Improving Communication Performance in GPU-Accelerated HPC Clusters

by

Iman Faraji

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
January 2018

Abstract

In recent years, GPUs have been adopted in many High-Performance Computing (HPC) clusters due to their massive computational power and energy efficiency. The Message Passing Interface (MPI) is the de-facto standard for parallel programming. Many HPC applications, written in MPI, use parallel processes and multiple GPUs to achieve higher performance and GPU memory capacity. In such applications, efficiently performing GPU inter-process communication is the key in the application performance.

In this dissertation, we present proposals to improve the GPU inter-process communication in HPC clusters using novel GPU-aware designs, efficient and scalable algorithms, topology-aware designs, and hardware features. Specifically, we propose various approaches to improve the efficiency of MPI communication routines in GPU clusters. We also propose designs that evaluate the total application inter-process communication and provide solutions to improve its efficiency.

First, we propose efficient GPU-aware algorithms to improve MPI collective performance. We show the importance of minimizing CPU intervention on GPU collective performance. We also utilize GPU features to enhance both collective communication and computation.

As inter-process communications scale to across multi-GPU nodes and clusters,

efficient inter-process communication routines must consider the physical structure of the underlying system. Given the hierarchical nature of the GPU clusters with multi-GPU nodes, we propose hierarchy-aware designs for GPU collectives and show that different algorithms are favored at different hierarchy levels.

With the presence of multiple data copy mechanisms in modern GPU clusters, it is crucial to make an informed decision on how to use them for efficient inter-process communications. In this regard, we propose designs that intelligently decide which data copy mechanisms to use in GPU collectives. Using these designs, we reveal the importance of using multiple data copy mechanisms in performing multiple inter-process communications.

Finally, we provide topology-aware solutions to improve the application inter-process communication efficiency, both within multi-GPU nodes and across GPU clusters. First, we study the performance of different communication channels used for GPU inter-process communications. Next, we propose topology-aware designs that consider both the system physical topology and application communication pattern. These designs improve the communication performance by performing more intensive inter-process communication on stronger communication channels.

Statement of Collaboration

The work in Chapter 6 was conducted collaboratively with Dr. Hessam Mirsadeghi. The proposed design in Section 6.3.1 on topology-aware GPU selection schemes on a multi-GPU node was mainly proposed and developed by me, and Dr. Mirsadeghi provided some technical support and insights. The proposed work in Section 6.3.2 on a 3-phase mapping approach for GPU clusters was mainly proposed and developed by Dr. Mirsadeghi; I provided technical and intellectual assistance, and took care of designing and integrating the third phase of the design.

Microbenchmarks were developed jointly. Dr. Mirsadeghi specifically provided some technical supports regarding the implementation of different communication patterns that were used in our microbenchmarks. The extension of the microbenchmarks to support GPU devices was done by me. Moreover, I developed a microbenchmark suite for cluster-wide experiments; this benchmark is capable of providing simultaneous communications among CPUs and among GPUs with different communication patterns.

All the pre- and post-analysis results in this chapter were gathered by me. This includes gathering the motivational, profiling, microbenchmark, and application results. Analysis of the microbenchmark and application results were performed jointly, while analysis of the motivational and profiling results were mainly performed by me.

I also extended the FPMPI profiler to gather the HOOMD-Blue application profiling results.

Acknowledgments

My deep gratitude first goes to my supervisor, Dr. Ahmad Afsahi, for his invaluable support and feedback in writing this dissertation. I am thankful for his intellectual support and expert guidance in my PhD research and study. I also thank him for the patience, focus, and enthusiasm that he has with research that was contagious and motivational for me, particularly during tough times of my PhD career. I would also like to thank the my thesis examining committee, Dr. Tom Dean, Dr. Patrick Martin , Dr. Steven Blostein, and Dr. Tarek Abdelrahman for their feedbacks and comments.

I would like to thank for the financial support provided by Natural Science and Engineering Research Council of Canada (NSERC), Queen's Graduate Award (QGA), and International Tuition Award (ITA) to conduct this research. I also thank Electrical and Computer Engineering Department for the teaching assistantship and fellowship awards. I would also like to acknowledge the resource support from Compute Canada, Calcul Quebec, and the HPC Advisory Council, and especially thank Maxime Boissonneault and Pak Lui for their technical support.

I am thankful to all my colleagues in Parallel Processing Research Laboratory, Dr. Mohammad Javad Rashti, Dr. Ryan Grant, Dr. Reza Zamani, Dr. Judicael Zounmevo, Dr. Hessam Mirsadeghi, Grigori Inozemtsev, Mahdieh Ghazimirsaeeed,

Kaushal Kumar, Mac Fregeau, and Ramapriya Balasubramaniam for their support and great discussions during my PhD program. I am particularly thankful to Hessam Mirsadeghi for his collaboration in the topology-aware research and the technical assistance and intellectual support that he provided.

Finally, I would like to specially thank my family for all of their love, support, and encouragement. For my parents, Mitra and Saeed, who raised me with love and continuously and faithfully supported me in all of my pursuits. For my sister, Saharnaz and her caring support. I would also like to express my immense gratitude to my wife, Arghavan for her love, support, encouragement and patience. I am thankful for her invaluable presence during my PhD journey and look forward to embarking upon new journeys in life with her.

Contents

List of Tables

List of Figures

List of Abbreviations

BTB Binomial Tree Based

CFD Computational Fluid Dynamic

CPU Central Processing Unit

CUDA Compute Unified Data Architecture

ECC Error Correcting Code

ExaFLOP 10^{18} FLoating-point Operations Per Second

FLOPS FLoating-point Operations Per Second

GPU Graphics Processing Units

GSB GPU Shared Buffer

HCA Host Channel Adapter

HPC High Performance Computing

IB InfiniBand

IPC Inter-Process Communication

MD Molecular Dynamic

MPI Message Passing Interface

MPP Massively Parallel Processor

MPS Multi Process Service

NIC Network Interface Card

NVCC NVIDIA C Compiler

NVML NVIDIA Management Library

NVPROF NVIDIA Profiler

NVTX NVIDIA Tools Extension

NVVP NVIDIA Visual Profiler

OpenCL Open Computing Language

OS Operating System

RDMA Remote Direct Memory Access

PCIe Peripheral Component Interconnect Express

PetaFlops 10^{15} FLoating-point Operations Per Second

RMA Remote Memory Access

SIMT Single Instruction Multiple Thread

SM Steaming Multiprocessor

SP Steaming Processor

TPS Time Steps Per Second

Chapter 1

Introduction

High-performance computing (HPC) refers to aggregating the computational power of different processing units to deliver high performance for running challenging applications. These applications can span across different areas such as Molecular Dynamic (MD) simulation in chemistry [27, 18], Computational Fluid Dynamic (CFD) [67] and thermodynamics [45] in physics, N-Body simulations in cosmology [31], and training using deep learning in applications such as speech recognition [14, 2], to name a few. Massive computational power provided by HPC is a key requirement for the high performance calculations and trainings of these applications.

Clusters are the predominant architecture for HPC systems; according to the TOP500 list of the world's most powerful supercomputers [91], 432 of the top 500 supercomputers are clusters. A computer cluster consists of a number of computing systems that are loosely coupled through a high speed interconnection network. The key advantage of the cluster architecture is that it provides high performance with

high degree of availability, and relatively lower cost compared to the alternative architectures, such as massively parallel processors (MPP)[1]. The node architecture of a modern HPC cluster typically consists of multi-core processors, co-processors, and accelerators. GPUs, among other co-processors and accelerators, have successfully established themselves in HPC clusters due to their high performance and energy efficiency. These factors are the key requirements of the future supercomputers, thus paving the way for GPUs to be continually used in current petaflop (10^{15} FLoating-point Operations Per Second) and future exascale (10^{18} FLOP) systems. As a matter of fact, the two next 100+ petaflops supercomputers announced by the CORAL program [20] (i.e., Sierra [83] and Summit [89]) will use GPUs and move the world closer to exascale.

GPUs are composed of thousands of processing units, and use a data-parallel model to distribute the application dataset among them to parallelize and accelerate computation. The massive computational capability of the GPU hardware makes it a good candidate to offload and accelerate compute-intensive portions of the applications running on HPC clusters.

Inter-process communication is one of the key factors in determining the performance of HPC applications. Different processes may communicate with each other using different programming models, such as shared memory or message passing model. The message passing programming model provides higher scalability and thus is considered as the main programming model for high-end computing systems. The Message Passing Interface (MPI) [56] is the de-facto standard for the message passing programming model. In HPC clusters with GPU accelerators, while computation can

[1]Although our proposals in this dissertation are evaluated on GPU-accelerated HPC clusters, they can also be adapted and used on the GPU-accelerated MPP architectures.

be offloaded and accelerated on the GPUs, support from MPI library is required to allow processes to communicate the data residing in GPU memory.

While GPUs can accelerate the offloaded computation, inefficient GPU-to-GPU communications may wipe out the benefits of offloading in the first place. Consequently, efficient usage of GPU accelerators in HPC clusters demands both efficient GPU computations and inter-process communications. In this dissertation, we seek to improve GPU inter-process communication performance by improving communication latency/bandwidth and application communication efficiency (communication efficiency in short). Communication latency refers to the inter-process communication time, and communication bandwidth refers to the message volume transferred per second. Improving communication efficiency refers to improving the total application communication performance involving all GPU inter-process communications. For example, application communication efficiency can be improved by overlapping different GPU inter-process communications or by efficiently assigning GPUs to processes, while the latency or bandwidth of individual GPU inter-processes communications may not be necessarily improved.

There are different ways to improve the performance of GPU inter-process communications; this includes devising efficient GPU-aware designs and algorithms, designs that are aware of the topology of the GPU clusters, as well as exploiting advanced GPU hardware features. In this dissertation, we use various combinations of these approaches to propose designs that seek to improve the performance of GPU inter-process communications.

1.1 Problem Statement

In MPI, processes can pass messages to each other using three different communication models: 1) point-to-point; 2) collective; and 3) Remote Memory Access (RMA). In point-to-point, a pair of processes communicate with each other in a cooperative fashion. Collective operations, on the other hand, involve communications among two or more processes. In RMA, a process is involved in a one-sided communication. The current MPI standard is developed for systems with multiple CPUs and no accelerators. With the emergence of GPU accelerators, MPI need to evolve to provide efficient support for such accelerators. Integrating GPU awareness into communication runtime libraries requires careful designs and optimizations that are specific to the GPU architectures. It is therefore crucial to understand that such designs and optimizations do not have to necessarily follow the designs that work efficiently on traditional CPU clusters. This is due to the fact that CPUs and GPUs have different architecture, hardware features, and programming model. On the other hand, GPU devices, similar to multi-core CPUs, use a hierarchy of communication channels to interconnect with each other. However, these communication channels have different nature and physical topology. Consequently, many of the concepts that are used to improve CPU inter-process communications in homogeneous nodes and clusters cannot apply to the heterogeneous nodes and clusters and must be redesigned. Taking these into account, in this dissertation we seek to address the following questions:

- How can GPU-aware algorithms benefit MPI collective communication routines? What GPU features can be effectively used in these designs?
- Can hierarchical designs outperform the existing flat designs in MPI collective

communication performance on clusters of multi-GPU nodes? If yes, what is the sensitivity of different algorithms to different hierarchy levels and which combination of them is the most effective one?

- Which data copy mechanisms should be used for efficient GPU inter-process communications? Can multiple data copy mechanisms be used in conjunction with each other to perform multiple GPU inter-process communications and enhance collective communications?

- What are the different communication channels in a cluster of multi-GPU nodes and how does their performance vary from each other? What is the communication pattern of GPU applications and how it can be profiled? How can topology-aware designs improve the efficiency of inter-process communications in a multi-GPU node and across the GPU clusters?

1.2 Contributions

This dissertation presents several proposals to improve the GPU inter-process communications in HPC clusters with GPU accelerators. It contributes by addressing several sources of inefficiencies in such communications by providing new or improved designs.

(1) Efficient GPU Collective Communication Algorithms

On one hand, collective communications contribute to considerable portions of MPI applications runtime [75]; on the other hand, efficient GPU inter-process communication is of crucial importance in GPU-accelerated applications [70]. Taking these into account, in Chapter 3, we propose GPU-aware algorithms [23, 22] to improve the performance of the GPU collective operations. We provide designs and

experimental results[2] for MPI_Allreduce operation that involves both collective communication and computation. Our proposed algorithm can be applied to other collective communications with minor changes as well. In these designs, we also incorporate advanced GPU features for further performance improvement. The proposed designs show the importance of minimizing CPU intervention in GPU collective operations. We achieve this by proposing designs that are capable of performing inter-process communications through GPU shared memory regions, and utilizing in-GPU kernels to manipulate the collective data residing in GPU memory. Our proposed collective designs in this chapter provide up to 22 and 5 times performance improvement over the existing designs within a single-GPU node and across the cluster of single-GPU nodes, respectively.

(2) Hierarchical Framework for GPU Collective Communications

The contributions in Chapter 4 revolve around utilizing the hierarchical nature of HPC clusters with multi-GPU nodes. This chapter proposes a hierarchical framework for collective communication operations in clusters of multi-GPU nodes [22]. The proposed hierarchical framework breaks down the collective operations into different stages. Operations within each stage are performed on a single hierarchy level. This chapter evaluates the effect of different algorithms within different hierarchy levels. It also shows that by choosing the right set of algorithms in the hierarchal framework, collective communication operations can highly outperform the native flat designs. Using our hierarchical framework, we provide up to 80% and 65% performance improvement on MPI_Allreduce over the existing flat designs within a multi-GPU node and across the cluster of multi-GPU nodes, respectively.

[2]In this dissertation, all result are reproducible by following the proposed design steps and using the same environment and settings.

(3) Efficient GPU Communications through Smart data copy mechanism Selection

Chapter 5 first evaluates the impact of using different data copy mechanisms on GPU inter-process communications. Depending on the GPU communication characteristics such as the message size, one data copy mechanism is usually preferred over the other. We then provide evidence that multiple GPU inter-process communications can take advantage of jointly using different data copy mechanisms. Accordingly, this chapter proposes alternative designs for GPU collective operations that use different data copy mechanisms to perform collective operations [24, 22]. The proposed designs are capable of efficiently deciding the data copy mechanisms based on the information that they gather either during or prior to the runtime. The main contribution of this chapter is to show the potential of using different GPU data copy mechanisms with different communication channels to speedup the total GPU inter-process communications. Using our proposals, we show up to 2.62 times speedup in the total GPU inter-process communications.

(4) Topology-aware GPU Communications

Chapter 6 first provides a comprehensive evaluation of different communication channels that are used for inter-process communications in multi-GPU nodes. Evaluation results show substantial performance difference among different communication channels that are interconnecting processing units in multi-GPU nodes. Accordingly, Chapter 6 presents topology-aware mapping solutions that map MPI processes to processing units in a way to improve the total inter-process communications. The proposed topology-aware designs target both single [25, 26] and clustered multi-GPU

node(s) [55]. For single-GPU nodes, we use a non-trivial topology-aware GPU selection scheme that considers the application communication pattern and the physical topology of the node. Three metrics are proposed to represent the topology of the multi-GPU nodes: 1) latency; 2) bandwidth; and 3) communication distance. This chapter also contributes by extending the topology-aware GPU selection scheme from a multi-GPU node to across the GPU cluster. In this regard, cluster-wide topology-aware communication is defined as a mapping scheme which breaks down the mapping into three distinct phases: 1) internode process-to-node mapping; 2) intranode process-to-CPU-core binding; and 3) intranode process-to-GPU assignment. Performance results show substantial improvement over native designs at both microbenchmark and application levels. On a multi-GPU node, our topology-aware proposal provide up to 72% and 21% improvement in performance at the microbenchmark and application levels, respectively. Our proposals also improve the total benchmark runtime by 90% and show up to 8% application performance improvement across the GPU cluster.

