, L. J. Dursi, J. Chong, S. Northrup, J. Pinto, N. Knecht, and R. Van Zon. Scinet: Lessons learned from building a power-efficient top-20 system and data centre. *Journal of Physics: Conference Series*, 256(1), 2010.

[73] Z. Lotker, B. Patt-Shamir, and S. Pettie. Improved distributed approximate matching. In *Proc. Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 129–136. ACM, 2008.

[74] Z. Lotker, B. Patt-Shamir, and S. Pettie. Improved distributed approximate matching. *Journal of the ACM*, 62(5):38:1–38:17, 2015.

[75] M. Luo, D. K. Panda, K. Z. Ibrahim, and C. Iancu. Congestion avoidance on manycore high performance computing systems. In *Proc. International Conference on Supercomputing (ICS)*, pages 121–132, 2012.

[76] T. Lutz, C. Fensch, and M. Cole. PARTANS: An autotuning framework for stencil computation on multi-GPU systems. *ACM Transactions on Architecture and Code Optimization*, 9(4):59:1–59:24, 2013.

[77] T. Ma, G. Bosilca, A. Bouteiller, and J. Dongarra. HierKNEM: an adaptive framework for kernel-assisted and topology-aware collective communications on many-core clusters. In *Proc. International Parallel and Distributed Processing Symposium (IPDPS)*, pages 970–982, 2012.

[78] T. Ma, T. Herault, G. Bosilca, and J. J. Dongarra. Process distance-aware adaptive MPI collective communications. In *Proc. International Conference on Cluster Computing*, pages 196–204, 2011.

[79] M. Martinasso, G. Kwasniewski, S. R. Alam, T. C. Schulthess, and T. Hoefler. A PCIe congestion-aware performance model for densely populated accelerator servers. In

Proc. International Conference for High Performance Computing, Networking, Storage and Analysis (SC), pages 63:1–63:11, 2016.

[80] G. Mercier and J. Clet-Ortega. Towards an efficient process placement policy for MPI applications in multicore environments. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface (EuroPVM/MPI)*, pages 104–115. 2009.

[81] G. Mercier and E. Jeannot. Improving MPI applications performance on multicore clusters with rank reordering. In *Proc. European MPI Users' Group Meeting (EuroMPI)*, pages 39–49, 2011.

[82] S. H. Mirsadeghi and A. Afsahi. PTRAM: A parallel topology-and routing-aware mapping framework for large-scale HPC systems. In *Proc. International Parallel and Distributed Processing Symposium Workshops (IPDPSW/HIPS)*, pages 386–396, 2016.

[83] S. H. Mirsadeghi and A. Afsahi. Topology-aware rank reordering for MPI collectives. In *Proc. International Parallel and Distributed Processing Symposium Workshops (IPDPSW/IPDRM)*, pages 1759–1768, 2016.

[84] S. H. Mirsadeghi, I. Faraji, and A. Afsahi. MAGC: A mapping approach for GPU clusters. In *Proc. International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 50–58, 2016.

[85] Message Passing Interface Forum, http://www.mpi-forum.org/, last accessed 2017/05/30.

[86] MPICH: High-performance and widely portable MPI implementation, http://http://www.mpich.org/, last accessed 2017/05/30.

[87] MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE, http://mvapich.cse.ohio-state.edu/, last accessed 2017/05/30.

[88] Nek5000, https://nek5000.mcs.anl.gov/, last accessed 2017/05/30.

[89] NVLink High-Speed Interconnect, http://www.nvidia.com/object/nvlink.html, last accessed 2017/05/30.

[90] NVIDIA management library, https://developer.nvidia.com/nvidia-management-library-nvml/, last accessed 2017/05/30.

[91] Openfabrics alliance, http://www.openfabrics.org/, last accessed 2017/05/30.

[92] Open MPI: Open source high performance computing, http://www.open-mpi.org/, last accessed 2017/05/30.

[93] OSU Micro-Benchmarks, http://mvapich.cse.ohio-state.edu/benchmarks/, last accessed 2017/05/30.

[94] A. Ovcharenko, D. Ibanez, F. Delalondre, O. Sahni, K. E. Jansen, C. D. Carothers, and M. S. Shephard. Neighborhood communication paradigm to increase scalability in large-scale dynamic scientific applications. *Parallel Computing*, 38(3):140–156, 2012.

[95] F. Pellegrini and J. Roman. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *Proc. International Conference on High-Performance Computing and Networking (HPCN)*, pages 493–498, 1996.

[96] Partitioned Global Address Space, http://www.PGAS.org/, last accessed 2017/05/30.

[97] S. Potluri, H. Wang, D. Bureddy, A. K. Singh, C. Rosales, and D. K. Panda. Optimizing MPI communication on multi-GPU systems using CUDA inter-process communication. In *Proc. International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1848–1857, 2012.

[98] Partnership for advanced computing in Europe, http://www.prace-ri.eu/, last accessed 2017/05/30.

[99] B. Prisacari, G. Rodriguez, C. Minkenberg, and T. Hoefler. Bandwidth-optimal all-to-all exchanges in fat tree networks. In *Proc. International Conference on Supercomputing (ICS)*, pages 139–148, 2013.

[100] R. Rabenseifner. Automatic MPI counter profiling of all users: First results on a CRAY T3E 900-512. In *Proc. European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting (EuroPVM/MPI)*, pages 77–85, 1999.