1.3 Dissertation Outline

The rest of this dissertation is organized as follows. Chapter 2 provides some background materials on HPC clusters with GPUs, modern GPU architectures, and their advanced programming tools and features. This chapter also overviews the Message Passing Interface and GPU-aware MPI communication libraries. In Chapter 3, GPU-aware algorithms for improving collective operations are presented. Chapter 4 presents a hierarchical framework for GPU collectives targeting clusters of multi-GPU nodes; it also evaluates the sensitivity of different algorithms to different hierarchy

levels for these operations. Chapter 5 shows the efficiency of using different data copy mechanisms to perform multiple inter-process communications; this chapter builds on this observation and provides novel designs for GPU collective operations. In Chapter 6, we provide topology-aware solutions for efficient communication in multi-GPU nodes and clusters. Finally, Chapter 7 concludes this dissertation and outlines some future research directions.

Chapter 2

Background

Over the past decade, the high-performance computing landscape has changed significantly, particularly due to the emergence of accelerators. In particular, GPU accelerators have established themselves in modern heterogeneous HPC clusters by offering high performance and energy efficiency. Consequently, heterogeneous clusters with GPU accelerators have become the platform of choice for many HPC applications. In this chapter, we provide a brief overview of the HPC clusters with GPU accelerators. Then, we discuss the state-of-the-art architecture, programming model, and features of the GPU accelerators. We introduce the Message Passing Interface (MPI) as the de-facto standard for parallel programming and discuss integration of GPU-awareness into some of its implementations. Finally, we go over modern interconnects and communication channels that are typically used in HPC clusters with GPU accelerators.

2.1 HPC Clusters with Accelerators

An HPC cluster is composed of a large number of independent compute nodes loosely interconnected together via an interconnection network, together providing a single

computing resource. This is in contrast to the Massively Parallel Processor architecture, where there is only one machine with all processing units tightly interconnected together usually with a custom designed interconnect. MPPs, due to their custom and proprietary design are more expensive than clusters. However, the custom design of MPPs can potentially make them a better fit for some specific applications. In this dissertation, our designs target HPC clusters with GPU accelerators.

HPC clusters benefit from high bandwidth and low latency interconnection networks due to the high demand of HPC applications for frequent communications among HPC nodes. The node architecture of modern HPC clusters consists of multi-core processors and accelerators/co-processors. Accelerators/co-processors provide high compute capacity and low power dissipation, making them a promising candidate in improving the performance per watt of the HPC clusters. GPUs, compared to other accelerators and co-processors have gained the widest adoption in modern HPC clusters. This trend is continued by the announcement of Sierra and Summit, two 100+ petaflops supercomputers, in the CORAL program [20]. The high share of GPUs in HPC clusters is mainly rooted in meeting the high performance, memory bandwidth capacity, and power efficiency that is required by the compute engines of the next generation HPC clusters.

2.2 Graphics Processing Units (GPUs)

In this section, we introduce the architecture, programming model, and advanced hardware features of the state-of-the-art GPU accelerators. NVIDIA [62] and Advanced Micro Device (AMD) [1] are the two leading manufacturer of GPU chips.

NVIDIA GPUs, however, have higher adoption in the top500 supercomputers. According to TOP500 list, only one out of 72 supercomputers with GPU accelerators use AMD GPUs and the rest exploit NVIDIA GPUs. Taking this into account, we only target NVIDIA GPUs and perform our experiments on them. Thus, the terminologies and GPU features used in this dissertation, unless otherwise specified, only apply to the NVIDIA GPUs.

2.2.1 GPU Architectures

A modern GPU architecture is assembled of an array of Streaming Multiprocessors (SM) [44]. SMs are composed of a set of Stream Processors (SPs), each executing instructions of a thread at a given time (cycle). SPs share control logic and an instruction cache, while SMs have access to the global memory. GPUs follow SIMT (Single Instruction Multiple Threads) model [47] in which a group of (currently 32) threads known as a warp execute the same instruction. Thousands of such SPs exist in modern GPU devices; this means that each GPU is capable of executing thousands of threads at any given moment. But in reality, it is possible to issue many more threads than the number of existing SPs.

Thread Organization:

Parallel execution on GPUs is handled by fine-grained threads. Upon GPU kernel invocation, a grid of threads is issued, all executing the same kernel. Threads in CUDA (Compute Unified Data Architecture) are organized in a two-level hierarchy: grid are at the top of the hierarchy, consisting of blocks; blocks are placed at the lower hierarchy, consisting of threads. Threads within a block have access to the shared memory and executing the same instructions, while threads between the blocks have

access to the global memory and can execute different instructions. Both threads and blocks are coordinated so individual threads can distinguish themselves among each other and detect the portion of the data they are supposed to work on. The maximum number of blocks/threads that can fit in a single grid/block is implementation dependent, and is known as gridDim (gird dimension) and blockDim (block dimension), respectively. The SM bundles a group of threads into a warp to work in an SIMT fashion; this way all of the threads within a warp require only a single control unit.

GPU Memories:

Memory access latency is one of the limiting factors in CUDA applications that undermines its computational capabilities. To address this, there exists different types of GPU memories. The GPU global memory is the slowest and most spacious GPU memory; this memory space can be accessed from the GPU that it belongs to, the host, or other peer accessible GPUs in the system. Constant memory can be read or written by the host, but threads on the device can have read-only access to it; it provides a fast, high-bandwidth access, specifically when multiple threads try to access the same location. Shared memory is a fast on-chip memory that threads on a block have access to (a portion of the shared memory, depending on the number of blocks, is assigned to each SM). Registers are also fast on-chip memories, but are allocated to single threads, so each thread can only access its pertinent register.

Floating-Point Units:

GPU supports single- and double-precision floating point (as well as half-precision in the latest Pascal GPU architecture) operations which has expanded the domain of supported applications on GPU devices.

Streams and Events:

Stream is a sequence of CUDA instructions, executed serially. After Compute Capability 2, multiple CUDA streams can run on a single GPU. An event is a dummy instruction injected into the stream. One purpose of the event is to calculate the elapsed time between events; by also synchronizing on an event, one can make sure all of the instructions before the event have already been executed.

2.2.2 GPU Programming Languages and Tools

GPU Programming Languages

All of the GPU programming languages discussed here are aimed to provide an environment in which GPU and CPU programs can coexist with each other. The main goal of these programming languages is to offload the GPU friendly potion of the program into the GPU memory. To achieve this, different programming languages use different techniques and APIs. In the rest of this section, we will review three of the most popular GPU programming languages, namely CUDA [61], OpenCL [88], and OpenACC [103].

CUDA: CUDA (Compute Unified Device Architecture) [61] is an extension to ISO C, developed by the NVIDIA Corporation [62]. One way to look at a CUDA program is to consider it as C code which only includes host code. To make use of the GPU resources, the code can be optimized by adding device keywords and APIs. The CUDA compiler is called NVCC (NVIDIA C Compiler) and is responsible for separating the host code and the device code to be compiled by the C compiler and the runtime component of NVCC, respectively. In this dissertation, we use CUDA as our programming platform.

OpenCL: OpenCL [88] is another GPU programming language, designed for heterogeneous systems. OpenCL was initiated by Apple and maintained by the non-profit technology consortium Khronos Group [58], and has been adopted by Intel, AMD and NVIDIA. OpenCL is fundamentally the same as CUDA and in most cases there is a one-to-one correspondence between their features. Like CUDA, OpenCL is also based on the C language and targets heterogeneous systems. Unlike CUDA, OpenCL is not platform-, vendor-, or hardware-specific. Portability of OpenCL across various platforms and hardware does not come at no cost. It inevitably requires incorporating complex device management model and would require optional features that only specific devices can use.

OpenACC: The OpenACC specification [103] is provided by a non-profit foundation which is initially formed by Portland Group Inc. (PGI), Cray Inc., and NVIDIA. The goal of OpenACC is to provide an environment for the scientists to easily accelerate their programs using directives. OpenACC APIs are, in general, a set of compiler directives and library routines that can be used to specify loops or other regions of the code to be offloaded to the accelerator devices, including GPUs. OpenACC unlike CUDA and OpenCL, removes the burden of initializing the kernel and associated data movement from the user, leaving these heavy lifting details to the compiler and the runtime library. However, like CUDA and OpenCL, the programmer is still responsible for recognizing parallelizable regions of the code and has to specify the data that is going to be locally available to the accelerator. OpenACC compared to the other two aforementioned programming languages is attractive in such a way that it provides less barriers for heterogeneous programmers to utilize accelerators. On the other hand, OpenACC directives just provide some hints from the programmer,

and efficient use of the accelerator is up to the compiler while CUDA/OpenCL relies less on the compiler as parallelism is explicitly mentioned by the programmer. In a nutshell, in CUDA/OpenCL, programmers have more flexibility to exploit the accelerator resources.

In this dissertation, we only use the CUDA programming language which is specifically tuned for NVIDIA GPUs. We use NVIDIA GPU features (e.g., Hyper-Q), tools (e.g., NVIDIA profiling tool), and libraries (e.g., NVIDIA management library) to monitor and enhance the efficiency of our proposals. However, none of these tools is required to implement our proposed designs in other programming libraries, such as OpenCL. Our designs in CUDA can be potentially converted to OpenCL using CUDA-to-OpenCL translator, such as CU2CL [51], which provides source-to-source translation of CUDA to OpenCL codes.

GPU programming Tools

NVIDIA Management Library: The NVIDIA Management Library (NVML) [63] includes a set of C-based APIs that can be used for extracting various states and characteristics of the NVIDIA GPU devices, including monitoring, managing and querying the GPU states and topology information. The NVML library can be used to query various GPU states and information. Some of the query-able states include the GPU performance state, current GPU core temperature and board power draw, and GPU resource/memory utilization. The NVML library also allows the user to modify various GPU states. Some of the modifiable states include, enabling/disabling Error Correcting Code (ECC), changing the GPU compute mode (to control whether compute processes can run exclusively or concurrently with other processes). The

NVML library also provides a set of APIs to retrieve some information from the GPU(s). Some of this retrievable information includes, the GPU BUSID/UUID, and the topology information. Moreover, NVML provides APIs to retrieve the GPU topology information of the node. For instance, the topology API function *nvmlDeviceGetTopologyCommonAncestor()* can be used to find the common ancestor in a GPU pair. The retrieved common ancestor value represents the node level relationship between two GPUs (the larger this value, the higher the topology level between the GPUs would be). The *nvmlDeviceGetTopologyNearestGPUs()* is another example of NVML topology API, which provides a set of GPUs that are nearest to a given GPU at a specific interconnectivity level. We use NVML topology APIs in Chapter 6 to extract the topology information of the multi-GPU node.

GPU Profiling Tools

GPU Profiling tools and APIs allow one to better understand and optimize the GPU computation and communication performance of HPC applications.

NVIDIA Profiling Tools: NVIDIA provides a set of profiling tools and libraries [65] to trace the CPU and GPU activities of the application. The NVIDIA Visual Profiler (NVVP) is a graphical profiling tool that provides a timeline to demonstrate these activities. The NVIDIA Profiler (NVPROF) allows to collect and view the application profiling data from the command-line. With NVVP and NVPROF, one can collect and show the trace of the application GPU calls that are made by the CPU threads. However, in order to understand what tasks are being performed by the CPU threads outside of the GPU function calls, the NVIDIA Tools Extension API (NVTX) can be used. By adding the NVTX markers and ranges to the application, the Timeline View is capable of showing both the CPU and GPU activities that are being executed

by the CPU threads. More specifically, we use NVTX to annotate MPI routines, and assign MPI ranks to their associated process ids and GPU contexts on the profiler timeline. We also use NVPROF to present an overview of the instructions launched by the CUDA runtime or driver API. The log file provided by the NVPROF and NVTX is eventually fed to NVVP which provides a trace of the CPU and GPU activities of the application. We use NVIDIA profiling libraries in Chapter 5 to profile how different data copy mechanisms are used in our proposed designs for GPU collective operations.

GPU-Aware FPMPI Profiling library: FPMPI [29] is a profiling library which provides various information about the underlying MPI (MPI will be discussed in Section 2.3) communications of an application. Such information, in general, can be categorized into three groups: 1) synchronization data; 2) asynchronous communication data; and 3) topology data. The synchronous communication routines provide some related profiling data, while the asynchronous data lists the asynchronous communication routines. The topology data provides a brief output of the communication topology. While FPMPI is capable of providing such profiling information, it does not distinguish between the CPU and GPU communications. In other words, FPMPI provides one list of profiling data for all MPI routines regardless of where their associated communication buffer is allocated. In this regard, we have extended the FPMPI library to provide profiling support for both CPU and GPU communications. The extended profiler allows us to separately extract the CPU and GPU communication characteristics of an application. To this end, we leverage various CUDA APIs to analyze the buffer(s) in MPI routines. By analyzing the buffer(s), we can determine whether it is located on the host main memory or on the GPU global memory. We

also instrument Open MPI [68] to expose specific information that will be queried by the FPMPI library. For instance, we add the address type of the send/receive buffers of MPI routines to the MPI_Request object to distinguish among different types of communications (i.e., CPU versus GPU). Our proposed GPU-aware FPMPI library is capable of providing a separate profiling list for both the CPU and GPU MPI routines. The GPU-aware FPMPI is used to profile the GPU application in Chapter 6.

2.2.3 GPU Advanced Features

In this section, we review some of the state-of-the-art features that exist in the latest generations of the NVIDIA GPUs.

GPU Inter-process communications Modern NVIDIA GPUs provide data copy mechanisms that can facilitate and improve the GPU inter-process communications. In this regard NVIDIA introduced CUDA Inter-Process Copy (IPC) and GPUDirect Remote Direct Memory Access (RDMA) technology for intranode and internode GPU inter-process communications.

With The NVIDIA CUDA IPC, data can be directly copied (without host intervention) from the GPU address space of one process to the GPU address space of another process within the same root complex on the node. The CUDA IPC copy requires a process to expose a portion of its address space to the remote processes. In this regard, a memory handle of the shared address is created and passed to the remote processes. The remote processes can then access and modify the shared remote address space; however, synchronization between the involved processes is required to guarantee the completion of the copy. This synchronization is performed using a

shared CUDA IPC event, by one process recording it after initiating its IPC copy and the other process querying its completion.

GPUDirect RDMA is a capability that enables GPUs on different nodes to directly exchange data without needing to go through the CPU/system memory. This feature is introduced in the NVIDIA Kepler-class and allows third party devices such as InfiniBand (IB) [32] adapters to directly access memory on multiple GPUs within the same system but on different nodes, thus allowing them to directly communicate with each other. None of the proposed techniques in this dissertation rely on this feature.

Hyper-Q and Multi Process Service: Hyper-Q [62] is an NVIDIA feature that provides potential concurrency among CUDA tasks from a single process. However, Hyper-Q by itself cannot provide concurrency among CUDA requests from multiple processes to the GPU compute and memory engine, thus these tasks would have to serialize. In order to provide such concurrency across multiple processes, NVIDIA has introduced the Multi Process Service (MPS) [64] for GPUs with compute capability of 3.5 and above. The MPS service acts as a funnel to collect CUDA tasks from multiple intranode processes and issue them to the GPU as if coming from a single process so that the Hyper-Q feature can take effect. Without this service, each of the MPI processes has to allocate storage and scheduling resources on the GPU, and only work from a single context can be launched on the GPU engines at a time. In contrast, with the MPS service enabled, there is only a single context, known as MPS context, present on the GPU. This allows all processes to share the GPU storage and scheduling resources, eliminating the overhead of the context switching. This feature is used in Chapter 5 to enhance the overlap between different GPU data copy mechanisms used in GPU collective communications.

Unified Virtual Addressing (UVA) UVA is an NVIDIA feature which has become available after CUDA 4.0. It maps GPU buffers into a single virtual address space and provides an aggregated virtual address space that is shared among the CPUs and the GPUs in the node. We use this feature in Chapter 3 to 6 to distinguish the GPU and CPU physical memory locations based on the buffer address value.

Unified Memory (UM) NVIDIA UM was introduced with CUDA 6. It creates a pool of managed memory that is shared between the CPU and GPU. With this feature the system automatically migrates data allocated in unified memory between host and device. This feature would allow codes running on the CPU and GPU to seamlessly use the system CPU and GPU memories. This feature eases hybrid programming and is not used in this dissertation.

2.3 Message Passing Interface (MPI)

The Message Passing Interface (MPI) [56] is a message-passing library which is considered as the de-facto standard programming model in HPC clusters. MPI has mainly gained interest due to its high performance, scalability, and portability. MPI has resulted from a joint effort of numerous groups and individuals starting in 1992. The first version of MPI standard (i.e, MPI-1.0) was released in 1994; The second MPI standard, MPI-2, was completed in 1998. MPI-3 was approved in 2012, and the latest available version of the MPI standard was released in 2015 (i.e., MPI-3.1). In the rest of this section, we first overview the message passing communication subsystem. Next, we introduce various MPI communication models and specifically overview the well-known algorithms that are typically used with the collective communication model. In this section, we also overview the current GPU support of some of the

well-known MPI libraries.

2.3.1 Message Passing Communication Subsystem

Fig. 2.1 shows different levels of the message passing subsystem architecture. The application is at the highest level. Below the application level is the middleware library level; the MPI routines are exposed to the application developer at this level. The middleware library sits on top of the kernel-level or the user-level communication libraries. Both the middleware and the user-level libraries are directly accessible by the user and do not require any OS kernel intervention. At the kernel communication level, libraries, such as socket, interact with the kernel-level network protocols (such as TCP/IP); the kernel-level interfaces the kernel-level library with the Network Interface Card (NIC) driver. The user-level library, on the other hand, provides communication libraries that can bypass the OS; communications through this layer decreases the processing overhead between the middleware library and the NIC. As an example, the user level verb library allows MPI to interface to the Mellanox InfiniBand NIC driver or directly to the NIC hardware. The NIC hardware is connected to the network fabric (such as InfiniBand), which connects the node to the rest of the computer cluster. Our proposals in this dissertation are applied in the middleware library and we also utilize some functions from the user-level library.

2.3.2 MPI Communication Models

In MPI, processes pass messages to each other in a cooperative fashion; this is known as the classical point-to-point communication model. MPI also provides an extension to this model for collective and Remote Memory Access (RMA) operations, we discuss

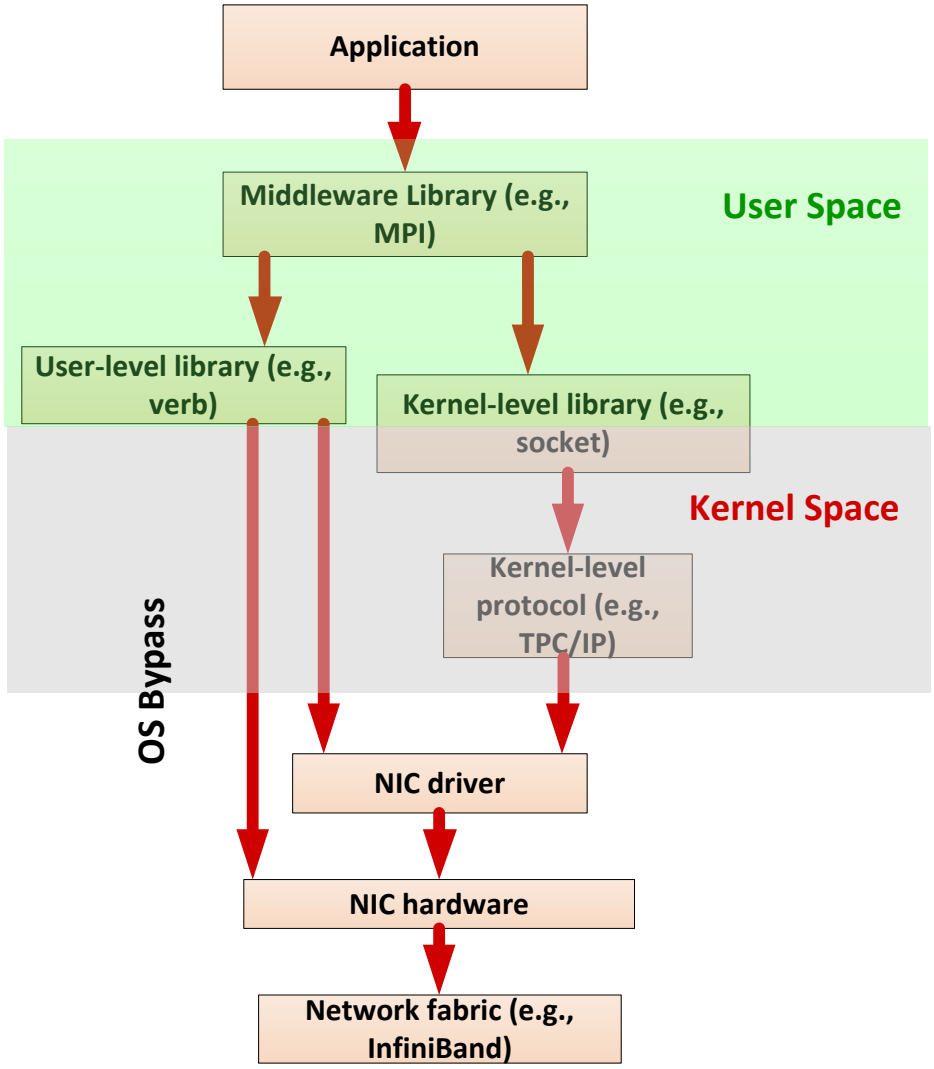


Figure 2.1: Message passing communication diagram

these communication models in more detail below.

Point-to-point: In MPI point-to-point communication, both sender and receiver take part in the communication. The sender calls a send routine, such as MPI_Send(), and the receiver calls a receive routine such as MPI_Recv(), both providing matching operations in order to recognize and select the right message. MPI point-to-point

communication comes in two flavors: 1) blocking and 2) non-blocking. The sender and receiver can either be blocking or non-blocking. The blocking sender blocks the calling process until it is safe to reuse the send buffer again; the blocking receiver blocks the receiver process until the receive operation is complete. In non-blocking send, the send operation will return as soon as the data is copied into the send communication buffer; non-blocking receive returns as soon as the receive request is posted. Non-blocking operations require polling or waiting to verify the completion of the message.

Collectives: MPI also supports collective communication operations which involves communications among two or more processes. Collective communications simplifies programming parallel applications, and facilitates implementation of efficient communication on various machines; this would in turn promote the portability of the application. The performance of different collective operations with different configurations, such as message size and process count, highly rely on the algorithm used to implement them [76, 10, 90].

Prior to MPI-3 all of the collective operations were blocking. Non-blocking collectives, first introduced in the MPI-3, are used to optimize collective communication by enabling overlap between communication and computation. In the blocking collective, the caller is blocked until it is safe to use the buffer it passed to the collective operation; while, in the non-blocking version, the call returns immediately which can be queried later to check its completion.

MPI collectives, in general, can be categorized as computational, synchronization, and data movement routines. In computational collectives, a group of processes work together to perform computations on a dataset that is distributed among them.

For example, MPI_Reduce performs a reduction operation (such as addition) on a dataset that is distributed among processes and stores the result in the root process. MPI_Allreduce is another example of the computational collectives in which a reduction operation is performed on a distributed dataset, and the result is gathered by all processes.

The only operation in the synchronization category is MPI_Barrier which also comes in a non-blocking format (i.e., MPI_Ibarrier). This routine is considered to be complete once all of the processes have called it. Collective operations for data movement have two types. The first type distributes/stores the result from/into one process known as root (e.g., MPI_Bcast and MPI_Gather); the second type stores the final result in all of the processes (e.g., MPI_Allgather); another collective operation in this type is MPI_Alltoall in which each process sends a different chunk of data to each of the other processes. Neighborhood and Non-blocking collectives are the newly added collective operations in the MPI-3 standard [56].

Neighborhood collectives are introduced in MPI-3 standard [56] for the high demand of applications with sparse communication patterns (e.g., applications with stencil kernels). Before MPI-3, it was possible to use MPI process topology to create process topology graph, however, no communication function was introduced to utilize them. Neighborhood collectives enable collective operations to perform computation along the edges of the process topology.

RMA: In an RMA operation, a process is involved in a one-sided communication, specifying communication parameters for both sending and receiving sides. One side of the communication can directly read or write from/into an exposed memory window of the other side. MPI-2 RMA was strictly devised for limited application

behavior patterns, and it had many missing features. MPI-3 addresses some of these shortcomings and supports broader application domain by optimizing window creation, memory model, and the synchronization methods. In this dissertation, we do not consider MPI RMA operations and mainly target collective communication routines.

MPI Libraries: Today, various MPI implementations are available; some are proprietary and some are open source. Intel-MPI [35] is an example of a proprietary implementation, while MPICH [57], MVAPICH [59], and Open MPI [68] are three of the open source implementations of the MPI library. MPICH is a widely portable implementation of the MPI standard that is maintained by the Argonne National Laboratory. MPICH3 is the base source code for many other open source and proprietary implementations that target specific interconnection networks. MVAPICH/MVAPICH2 is an example of such implementations, which is maintained by the Ohio State University and is optimized for using InfiniBand [32], Omni-Path [19], iWARP [78], and RoCE [5] networking technologies. Open MPI is also an open source implementation of the MPI standard that is maintained by a consortium of academic, research, and industry partners. It combines several features from different MPI implementations and has widespread use due to its high community support. We utilize MVAPICH2 library in Chapter 3 and 4; we use both MVAPICH2 and MVAPICH2-GDR libraries in Chapter 5; in Chapter 6, Open MPI library is used.

2.3.3 MPI Collective Communication Algorithms

Collective communications are important and highly used component of MPI [76]. In HPC applications, there are stages of local computation followed by global communication. Programming such global communications can be simplified by using MPI collective operations. Efficient implementation of such operations on one hand removes this burden from the programmer and also promotes the portability of the application to different machines. In this regard, various algorithms have been proposed for different configurations of collective operations. Below, we overview some of the well-known collective algorithms.

Fan-in/fan-out: In the fan-in algorithm [28] with radix-n, a single process, the parent process, serially receives data from n other processes; this algorithm can be used in collective operations such as MPI_reduce and MPI_gather. In the fan-out algorithm with radix-m, the parent process informs processes that data is ready and m processes can attempt to simultaneously read this data; this algorithm can be used in collective operations such as MPI_Scatter and MPI_Bcast. The fan-in and fan-out operations can be combined together and used in collective operations such as MPI_Allreduce. The fan-in/fan-out algorithm is useful when synchronization cost among participating processes in the collective operation is costly. Fig. 2.2.(a) shows the general steps of the radix-7 fan-in/fan-out algorithm.

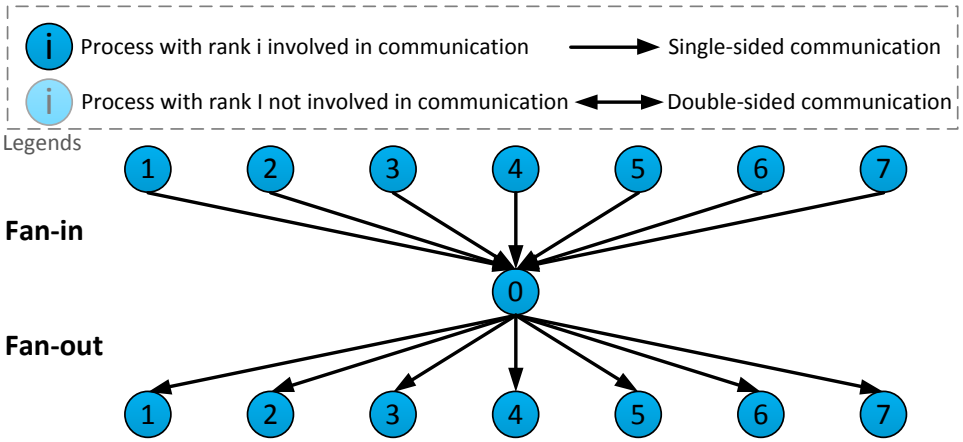
Binomial Tree: The steps of the binomial tree algorithm [90] are shown in Fig. 2.2.(b). In the first step, the root process sends data to process $root + P/2$ with P being the total number of processes; in the next step, both this and the root process act as new roots in their own subtrees and algorithm continues recursively. This communication takes a total of $log(P)$ steps. The binomial tree algorithm can be used in various collective operations such as broadcast, gather, and reduce. However,

this algorithm is traditionally favored for short message sizes in flat systems.

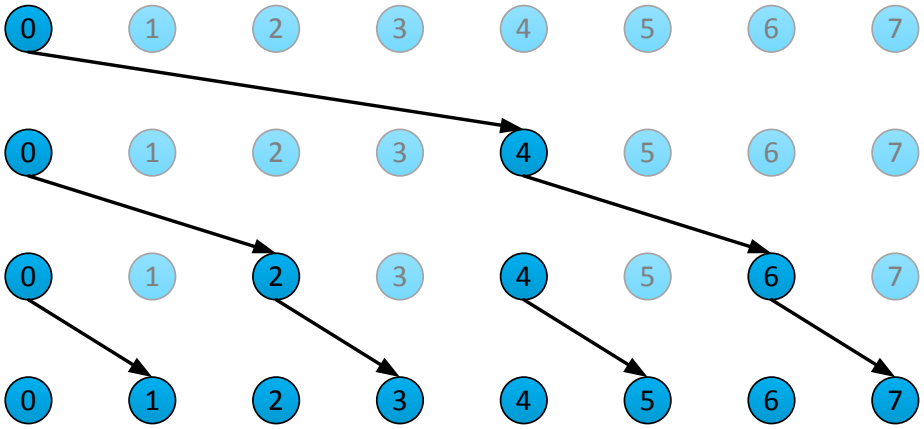
Recursive Doubling: Fig. 2.2.(c) shows the steps of the recursive doubling algorithm [90]. In the first step, processes that are a distance 1 apart exchange their data. In the second and third step, processes that are distance 2 and 4 apart exchange their data, respectively. For P number of processes, recursive doubling ends in $log(P)$ steps. Recursive doubling works well for power-of-two number of processes, while it is tricky to use it with non-power-of-two number of processes. Recursive Doubling is used in MPI collectives such as MPI_Allgather and MPI_Allreduce. In MPI_Allreduce, however, each process also performs a local reduction on the exchanged data in each step. Recursive doubling is usually avoided when large messages are involved and other algorithms, such as the ring algorithm, are used [76].

Rabenseifner: The Rabenseifner algorithm [76] implements MPI_Reduce as reduce-scatter followed by a gather to the root. At each reduce-scatter step, each process works on reducing a portion of the data. A gather operation (or allgather in case of MPI_Allreduce) is called to collect the reduction results into the root.

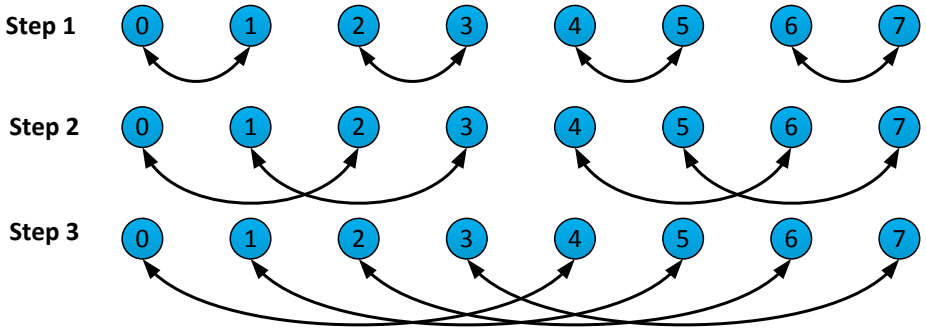
Ring: In the ring algorithm [90], each process sends its data around a virtual ring of processes. As shown in Fig. 2.2.(d), in each step process i sends its contribution to process $i + 1$ and receives the contribution from process i - 1 (with wrap-around). In each step of this algorithm, each process sends the data it received in the previous step. With P processes, the ring algorithm takes P - 1 steps to complete. The ring algorithm is used in MPI collective operations such as MPI_Allgather. It is showed, however, to only work efficiently on collectives with large message sizes.