[101] M. J. Rashti, J. Green, P. Balaji, A. Afsahi, and W. Gropp. Multi-core and network aware MPI topology functions. In *Proc. European MPI Users' Group Meeting (EuroMPI)*, pages 50–60, 2011.

[102] RDMA Consortium, http://www.rdmaconsortium.org/, last accessed 2017/05/30.

[103] J.-A. Rico-Gallego and J.-C. Díaz-Martín. On the performance of concurrent transfers in collective algorithms. In *Proc. European MPI Users' Group Meeting (EuroMPI)*, pages 143–144, 2013.

[104] E. R. Rodrigues, F. L. Madruga, P. O. A. Navaux, and J. Panetta. Multi-core aware process mapping and its impact on communication overhead of parallel applications. In *Proc. IEEE Symposium on Computers and Communications (ISCC)*, pages 811–817, 2009.

[105] A. L. Rosenberg. Issues in the study of graph embeddings. In *Graphtheoretic Concepts in Computer Science*, pages 150–176. Springer Berlin Heidelberg, 1981.

[106] P. Sack and W. Gropp. Faster topology-aware collective algorithms through non-minimal communication. In *Proc. Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 45–54, 2012.

[107] H. Shan, J. P. Singh, L. Oliker, and R. Biswas. Message passing and shared address space parallelism on an SMP cluster. *Parallel Computing*, 29(2):167–186, 2003.

[108] A. K. Singh, S. Potluri, H. Wang, K. Kandalla, S. Sur, and D. K. Panda. MPI alltoall personalized exchange on GPGPU clusters: Design alternatives and benefit. In *Proc. International Conference on Cluster Computing*, pages 420–427, 2011.

[109] V. Springel. The cosmological simulation code GADGET-2. *Monthly Notices of the Royal Astronomical Society*, 364(4):1105–1134, 2005.

[110] T. Sterling. HPC in phase change: Towards a new execution model. In *Proc. International conference on High Performance Computing for Computational Science*, pages 31–31, 2011.

[111] H. Subramoni, A. M. Augustine, M. Arnold, J. Perkins, X. Lu, K. Hamidouche, and D. K. Panda. INAM2: InfiniBand network analysis and monitoring with MPI. In *Proc. International Conference on High Performance Computing (HiPC)*, pages 300–320, 2016.

[112] H. Subramoni, K. Kandalla, J. Jose, K. Tomko, K. Schulz, D. Pekurovsky, and D. K. Panda. Designing topology-aware communication schedules for alltoall operations in large InfiniBand clusters. In *Proc. International Conference on Parallel Processing (ICPP)*, pages 231–240, 2014.

[113] H. Subramoni, K. C. Kandalla, J. Vienne, S. Sur, B. Barth, K. Tomko, R. Mclay, K. Schulz, and D. K. Panda. Design and evaluation of network topology-/speed- aware broadcast algorithms for InfiniBand clusters. In *Proc. International Conference on Cluster Computing*, pages 317–325, 2011.

[114] H. Subramoni, P. Lai, S. Sur, and D. K. Panda. Improving application performance and predictability using multiple virtual lanes in modern multi-core InfiniBand clusters. In *Proc. International Conference on Parallel Processing (ICPP)*, pages 462–471, 2010.

[115] C. D. Sudheer and A. Srinivasan. Optimization of the hop-byte metric for effective topology aware mapping. In *Proc. International Conference on High Performance Computing (HiPC)*, pages 1–9, 2012.

[116] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of collective communication operations in MPICH. *International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

[117] Top500 Supercomputing Systems, http://www.top500.org/, last accessed 2017/05/30.

[118] J. L. Traff. Implementing the MPI process topology mechanism. In *ACM/IEEE Conference on Supercomputing (SC)*, pages 28–28, 2002.

[119] J. L. Träff, A. Carpen-Amarie, S. Hunold, and A. Rougier. Message-combining algorithms for isomorphic, sparse collective communication. arXiv:1606.07676, 2016.

[120] J. L. Träff, F. D. Lübbe, A. Rougier, and S. Hunold. Isomorphic, sparse MPI-like collective communication operations for parallel stencil computations. In *Proc. European MPI Users' Group Meeting (EuroMPI)*, pages 10:1–10:10, 2015.

[121] O. Tuncer, V. J. Leung, and A. K. Coskun. PaCMap: Topology mapping of unstructured communication patterns onto non-contiguous allocations. In *Proc. International Conference on Supercomputing (ICS)*, pages 37–46, 2015.

[122] M. Wattenhofer and R. Wattenhofer. Distributed weighted matching. In *Proc. International Symposium on Distributed Computing (DISC)*, pages 335–348, 2004.

[123] K. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. Hilfinger, P. Husbands, et al. Productivity and performance using partitioned global address space languages. In *Proc. International Workshop on Parallel Symbolic Computation (PASCO)*, pages 24–32, 2007.

[124] A. B. Yoo, M. A. Jette, and M. Grondona. SLURM: Simple Linux utility for resource management. In *Proc. International Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, pages 44–60, 2003.

[125] E. Zahavi. Fat-tree routing and node ordering providing contention free traffic for MPI global collectives. *Journal of Parallel and Distributed Computing*, 72(11):1423–1432, 2012.

[126] C. Zimmer, S. Gupta, S. Atchley, S. S. Vazhkudai, and C. Albing. A multi-faceted approach to job placement for improved performance on extreme-scale systems. In *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 87:1–87:11, 2016.