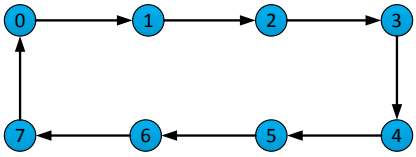
(a) Fan-in/fan-out



(b) Binomial Tree



(c) Recursive Doubling



(d) Ring

Figure 2.2: Well-known MPI collective algorithms

2.3.4 GPU-Aware MPI

There are many MPI applications that are written from scratch or have been adapted to run on the GPU clusters. In such clusters, the compute-intensive portion of the application is offloaded and accelerated on the GPU. In such applications, MPI processes are required to communicate the data that is residing on the GPU buffers. The GPU-aware MPI can remove the burden of learning a new programming language from the programmer to use MPI efficiently in conjunction with the GPU. The GPU-aware MPI would help the programmer to develop a more concise, readable and even efficient application to run on the GPU clusters.

In this regard, the GPU support has been added to the well-known MPI implementations such as MVAPICH2 [59] and Open MPI [68] in order to facilitate the data movement between MPI process from/to the GPU buffers. The GPU support in MPI libraries may follow a general approach which involves staging the GPU data into the host buffer and leveraging the CPU-based MPI routines. It may also involve further tunings by pipelining the transfers and using specifically designed algorithms for some MPI routines. The first step in the general approach involves copying the pertinent data from the GPU global memory into their host memory buffers; next, MPI operation is performed on data that resides on the host buffers, and finally the result is written back to the GPU memory. Depending on the message size, some MPI implementations such as MVAPICH2 may leverage a host-based pipelining design to hide the CUDA memory copy latency or use a more advanced design such as, Fine Grained Pipeline (FGP) algorithm [84] that is proposed for MPI_Allgather. The FGP algorithm exploits simultaneous asynchronous network transfers and CUDA copies in a store and forward fashion.

MVAPICH2-GDR 2.0 is a proprietary design of the MPI standard that leverages the GPUDirect RDMA technology to achieve significant improvement for small message GPU-to-GPU communication. MVAPICH2-GDR takes advantage of the loopback and gdrcopy features in the intranode point-to-point and collective operations for small messages. The loopback design replaces the CUDA memory copy that has an initial calling overhead that is not negligible for short messages; the loopback design provides a virtual network interface allowing the node to communicate with itself. The GDR copy has a non-blocking nature, allowing the transfer to progress in parallel and thus incurring a lower latency compared to the CUDA memory copy used in MVAPICH2.

Current GPU-aware collective operations neither utilize efficient GPU-aware algorithms, nor fully exploit modern GPU features. Moreover, while different data copy mechanisms have been proposed for GPU inter-process communications, efficiently leveraging them in collective operations has not been investigated. The existing GPU-aware collective operations leverage flat designs which are inefficient given the hierarchical structure of many multi-GPU nodes and GPU clusters.

2.4 Modern Interconnects and Communication Channels

GPU inter-process communication plays a crucial role in the performance of the applications running on the GPU clusters. GPU inter-process communications can happen within a single node (known as intranode communication), or across the network (known as internode communication). Both intranode and internode communications have been shown to considerably affect the performance of the applications running on the GPU clusters [70]. In this regard, several high-speed interconnection networks

and communication channels are used to interconnect GPUs within and across the nodes.

According to the Top500 [91], various interconnection networks can be used to interconnect HPC clusters with GPU accelerators, such as InfiniBand [32], Aries [19], Omni-Path [7], 10G Ethernet, and other proprietary interconnects. Among these interconnection networks, InfiniBand has the highest share among the top 500 supercomputers with GPU accelerators [91].

Today, a common practice with many of the GPU clusters is to use multi-GPU nodes to increase their computation power and bandwidth capacity. Inter-process communications within these nodes can go through different communication channels such as PCIe tree topology, NVLink connection [66], and inter-socket links.

In the rest of this section, we overview some of the popular interconnection networks and communications channels that are commonly used in HPC clusters with GPU accelerators.

2.4.1 Interconnection Networks

InfiniBand: The InfiniBand architecture is maintained by the InfiniBand Trade Association (IBTA) [32] which was formed in 1999. InfiniBand (IB) is highly in use among Top500 HPC clusters, specifically those that are equipped with GPU accelerators. This high share is mainly the fruit of its fast inter-process communication.

InfiniBand defines a System Area Network for connecting host nodes and external devices. Host nodes and external devices are connected to the fabric by Host Channel Adapters (HCA) and Target Channel Adapters (TCA), respectively. The interface to the HCAs is defined by a set of standard features known as verbs, while the interface

to the TCAs is vendor specific.

Various vendors are supporting InfiniBand networking technology and have their own implementation of it, such as Mellanox technologies [52] and Intel [34]. Mellanox has added a feature to the InfiniBand verb for managing ordered communications, known as CORE-Direct technology [15]. This feature allows collective operations to be offloaded into the HCAs, releasing the communication processing burden from the CPUs.

IB, in general, has two transfer semantics: channel semantic and memory semantic. In channel semantic, a send/receive approach is used; while in the memory semantic, data is read/written directly from/into the remote process address space without its involvement. The Remote Direct Memory Access (RDMA) feature of IB is used in the memory semantic, which bypasses the operating system and sends the data with no host intervention. The GPUDirect RDMA is a feature that allows the HCA to directly access the remote GPUs without any host intervention. This provides a significant decrease in the GPU inter-process communications across the network. The GPUDirect RDMA is jointly developed by NVIDIA and Mellanox.

2.4.2 Communication Channels

PCIe: PCIe (Peripheral Component Interconnect Express) is a high-speed computer bus used to connect peripheral devices (such as GPU) to the system. Fig. 2.3 shows an example connection of the PCIe devices to the node interconnection topology tree. As shown in the figure, the PCIe tree is a subset of the node topology tree. The root of the topology tree is called root node or root complex, that directly communicates with the socket; in PCIe tree, the leaf nodes are the end-point

PCIe devices that traverse a PCIe switch fabric which itself may consist of multiple PCIe switches. Communications over the PCIe communication channels can go over multiple lanes. In PCIe version 3.0, each lane provides 985 MB/s, thus a 16-lane PCIe 3.0 can effectively provide communication at a rate of around 16 GB/s. PCIe is also full-duplex leading the bi-directional bandwidth to be twice the uni-directional bandwidth.

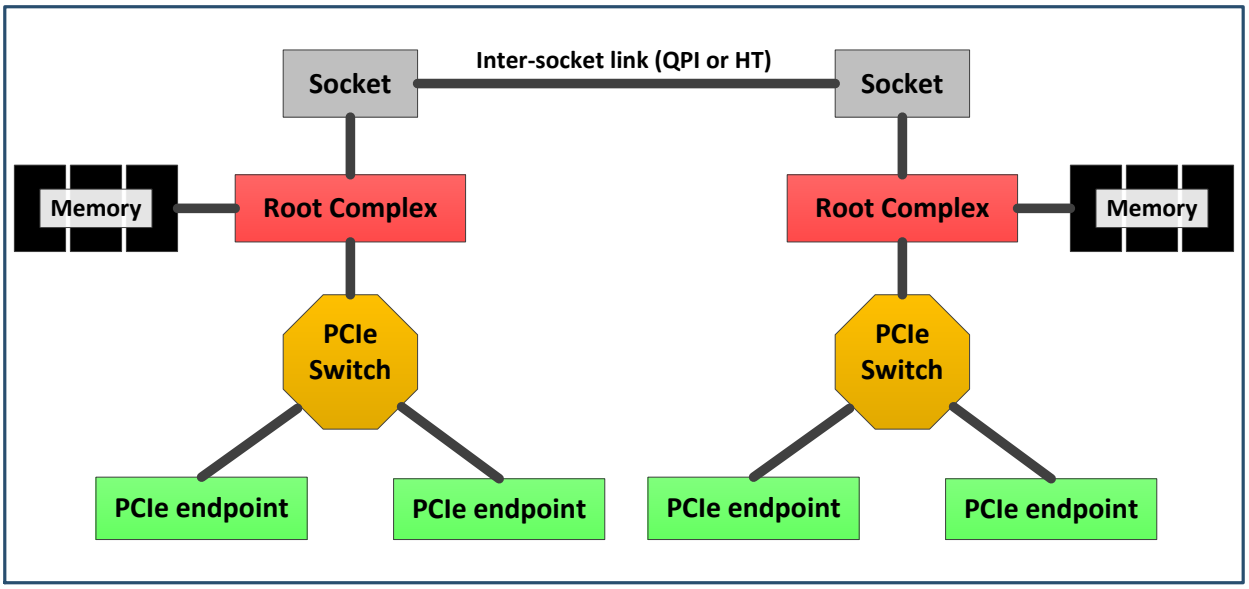


Figure 2.3: The node interconnection, topology tree

NVlink: NVLink [66] is designed as a high-bandwidth communication channel between GPU and GPU or between GPU and CPU within the same node. NVlink provides a common shared address space, allowing the programmer to directly read or write the local GPU global memory, or peer GPU global memory, or CPU main memory. NVLink uses bidirectional connections. Each connection can provide 20 and 40 GB per second uni- and bi-directional peak theoretical bandwidth, respectively.

The Pascal GP100 supports four NVLinks connection, leading to a total of 160 GBps of bidirectional bandwidth.

Inter-Socket Link: Intel Quick Path (QPI) [33] and HyperTransport (HT) [87] links (Fig. 2.3) are used for inter-socket communications in the Intel and AMD processors, respectively. Both QPI and HT are packet-based interconnect technologies that are specially optimized to achieve low latency.

Chapter 3

Efficient GPU Collective Communication Algorithms

In GPU clusters, efficient inter-process communications play a crucial role in the MPI application performance. Considerable portion of the MPI communication time is spent on collective communications in both HPC and deep learning applications. Rabenseifner [75] provides a five-year profiling study of the HPC applications running in production and showed more than 40% of the MPI communication time to be spent on two of the MPI collective operations (i.e., MPI_Allreduce and MPI_Reduce). The Deepbench microbenchmark [16] also lists MPI_Allreduce as one of the three main operations for deep learning applications. Taking this into account, in this chapter, we propose two algorithms to improve the performance of GPU collective operations. The GPU-aware algorithms in this chapter are evaluated using MPI_Allreduce as an example of MPI collective operations. However, our designs with minor changes can also be applied to other MPI collective operations. In this chapter, we make the following key contributions.

- We propose two novel designs for GPU collective operations targeting single

GPU nodes: 1) *GPU Shared Buffer-aware (GSB)*; and 2) *Binomial Tree Based (BTB)*. Both GSB and BTB utilize GPU shared buffer to perform their collective inter-process communications. The GSB design uses an aggregated GPU shared buffer to perform these communications in a first-come first-served order. The BTB design, on the other hand, follows the binomial tree based algorithm discussed to Chapter 2, for its communications, using a pair-wise GPU shared buffer that is distributed among processes.

- We leverage the GPU communication and computation capabilities to further improve the performance of our GPU collective designs. On the computation front, a CUDA kernel function is used to accelerate the collective computation within the GPU. On the communication front, CUDA IPC copies in conjunction with the GPU shared buffer are used to speed up the inter-process communications [23].

- We extend our GSB and BTB collective designs to across the cluster. To this aim, we use our proposed GSB Reduce/Broadcast and BTB Reduce/Broadcast designs at the intranode level, and the existing collective designs for Allreduce at the internode level of the collective communication operation across the cluster.

- We conduct our experiments on two different single-GPU platforms using the OSU microbenchmark [11]. We show that our GPU-aware designs can considerably improve the performance of MPI_Allreduce with large message sizes on both systems. We also provide an evaluation of our designs across the cluster and demonstrate similar findings.

3.1 Related Work

In traditional clusters, using efficient algorithms for MPI operations is a well studied problem in the literature [10, 76, 90]. Brock et al. [10] proposed algorithms for MPI_Alltoall that are mainly optimized with respect to the startup time and data transfer latency. Rabenseifner [76] proposed various algorithms for MPI_Allreduce and MPI_Reduce that are optimized for different message sizes and process counts. Thakur et al. [90] evaluated various algorithms for different MPI collectives on MPICH with the goal of minimizing the latency for short messages and the bandwidth usage for long messages. While these work provide various collective algorithms targeting the CPU domain in homogeneous environments, they may not necessarily be applied in the GPU domain in heterogeneous environments. The CPU and the GPU are different in terms of their architecture, as well as the topology and type of their communication channels. Taking this into account, the work in this chapter evaluates the effect of different algorithms in optimizing collective operations in the context of GPU.

In the GPU domain, the efficiency of exploiting GPU-aware algorithms for MPI operations is evaluated in various studies [85, 84, 12]. Singh et al. [85] optimized MPI_Alltoall by leveraging a pipelining mechanism to overlap device-to-host and host-to-device CUDA memory copies with the network communications. Singh also proposed an approach called Fine Grained Pipeline (FGP) [84] to implement the MPI_Allgather operation. The FGP approach utilizes simultaneous asynchronous network transfers and CUDA copies in a store and forward fashion. It also breaks the data into smaller pieces and operates independently on them in a pipelined manner to provide further overlap. Chu et al. [12] investigated various flat algorithms for

GPU-aware MPI_Allreduce across the nodes. In this chapter, we take on the challenge to incorporate GPU awareness into the MPI collective operations. More specifically, while previous work considers the same collective algorithm for both intranode and internode collective operations, we propose algorithms that are tuned to the intranode level and provide potential extension to across the cluster.

The benefit of using GPU features have been studied for various point-to-point communication. Wang, et al. [97] proposed a design for GPU point-to-point operations that unifies the data movement between the GPU copies and the network (InfiniBand) transfer. This design leverages the GPUDirect and the Unified Virtual Addressing (UVA) features to differentiate the GPU memory from the host memory. The optimization in their work is mainly achieved by pipelining three steps of the communication, GPU-to-host memory transfer, network transfer, and host-to-GPU memory transfer. Shi et al. proposed various techniques to optimize inter-node GPU point-to-point communications on small messages [82]. They utilized designs such as loopback and fastcopy to avoid the costly CUDA memory copy operations involved in the GPU point-to-point communication.

Various studies have also shown the benefit of using GPU features for non-contiguous transfers [96, 38, 104]. Wang et al. [96] leveraged CUDA 2-dimensional memory copy operation (i.e., *cudaMemcpy2D*) for fast in-GPU packing and unpacking the non-contiguous data. Jenkins et al. [38] proposed to use GPU kernels to perform in-GPU packing and unpacking of non-contiguous data. Wu et al. [104] used GPU kernels to perform packing and unpacking operations and GPUDirect to transfer the data over the network. The benefit of using CUDA kernels have been also used in complex and derived datatype processing on the GPU [37, 81].

The GPU features have also been used in various MPI collective operations [97, 72, 73, 39]. Potluri et al. proposed various designs for internode MPI communications between GPU memories using GPUDirect RDMA [72]. The authors, as a part of their work, showed the benefit of using GPUDirect on some MPI collectives across the node. The benefit of using CUDA IPC feature in intranode communications is studied in [73, 39]. However, this feature is only explored for one-sided and two-sided communications. In this chapter, on the other hand, we exploit GPU-aware features to speed up collective computation and communication within the node. To this aim, for the first time, we propose to use in-GPU kernel functions and GPU shared buffer in conjunction with CUDA IPC to improve intranode collective operations.

3.2 GPU-aware Collective Communication Algorithms

GPU support for collective communications has been already added to some of the well-known MPI implementations, such as MVAPICH2, MVAPICH2-GDR [59], and Open MPI [68]. These operations may follow a general approach which involves staging the GPU data into the host buffer and leveraging the traditional CPU-based MPI routines. They may also involve further tunings by pipelining the transfers and using specifically designed algorithms. For example, MVAPICH2 uses specific designs for MPI_Alltoall [85] and MPI_Allgather [84]. However, these GPU collective operations use costly CPU-assisted communications and computations. In this regard, we propose alternative solutions for GPU collective operations. We evaluate different algorithms in our designs and utilize a GPU shared buffer and in-GPU reduction kernels to speed up collective communication and potential computation, respectively. In our designs, we utilize GPU shared buffer to hold the collective pertinent data.

This buffer is a pre-allocated area in the address space of the GPU global memory. This address is accessible by intranode processes and can be used as a shared medium for inter-process communications and storing the collectives pertinent data. As a case study, in this section, we propose two designs for the GPU MPI_Allreduce collective operation that leverages different algorithms : 1) A GPU Shared-Buffer (GSB) aware approach; and 2) A Binomial Tree Based (BTB) approach. We will discuss the different components of our proposed design for GSB and BTB Allreduce, including the GSB/BTB Reduce and the GSB/BTB Broadcast, and how they are integrated into this operation. Such collectives can be used to implement the associated MPI collectives.

3.2.1 GPU Shared Buffer-Aware Design (GSB)

In the GPU Shared Buffer-Aware (GSB) design for MPI_Allreduce, we use an aggregated GPU shared buffer area to gather the pertinent collective data, manipulate the data, and make the collective result available to the designated processes. In our GSB design we use the *fan-in/fan-out* approach, discussed in Chapter 2. The GPU shared buffer in our design is an aggregated space that is allocated in its entirety in the address space of a predefined process. Below we discuss the general stages involved in designing MPI_Allreduce using the GSB approach; it consists of a GSB Reduce and a GSB Broadcast stage.

GSB Allreduce

The GSB Allreduce exploits the GSB design to implement MPI_Allreduce. The GSB Allreduce is implemented by following two general stages: Stage1: GSB Reduce; and Stage2: GSB Broadcast. In the rest of this section, we first introduce the GSB

Reduce and Broadcast components of the GSB Allreduce, and then further discuss the implementation details of the GSB Allreduce design.

Stage1: GSB Reduce The GSB Reduce is used to implement the reduce component of the GSB Allreduce. The GSB Reduce, uses the fan-in algorithm to gather the collective pertinent data into the GPU shared buffer. The gathered data is then reduced inside the GPU shared buffer using a GPU reduction kernel. The reduced data is then read by the root process. It should be mentioned that the GSB Reduce can be used to implement MPI_Reduce.

Stage2: GSB Broadcast The GSB Broadcast is used to implement the broadcast component of the GSB Allreduce. In the GSB Broadcast stage, all processes using the fan-out algorithms read the collective data from the GPU shared buffer. With the root process first copying the collective data into the GPU shared buffer, it is evident that the GSB Broadcast can also be used to implement MPI_Broadcast operation.

Fig. 3.1 shows the different components of the GSB Allreduce design. All intranode processes copy their pertinent data from their GPU send buffers in their address space in the GPU global memory to the GPU shared buffer to complete the gather stage. Once all the data is available in the GPU shared buffer, the reduction operation takes place on the aggregated data and the result is stored back into the GPU shared buffer. Upon availability of the result, all processes copy it into their own GPU receive buffers.

According to the Fig. 3.1, the GPU shared buffer in its entirety is allocated in the address space of a predefined process (without loss of generality, we assume process with `rank 0` as the predefined process). Processes exploit CUDA IPC to communicate through the GPU shared buffer region. Processes on the same node also have access

to a shared directory which keeps track of the IPC copies into the GPU shared buffer and report their completion. This buffer can be allocated either on the GPU global memory or on the host memory. We have evaluated both design alternatives and decided to keep the directory on the host memory. We provide some justifications for this decision later in this section.

The size of the shared directory is `intra_comm_size` bits, representing the number of processes on the node. Each of the first `intra_comm_size - 1` bits (`copy flags` - see Fig. 3.1) is associated with one process rank and is set once this process initiates its IPC copy into the GPU shared buffer. The last bit in the directory (`completion flag`) indicates the completion of the collective operation and the availability of the results in the GPU shared buffer. This bit is set by the pre-defined process with `rank 0`.

The GPU shared buffer area should be sufficiently large to hold the gathered dataset from all intranode processes (256 MB is used in our experiments). This is directly related to the number of processes/GPUs per node, as well as the size of the dataset that is typically in-use in applications leveraging MPI_Allreduce. As such, the size of the allocated GPU shared buffer is much less than the amount of global memory available in modern GPUs (5 GB on Kepler K20 and 12 GB on Kepler K80). Therefore, this is not a scalability concern in our design.

Fig. 3.1 shows the general steps of the GPU shared-buffer aware MPI_Allreduce design, as follows:

`Step 1.` All processes copy their share of data from their send buffers into their associated addresses in the GPU shared buffer area.

`Step 2.` All processes (except `process 0`) set their associated `copy flags` in the

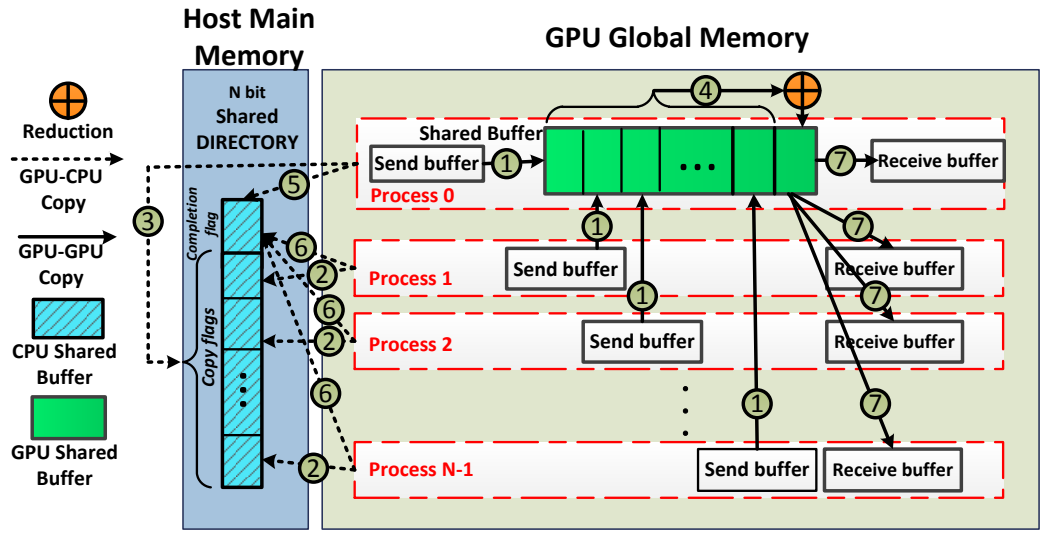


Figure 3.1: Steps of the GPU shared-buffer aware approach for MPI_Allreduce

shared directory after initiating their IPC copies.

`Step 3.` `Process 0` waits on all `copy flags` to be set (this step can overlap with `Step 2`).

`Step 4.` Once all `copy flags` are set, all pertinent data is available in the GPU shared buffer. `Process 0` then performs an in-GPU element-wise reduction on the aggregated data and stores the result in a predefined location inside the GPU shared buffer.

`Step 5.` Once the collective result becomes available in the GPU shared buffer, `process 0` toggles the `completion flag` to indicate the completion of the collective operation.

`Step 6.` All processes (except `process 0`) query the `completion flag` (this step can overlap with `Steps 3, 4,` and `5`).

`Step 7.` Once the `completion flag` is toggled, all processes copy their share of collective result into their respective receive buffers.

Note that in `Step 5`, the `completion flag` has to be toggled in each instance of MPI_Allreduce call; otherwise, successive MPI_Allreduce calls may end up reading stale data.

Implementation Details:

The GPU shared-buffer aware approach leverages pre-allocated CPU and GPU shared buffers. The CPU shared memory region is allocated during MPI initialization stage, `MPI_Init()`, and is attached to the address space of the intranode processes. The memory handle of the GPU shared buffer is also created, broadcast, and mapped to the address space of other processes during this stage. This is performed only once to mitigate the costly operation of exposing the GPU shared buffer address space.

The IPC copies in CUDA have an asynchronous behavior, even if synchronous CUDA memory copy operations are issued on the shared region. Therefore, the `copy flags` in the shared directory only indicate the initiation of the IPC copies, but not their completion. To guarantee the completion of the IPC copies, we leverage the IPC events. The IPC event can be shared and used among processes residing on the same node. To share an IPC event, the handle of the allocated event is created and passed on by the predefined process to the other processes. To guarantee the completion of the IPC copies, an IPC event using `cudaEventRecord()` command is immediately recorded after each IPC copy (`Step 1` in Fig. 3.1). Recording an event is then followed by setting the associated `copy flag` in the shared directory in `Step 2`. Once each `copy flag` is set, `process 0` issues `cudaStreamWaitEvent()` on the associated event until the inter-process copy completes. The `process 0` will wait on all IPC events, by checking the `copy flags` that are tagged as unfinished in a round-robin fashion; The `process 0` will also tag each copy as complete once it finishes.

This way, the completion of all IPC copies can be guaranteed. Finally, all allocated shared buffers and events are freed within the `MPI_Finalize()` call.

3.2.2 GPU-Aware Binomial Tree Based Design (BTB)

The BTB approach utilizes the *binomial* algorithm (Chapter 2) in conjunction with GPU shared buffer to perform collective operations. The GPU shared buffer in the BTB approach (unlike the GSB approach) is distributed among processes involved in the collective operation.

BTB Allreduce

The BTB Allreduce follows two stages to implement MPI_Allreduce using the BTB design: Stage1: BTB Reduce; and Stage2: BTB Broadcast. Below, we discuss the general stages involved in designing MPI_Allreduce using the BTB approach; it consists of a BTB Reduce and a BTB Broadcast stage; In the following, we first introduce these stages, and then further discuss the implementation details of the BTB Allreduce design.

Stage1: BTB Reduce The BTB Reduce implements the reduction operation by following the binomial algorithm. In each level of the algorithm, processes copy their collective pertinent data into the GPU shared buffer of their peer process. The received and the peer process data are then reduced inside the GPU shared buffer using a GPU reduction kernel and the algorithm proceeds to the next level. In the BTB Reduce, the binomial tree is traversed from the leaf to the root process.

Stage2: BTB Broadcast To implement the broadcast component of the BTB Allreduce, all processes, using the the binomial tree algorithm read their collective

data from the GPU shared buffer. The binomial tree in the BTB broadcast is traversed from the root to the leaf. With the root process first copying the collective data into the GPU shared buffer, the BTB Broadcast can also be used to implement MPI_Broadcast operation.

Fig. 3.2 shows the detailed steps involved in the BTB Reduce stage followed by the BTB Broadcast stage of the BTB Allreduce. Considering that the steps involved in different levels of the collective operation are similar, we only discuss the first and last levels of the BTB reduce part of our design.

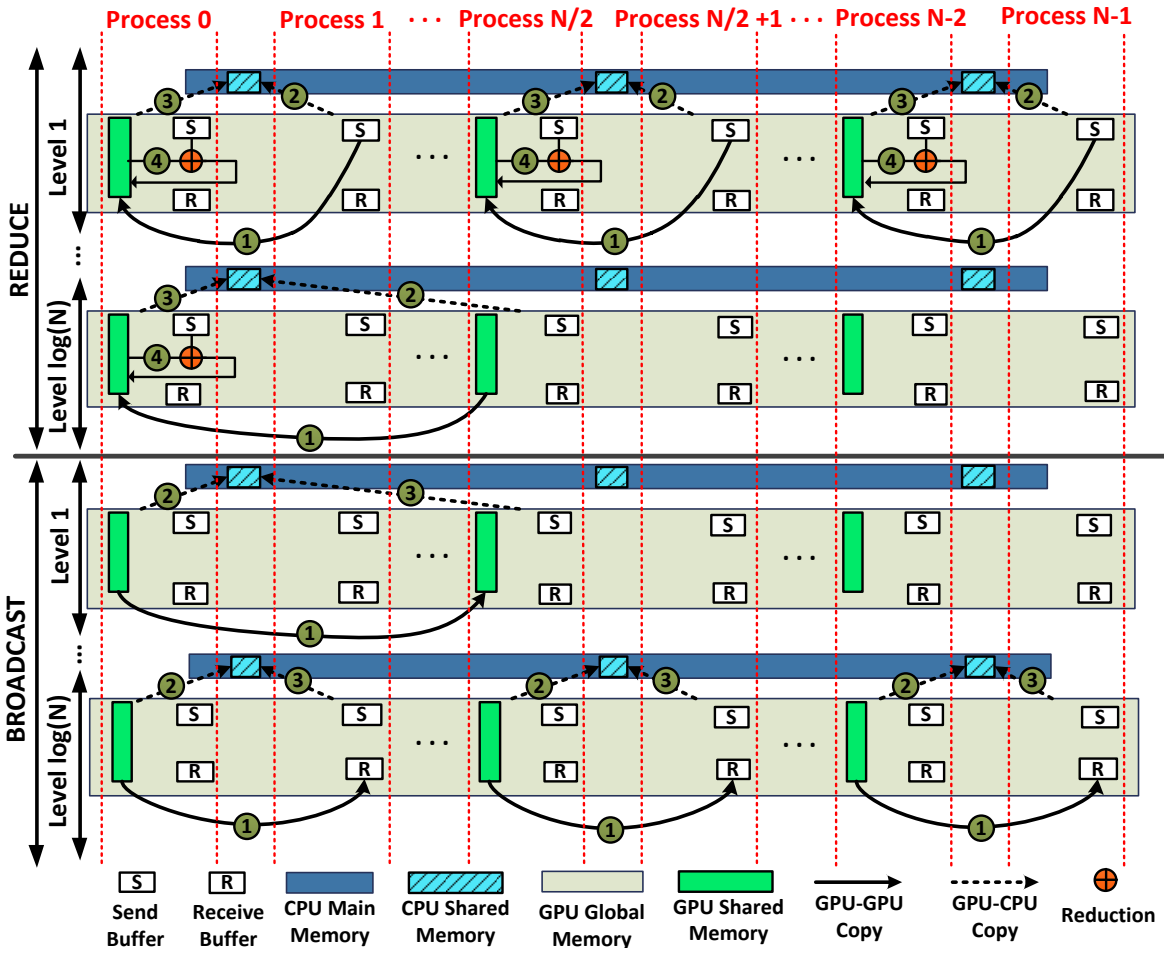


Figure 3.2: Steps of the GPU-aware MPI_Allreduce using the BTB design

BTB Reduce Stage-Level 1:

`Step 1.` First, each odd-ranked process uses IPC to copy the contents of its send buffer to the GPU shared buffer of its adjacent, even-ranked peer process.

`Step 2.` Odd-ranked processes set their associated copy flag in the directory (allocated on the CPU shared buffer) after initiating their IPC copies.

`Step 3.` Even-ranked processes query the flag in the directory to check the initiation of the IPC copy from their peer process.

`Step 4.` Once the flag is set, each even-ranked process performs an element-wise reduction between the data in its GPU send buffer and the pertinent data in its GPU shared buffer. The reduced result is stored back into the GPU shared buffer.

This algorithm then proceeds to the next levels of the binomial tree. In each level, the distance between the peer processes is doubled compared to the previous level. This algorithm terminates when the last binomial tree level (i.e., *Level log(N)*) is processed. The steps of the final level is as follows:

BTB Reduce Stage-Level log(N):

`Step 1. Process N/2` IPC copies the content of its GPU shared buffer into the GPU shared buffer of its peer process (i.e., `Process 0`).

`Step 2. Process N/2` sets its corresponding flag in the directory after initiating its IPC copy.

`Step 3. Process 0` queries the directory flag.

`Step 4.` Once the flag is set, `process 0` performs the final reduction on the pertinent data into the GPU shared buffer.

BTB Broadcast Stage

In the Broadcast Stage, `process 0` broadcasts the reduced data from its GPU

shared buffer to the rest of the processes using the BTB-based broadcast algorithm. In the broadcast step, IPC copy and setting/querying the directory are performed similar to the reduce step. However, unlike the reduce step, the distance between the peer processes is halved in each step.

Implementation Details

The BTB design, similar to the GSB design, utilizes both GPU and host shared buffers. The GPU shared buffer is allocated by only half of the participating processes. Thus, the size of the shared buffer in the BTB approach is half the size of the shared buffer in the GSB approach. The directory is allocated on the host memory and its size is also half the size of the directory in the GSB approach. The memory and event handles are created and passed along during the MPI initialization time. In the BTB design, we guarantee the order of the communications and computations by enforcing processes on the same tree level to only communicate with each other. To this aim, we store the tree level of the sending process in its associated entry in the shared directory. This way, the sending and receiving processes can check and match their tree levels before initiating the IPC copy. Matching the tree level indicates that all of the copies in previous tree levels have been already completed. Consequently, the ordered communication/computation can be guaranteed, preventing any potential race condition.

Other Design Considerations

Both GSB and BTB designs allocate the shared buffer on the GPU global memory while the directory is kept on the host main memory. In the first glance, having the directory on the GPU memory with a kernel function querying its entries seems to be justified; however, this can potentially lead to spin-waiting on the directory forever,

as the process querying the directory will take over the GPU resources and would prohibit other processes to access them. We tried to address this issue by forcing the querying process to release the GPU in time-steps. However, the performance results were not promising and selecting the appropriate value for the time-step was dependent on many factors such as message size and process count.

We also tried to query the directory using CUDA asynchronous copy operations. Though this approach was feasible, it had high detrimental effect on the performance. The performance slowdown is basically due to the high number of asynchronous copy calls issued by the querying processes. These calls have to be synchronized at the beginning of each MPI_Allreduce invocation. Synchronization calls are costly, as they require waiting on all previously issued copies on the directory to complete. Avoiding synchronization calls can result in accessing stale data on the directory, which were stored in the previous invocation of MPI_Allreduce. This can ultimately result in inaccurate directory checking. Taking everything into consideration, to achieve the highest performance, we decided to allocate the directory on the host main memory, while keeping the shared buffer on the GPU global memory.

3.2.3 Cluster-wide Extension of the GSB and BTB Algorithms

To extend the GSB (or BTB) design to across the cluster, we break the collective operation into intranode intra-GPU and internode steps. For MPI_Allreduce, we extend the GSB or BTB design to across the cluster by following the steps below. In the first step, the GSB Reduce (or alternatively BTB Reduce) design is used to perform the intranode intra-GPU reduce and store the result in the address space of the *leader process*. Next, MVAPICH2 MPI_Allreduce is performed across the nodes

and among the leader processes. In the final step, the leader processes perform an intranode intra-GPU broadcast using the GSB Broadcast or BTB Broadcast design. This is an example of using a hierarchical design for GPU collective operations; in Chapter 4, we comprehensively study a hierarchical framework for GPU collective operations. In particular, we discuss how a hierarchical framework can be applied to different collective operations and evaluate the sensitivity of different collective algorithms to different hierarchy levels in multi-GPU nods and clusters.

3.3 Experimental Results and Analysis

3.3.1 Experimental Platform

In this section, single-node experiments are performed on two different GPU nodes: 1) K20 GPU node; and 2) K80 GPU node. The K20 GPU node (System A) is a 16-core node that is equipped with an NVIDIA Kepler K20M GPU. This node has a dual socket Intel XEON E5-2650 clocked at 2.0 GHz, a 64 GB of memory, and is running a 64-bit Red Hat Enterprise Linux 6.2 and CUDA Toolkit 5.5. The K80 GPU node is a part of the Helios K80 cluster (System B) that is installed at Université Laval provided by Compute Canada and Caclul Québec. This node has eight K80 GPUs, 256 GB of memory, and two Intel Xeon Ivy Bridge E5-2697 v2 processors. Each of the Xeon processors provides twelve cores, operating at 2.7 GHz clock speed. Thus, there exists a total of 24 cores per node. Moreover, this node runs a 64-bit CentOS 6.7 and CUDA Toolkit 7.5. For our cluster-wide experiments we use four K80 nodes. These nodes have identical configuration and use QDR InfiniBand as their interconnect.

Microbenchmark Study

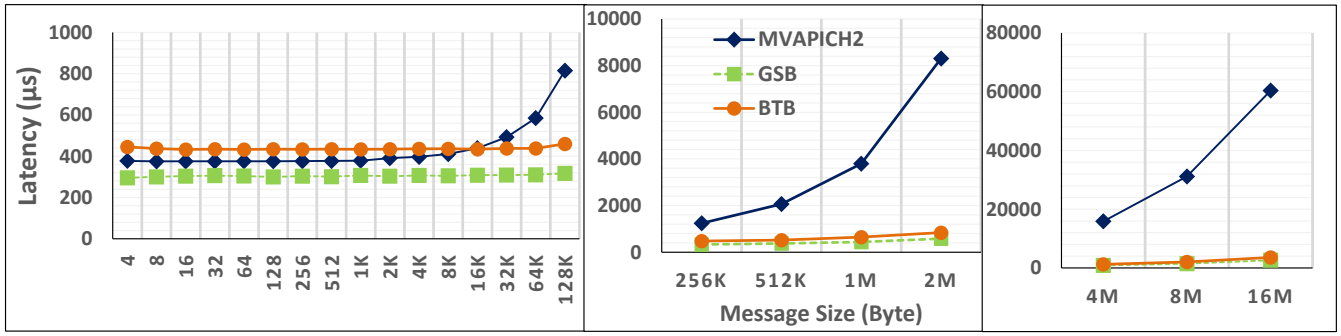
The experimental results in this chapter compare our proposed designs against the default MVAPICH2 design. While our designs can be implemented in and compared against any MPI library libraries, we opt to use MVAPICH2 library as it provides GPU-specific designs [85, 84] for collective operations. For this comparison, we use the OSU microbenchmark suite configured to support GPUs [11]. This suite provides a set of microbenchmarks to evaluate the performance of MPI point-to-point and collective operations. We utilize the collective benchmark from this suite to measure the latency of MPI_Allreduce operation. This benchmark reports the average latency of the MPI_Allreduce operation over a large number of iterations (100 to 1000 iterations depending on the message size) across different number of processes and message sizes. More specifically, on the K20 GPU node we use the OSU microbenchmark to evaluate our proposed designs against MVAPICH2-1.9 on 4 Byte to 16 MB message sizes and on 4, 8, and 16 processes. On the K80 GPU node, we use the OSU microbenchmark to compare our proposed designs against MVAPICH2-2.1, and present the results for 4 Byte to 16 MB message sizes and on 4, 8, 16, and 24 processes; we use the same configuration per node for our cluster-wide analysis. Noteworthy to mention, we get consistent results across different runs as ordered communications and computations in our designs would prevent the potential of any round-off errors or race conditions.

3.3.2 Single-Node Single-GPU Results

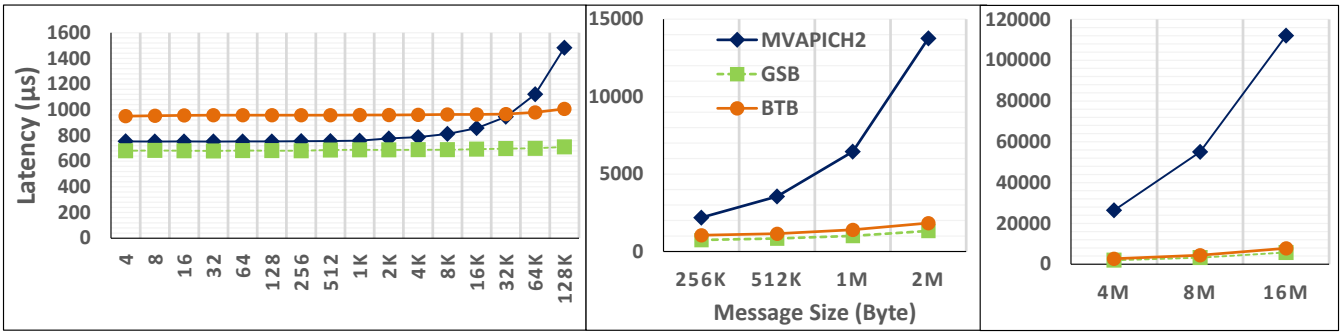
Fig. 3.3 and Fig. 3.4 compare MPI_Allreduce using our proposed GSB and BTB approaches against the MVAPICH2 design on a K20 and a K80 GPU node, respectively. According to Fig. 3.3, the GSB approach outperforms MVAPICH2 for all message sizes on 4 and 8 processes on K20 GPU; with 16 processes, the benefit of the GSB

starts at message sizes greater than 16KB. The proposed designs on this node show up to 22 times speedup over MVAPICH2. According to Fig. 3.4, the benefit of our proposed designs starts at 256 KB message size on the K80 GPU. Using our proposed designs, we can observe up to 19.5 times performance improvement for large messages over the MVAIPCH2 MPI_Allreduce. This is because MVAPICH2 uses host-based data staging and reduction which are costly specifically for large message sizes. On the other hand, our designs utilizes a GPU shared buffer to perform direct IPC copies and in-GPU reductions. The startup and the peer synchronization of the CUDA IPC copies impose high overhead in copying short message sizes; however, as the message size increases the startup overhead becomes negligible compared to the data transfer time.

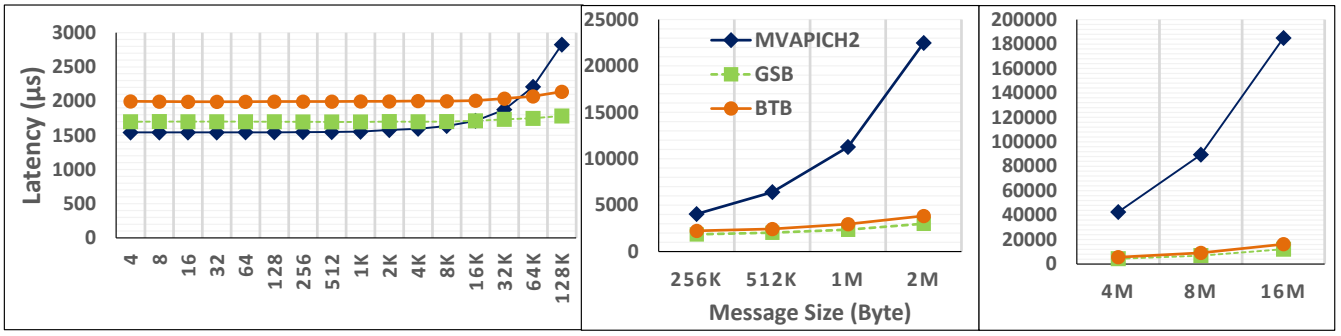
As shown in Fig. 3.3 and Fig. 3.4, the performance of the GSB approach is superior or at least similar to the BTB approach. This indicates that on a single-GPU platform, the logarithmic nature of the binomial algorithm cannot provide any improvement over the GPU Shared Buffer-aware approach with linear complexity. The performance results indicate a consistent behavior in the GSB and BTB approach for message sizes up to 128KB. In this range, the startup latency for the IPC copy and the in-GPU reduction kernel is almost oblivious to the message size and mainly dominates the collective runtime. However for larger message sizes, the reduction and the IPC copy time becomes mainly dependent on the message size. More specifically, the message size and the number of IPC calls to/from the shared buffer mainly determine the total execution time of large message sizes in our designs. These IPC copies cannot overlap with each other, thus increasing the number of processes in our design increases the total execution time accordingly.



(a) Single K20 GPU node - 4 Processes

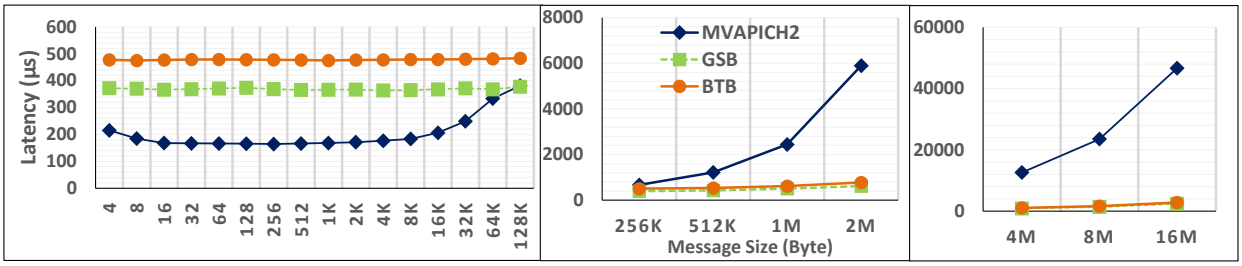


(b) Single K20 GPU node - 8 Processes

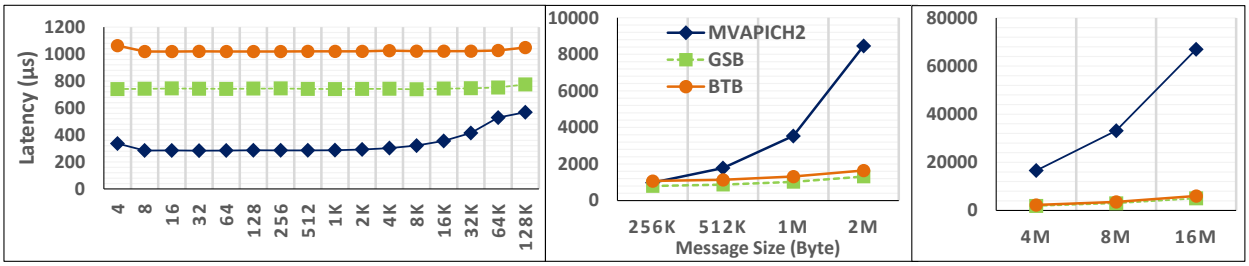


(c) Single K20 GPU node - 16 Processes

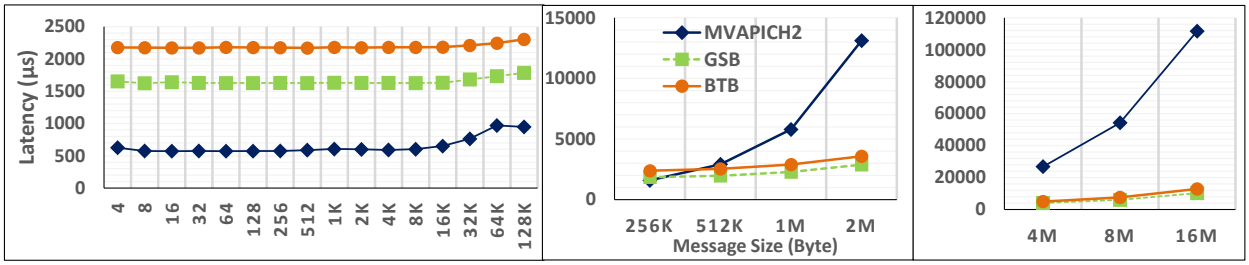
Figure 3.3: MVAPICH2 vs. GSB MPI_Allreduce vs. BTB MPI_Allreduce on a single K20 node (System A) with a single GPU



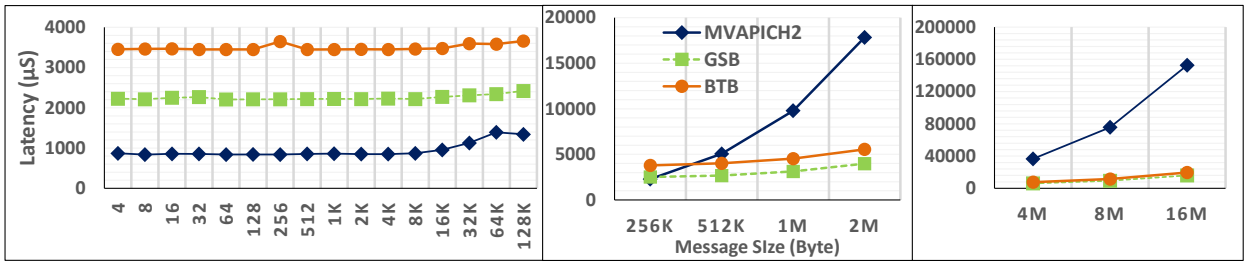
(a) Single K80 GPU node - 4 Processes



(b) Single K80 GPU node - 8 Processes



(c) Single K80 GPU node - 16 Processes



(d) Single K80 GPU node - 24 Processes

Figure 3.4: MVAPICH2 vs. GSB MPI_Allreduce vs. BTB MPI_Allreduce on a single K80 node (System B) with a single GPU

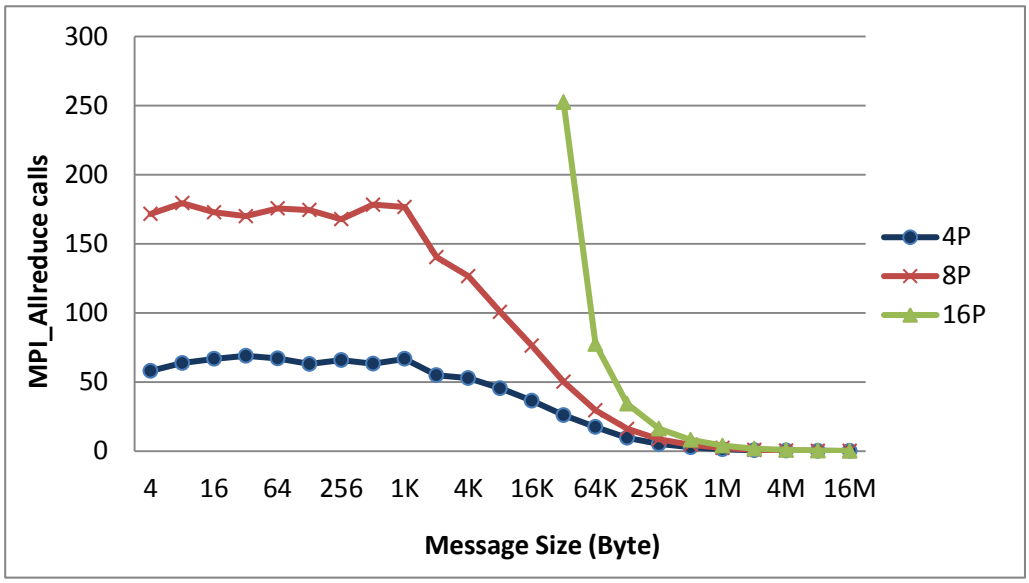


Figure 3.5: Number of MPI_Allreduce calls required to compensate for the initialization overhead

3.3.3 Initialization Overhead:

Recall that in our designs, both of the host and GPU shared buffers are created during `MPI_Init()` and freed during `MPI_Finalize()`. For an application to benefit from the proposed designs, the one-time cost of this initialization overhead must be amortized. Fig. 3.5 shows the number of MPI_Allreduce calls required to compensate this initialization overhead in the GPU shared-buffer aware approach on the K20 GPU node. As can be seen, for large message sizes, a few MPI_Allreduce calls (only a single call, in most cases) are just required to compensate for this overhead. For smaller messages, more calls are required. Note that, for the 16-process case, we only provide the results for message sizes above 16KB, as the benefit of our approach on 16 processes starts at messages greater than 16KB (Fig 3.3.c).

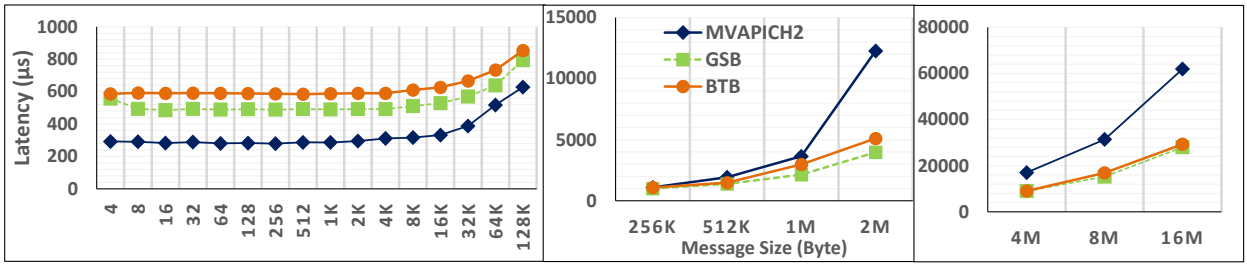
3.3.4 Cluster-Wide Results

In this section, we evaluate the GSB and BTB designs on a 4-node Helios cluster (System B) with a single GPU per node using MPI_Allreduce. Fig. 3.6 compares the GSB and BTB Allreduce with the MVAPICH2 MPI_Allreduce on a 4-node cluster. Our results show a similar trend with what we have observed earlier in our single-node experiments (Fig. 3.4). According to our cluster-wide results (Fig. 3.6), the benefit of the GSB Allreduce starts at 256 KB. The GSB approach also outperforms the BTB approach across all message sizes, while the performance gap between the two approaches reduces as the message size increases.

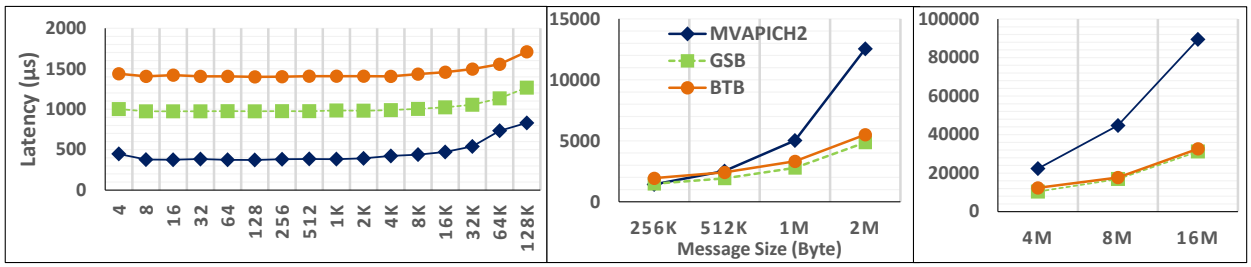
3.4 Summary

In this chapter, we investigated GPU-aware algorithms collective operations. We proposed two design alternatives, called GPU Shared Buffer-aware (GSB) and Binomial Tree Based (BTB). Both designs use IPC copies and in-GPU reductions, along with GPU and host shared-buffers to speedup collective performance. We evaluated our proposed designs using MPI_Allreduce on single-GPU platforms and showed that the GSB approach provides the highest improvement. In general, the GSB and the BTB Allreduce provide significant speedup for large message sizes over MVAPICH2 MPI_Allreduce by up to 22 and 16 times on a single node, respectively. The GSB and the BTB Allreduce also showed up to 5 times improvement over the MPI_Allreduce across the clusters.

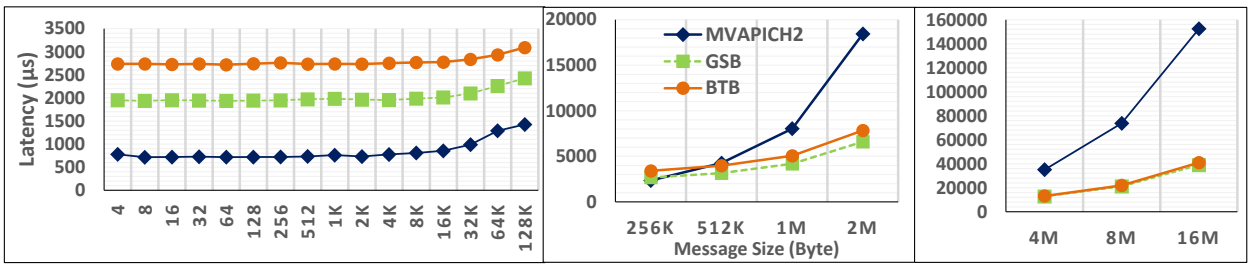
Our proposed designs in this chapter target single-GPU nodes and clusters; in Chapter 4, we propose a hierarchical framework for GPU collectives targeting clusters of multi-GPU nodes. In Chapter 3, while we observed considerable performance



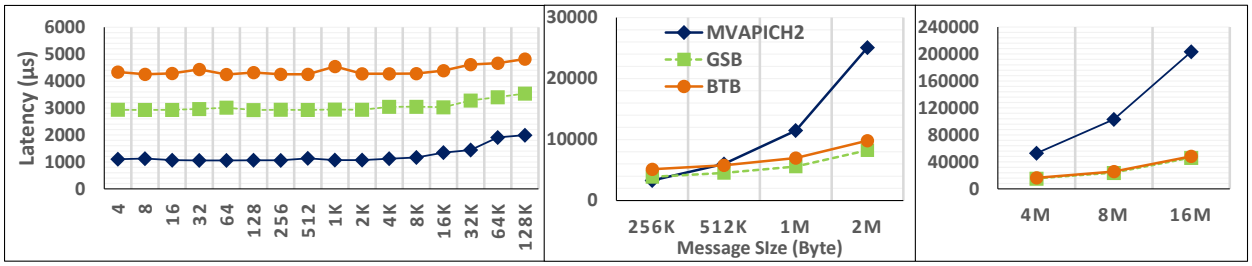
(a) 4 K80 nodes with single GPU per node - 16 Processes (4 processes per GPU)



(b) 4 K80 nodes with single GPU per node - 32 Processes (8 processes per GPU)



(c) 4 K80 nodes with single GPU per node - 64 Processes (16 processes per GPU)



(d) 4 K80 nodes with single GPU per node - 96 Processes (24 processes per GPU)

Figure 3.6: MVAPICH2 vs. GSB MPI_Allreduce vs. BTB MPI_Allreduce on 4 K80 node Helios cluster (System B) with single GPU per node

improvement on large messages, our designs lead to performance degradation for small and medium message sizes. In Chapter 5, we propose designs that can make informed decisions in selecting the right data copy mechanism in their collective operations to address small and medium size messages.

Chapter 4

Hierarchical Framework for GPU Collective Communications

Today many of the homogeneous HPC clusters have hierarchical structures, which usually consists of intranode hierarchical memory subsystem along a hierarchical interconnection network. In this regard, collective communications on these clusters usually utilize different algorithms at different hierarchy levels to enhance the performance.

GPU accelerators, in recent years, have been adopted in HPC clusters to reach higher compute density and power efficiency. Such clusters usually exploit multi-GPU nodes to reach even higher compute power and larger GPU memory capacity. Such heterogeneous clusters add to the hierarchical nature of clusters. Thus, GPU interprocess communications in these clusters can take place within different hierarchy levels, such as within a single GPU, across intranode GPUs, or across the network.

Given the hierarchical structure of the GPU clusters, in this chapter we extend our initial study in Chapter 3 and investigate various hierarchical solutions for GPU collectives running on clusters with multi-GPU nodes. In this regard, we propose a

general hierarchical framework for GPU collective operations. Our proposed framework for different collectives within the node or across the cluster may consist of up to three hierarchical levels, intranode intra-GPU, intranode inter-GPU and internode inter-GPU. The intention of this framework is to highlight the importance of selecting the right algorithm at each hierarchy level for the GPU collective operations. In this regard, we evaluate the effectiveness of using different GSB and BTB algorithms proposed in Chapter 3 on different hierarchy level in our framework and discuss our findings.

In general, our framework can be applied to any collective operation. In this chapter, as a test case scenario, we evaluate our designs on MPI_Allreduce operation. We conduct our experiments on a multi-GPU cluster and provide our results on up to 32 GPUs. In summary, we make the following key contributions.

- We propose a hierarchical framework for MPI collective operations targeting multi-GPU nodes and multi-GPU clusters. We show that given the hierarchical nature of the GPU clusters with multi-GPU nodes, hierarchical solutions can significantly improve the performance of GPU collective operations.

- We analyze the impact of different algorithms within the intranode hierarchy levels of our proposed framework and discuss our findings. In general, we show that while a collective communication algorithm might be a good fit in one hierarchy level, it may not be favored in another [22].

- We conduct our experiments within a multi-GPU node and across the GPU cluster with up to 32 GPUs. Our experimental results show that our hierarchical GPU-aware designs can significantly outperform the existing designs for large message sizes.

4.1 Related Work

In traditional clusters, utilizing hierarchical solutions for MPI collective operations is a well studied problem [43, 40, 86, 93, 80, 92, 28, 49, 74, 94, 46]. Researchers have proposed hierarchical solutions for collective operations on wide-area distributed clusters [43, 40]; the improvements achieved by these works are the result of minimizing communications over the slow wide-area links, while increasing the communications over the faster local links.

Hierarchical solutions have been also proposed for clusters of SMPs [86, 93, 80]. In these work, the intranode part of the collective operations are improved by utilizing intranode shared memory communication. Shared memory is also used in [28, 49] for specifically designing intranode collective operations. Graham and Shipman [28] evaluated various implementation options for shared memory-based collectives within a node. They showed the benefit of their shared-memory based collectives over the point-to-point-based collectives that only utilize shared memory within the transport layer. Mamidala et al. [49] proposed shared memory-based collectives designs that are aware of the multi-core aspect of the clusters. They evaluated various architectural attributes of the AMD and Intel processors that were used for data transfer on multi-core nodes. They utilized these insights to develop different intranode collective operations. Qian and Afsahi [74] proposed hierarchical designs for MPI_Alltoall and MPI_Allgather for multi-core SMP clusters. The authors take into account both the system shared memory and the arrival pattern of the collective processes in their designs.

Traff and Rougier [94] discussed the use of MPI 3.0 functionality that reflect the hierarchy of the cluster in implementing hierarchical collectives. The authors

concluded that utilizing MPI 3.0 hierarchy-aware features can be an effective and portable means for application developers to implement their own (non-MPI) hierarchical collectives. Li et al. [46] also proposed shared memory-aware collective designs to avoid the intermediate copies involved in point-to-point-based and shared-memory based collectives. To this aim, they proposed alternative thread-based designs for the typical process-based collective operations. This way, different threads involved in the collective operation can share their address space and communicate with each other without any intermediate copy. The authors also considered the NUMA effect in their collective design. The performance of their thread-based collectives showed to significantly outperform the process-based designs. However, the process-based MPI applications need to be modified and become thread-based. While these work study the impact of hierarchical designs on homogeneous CPU clusters, in this chapter we propose hierarchical solutions for GPU clusters with multi-GPU nodes.

Some researchers have studied various collective operations for GPU clusters. Internode GPU collectives have been studied by various researchers [85, 84, 72, 12]. These studies utilize a combination of different GPU-aware algorithms and the underlying hardware features to improve collective operation across the node. However, none of these work take into account the hierarchical structure of the clusters with multi-GPU nodes. In this chapter, on the other hand, we propose a hierarchical framework for GPU collective operations. Using this framework, we break down the collective operation into different hierarchy levels. We also evaluate the sensitivity of different algorithms on different hierarchy levels and accordingly suggest the best combination for GPU collective operations.

4.2 Hierarchical Collective Framework for a Multi-GPU Node and GPU Clusters

4.2.1 Designs for a Multi-GPU Node

In a multi-GPU node, processes can be assigned to different GPUs or share a single one. The intranode intra-GPU communications have different characteristics compared to the intranode inter-GPU communications; for example, the intranode intra-GPU communication bandwidth is at least an order of magnitude higher than the intranode inter-GPU communications. Moreover, multiple intranode intra-GPU communications, unlike intranode inter-GPU communications, would serialize. The existing collective designs as well as the proposed designs in Chapter 3 when applied to a multi-GPU node would consider a flat design which is oblivious to these differences. Taking these into consideration, we propose a hierarchical communication framework for multi-GPU nodes consisting of two general hierarchies: 1) an intranode intra-GPU communication hierarchy; and 2) an intranode inter-GPU communication hierarchy. We also investigate the efficiency of the GSB and BTB algorithms proposed in Chapter 3 within these hierarchy levels. Below, we discuss how our proposed hierarchical framework can be applied, as a case study, to MPI_Allreduce.

MPI_Allreduce

To apply our proposed hierarchical communication framework to MPI_Allreduce, we propose a hierarchical design for MPI_Allreduce collective operation in multi-GPU nodes that consists of two stages: Stage1) Reduce; and Stage 2) Broadcast. In the rest of this section, we discuss the Reduce Stage and the Broadcast Stage, and explain how our framework is applied to them.

Stage1 Reduce:

Step1-Intranode Intra-GPU: All processes sharing the same GPU use the GSB Reduce (or BTB Reduce) algorithm to reduce the data into the GPU shared buffer of their *leader process*.

Step2-Intranode Inter-GPU: All the intra-GPU leader processes use the GSB Reduce (or BTB Reduce) algorithm to further reduce the data into the GPU shared buffers of the root process. Next, the root process reads the reduced data into its receive buffer. Noteworthy to mention, the reduce stage can be used to implement MPI_Reduce as well.

Stage2 Broadcast

Step1-Intranode Inter-GPU: The root process uses the GSB Broadcast (or BTB Broadcast) algorithm to broadcast the collective data into the GPU shared buffer of the leader process in other intranode GPUs.

Step2-Intranode Intra-GPU: All the leader processes use the GSB Broadcast (or BTB Broadcast) algorithm to broadcast the data into the GPU shared buffers of all processes within the same GPU. It is evident that the Broadcast Stage can be used to implement MPI_Broadcast as well.

Fig. 4.1 shows the general steps involved to apply our proposed hierarchical framework into MPI_Allreduce targeting a multi-GPU node. According to the figure, in Step 1 of the Reduce Stage, intranode intra-GPU processes reduce their pertinent data and store it in the GPU shared buffer of their pre-defined leader process (without loss of generality, process with rank 0 is considered as the leader process). In Step 2 of the Reduce Stage, the reduce operation is performed among the intra-GPU leader processes within the node and the result is stored in the pre-defined *node leader process*. The result of this operation is then used in the Broadcast Stage. In Step 1

of the Broadcast Stage, the node leader broadcasts the data into the leader process in each GPU within the node. In Step 2 of the Broadcast Stage, the leader process in each intranode GPU broadcasts the data to the rest of the intranode intra-GPU processes.

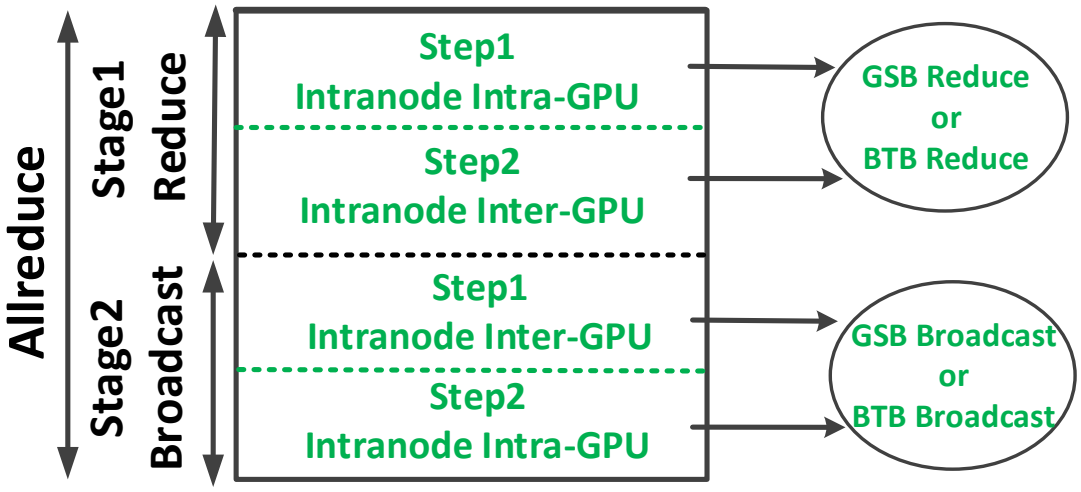


Figure 4.1: Steps of the hierarchical GPU-Aware MPI_Allreduce on a multi-GPU node

To investigate our hierarchical framework, we propose four design alternatives on a multi-GPU node: 1) GSB-GSB; 2) GSB-BTB; 3) BTB-GSB; and 4) BTB-BTB. In each scenario, the first term determines the algorithm for the intranode intra-GPU step and the second term determines the algorithm for the intranode inter-GPU step. As an example, the GSB-BTB design selects the GSB and the BTB algorithms for the intranode intra-GPU and the intranode inter-GPU steps, respectively. More specifically, GSB-BTB means GSB Reduce followed by BTB Reduce for the reduce stage, and BTB Broadcast followed by GSB Broadcast for the Broadcast stage.

Fig. 4.2 and Fig. 4.3 illustrate the general steps involved in performing the Reduce

stage of the MPI_Allreduce, as shown in Fig. 4.1, using the GSB-GSB and the GSB-BTB designs, respectively. According to these figures, both designs perform a GSB Reduce in their intranode intra-GPU level. For the intranode inter-GPU level, the GSB Reduce algorithm is used in the GSB-GSB design (Fig. 4.2), while the BTB Reduce is used in the GSB-BTB design (Fig. 4.3). Fig. 4.4 shows the general steps involved in performing the Broadcast stage of the MPI_Allreduce, as shown in Fig. 4.1, using the GSB-GSB algorithm.

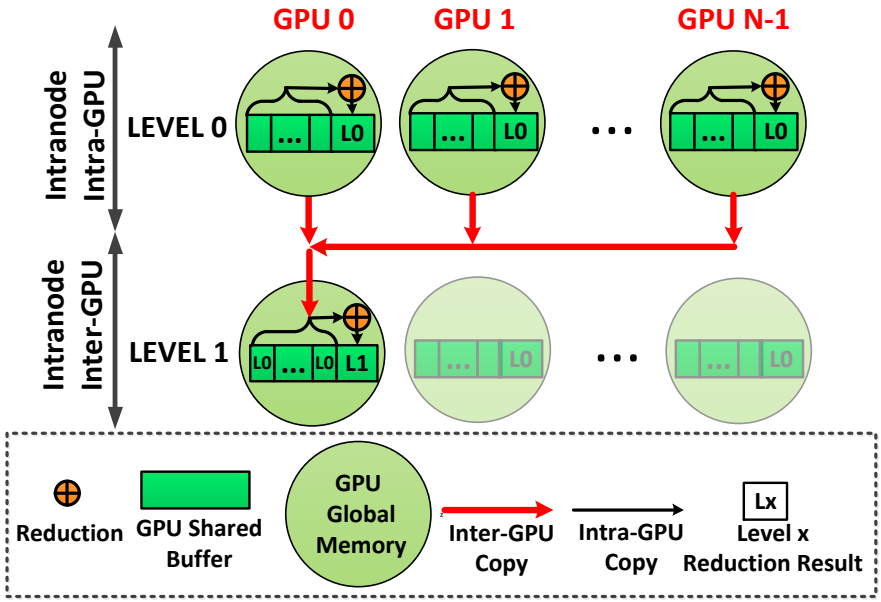


Figure 4.2: Hierarchical MPI_Allreduce utilizing Intranode Intra-GPU GSB Reduce and Intranode Inter-GPU GSB Reduce algorithms - Reduce stage

4.2.2 Designs for a GPU Cluster

We propose a hierarchical communication framework for clusters of multi-GPU nodes consisting of three general hierarchies: 1) an intranode intra-GPU communication hierachy; 2) an intranode inter-GPU communication hierachy; and 3) an internode

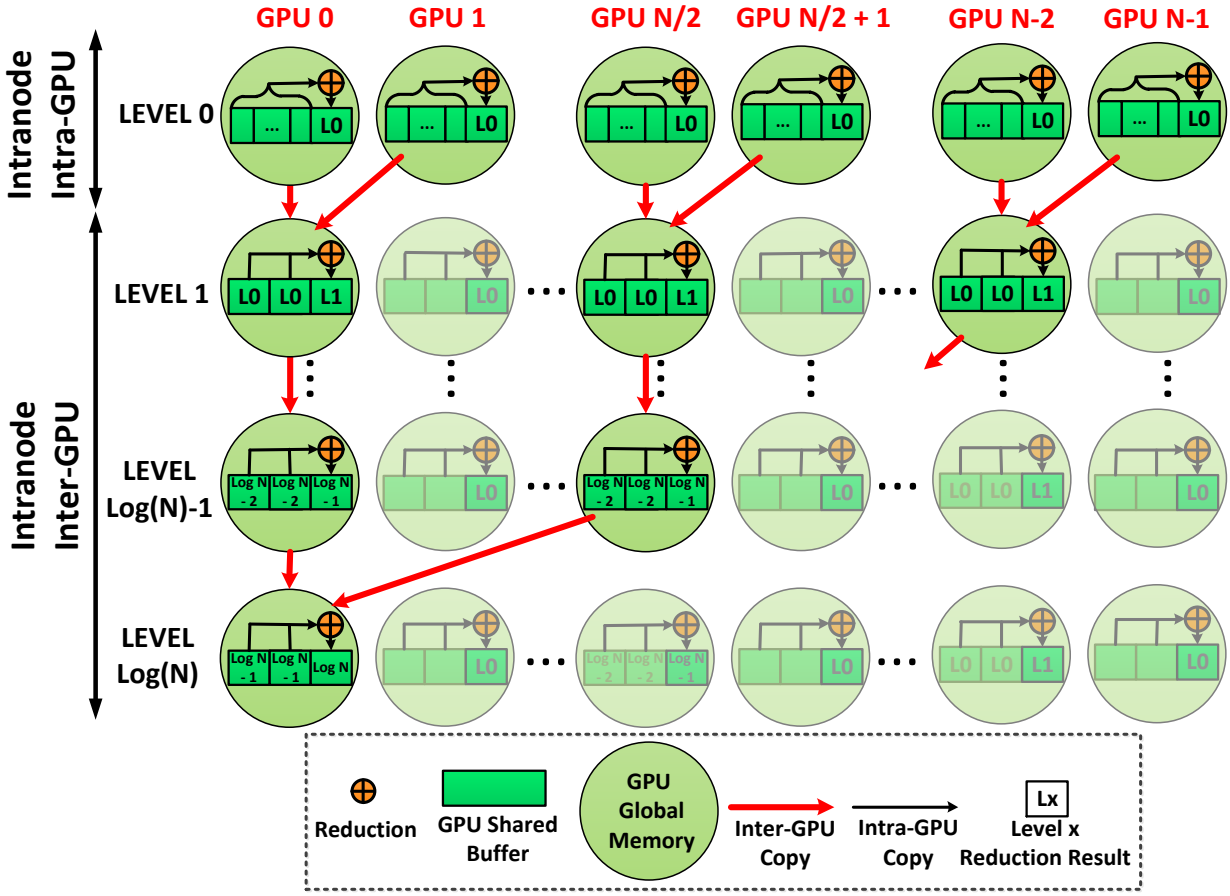


Figure 4.3: Hierarchical MPI_Allreduce utilizing Intranode Intra-GPU GSB and Intranode Inter-GPU BTB Reduce algorithms - Reduce stage

inter-GPU communication hierarchy.

MPI_Allreduce

To apply our proposed hierarchical communication framework to MPI_Allreduce, we propose a hierarchical design for MPI_Allreduce collective operation in clusters of multi-GPU nodes that consists of three hierarchical stages: Stage1) Intranode Reduce; Stage 2) Internode Allreduce; and Stage3) Intranode Broadcast. Fig. 4.5 illustrates these general stages. In the following, we discuss these stages and explain how our proposed framework is applied to them.

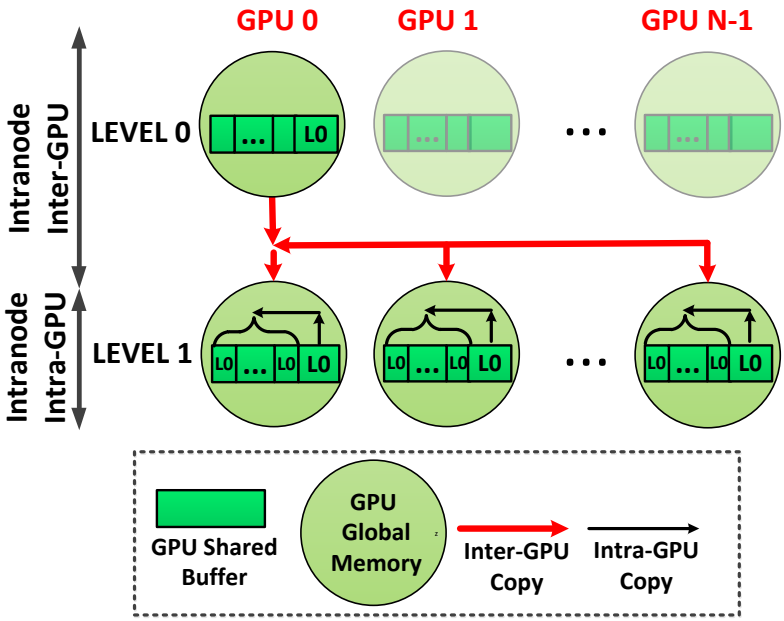


Figure 4.4: Hierarchical MPI_Allreduce utilizing Intranode Intra-GPU GSB Broadcast and Intranode Inter-GPU GSB Broadcast algorithms - Broadcast stage

Stage1: Intranode Reduce

Step1: Intranode Intra-GPU Reduce: Intranode intra-GPU processes reduce their pertinent data and store it in the GPU shared buffer of their pre-defined leader process; this step is performed using the GSB Reduce (or BTB Reduce) algorithm.

Step2: Intranode Inter-GPU Reduce: On each node, using the GSB Reduce (or BTB Reduce) algorithm the reduce operation is performed among the intra-GPU leader processes and the result is stored in the pre-defined node leader process.

Note that by adding an internode inter-GPU Reduce stage after the above two steps in Stage 1, MPI_Reduce can be can be implemented across the cluster.

Stage 2: Internode AllReduce:

The node leader processes engage in an Internode Inter-GPU Allreduce stage. We use the existing internode MVAPICH2 algorithm to perform MPI_Allreduce in this

step. At this point, the final reduction result is available at the node leader processes.

Stage 3:Intranode Broadcast

Step1: Intranode Inter-GPU Broadcast: The node leader process in each node use the GSB Broadcast (or BTB Broadcast) to broadcast the data to the GPU leader processes.

Step2: Intranode Intra-GPU Broadcast: Each GPU leader process use the GSB Broadcast (or BTB Broadcast) to broadcast the reduced data to its intranode intra-GPU processes.

Note that by adding an internode inter-GPU Broadcast stage before the above two steps, MPI_Bcast can be implemented across the multi-GPU cluster.

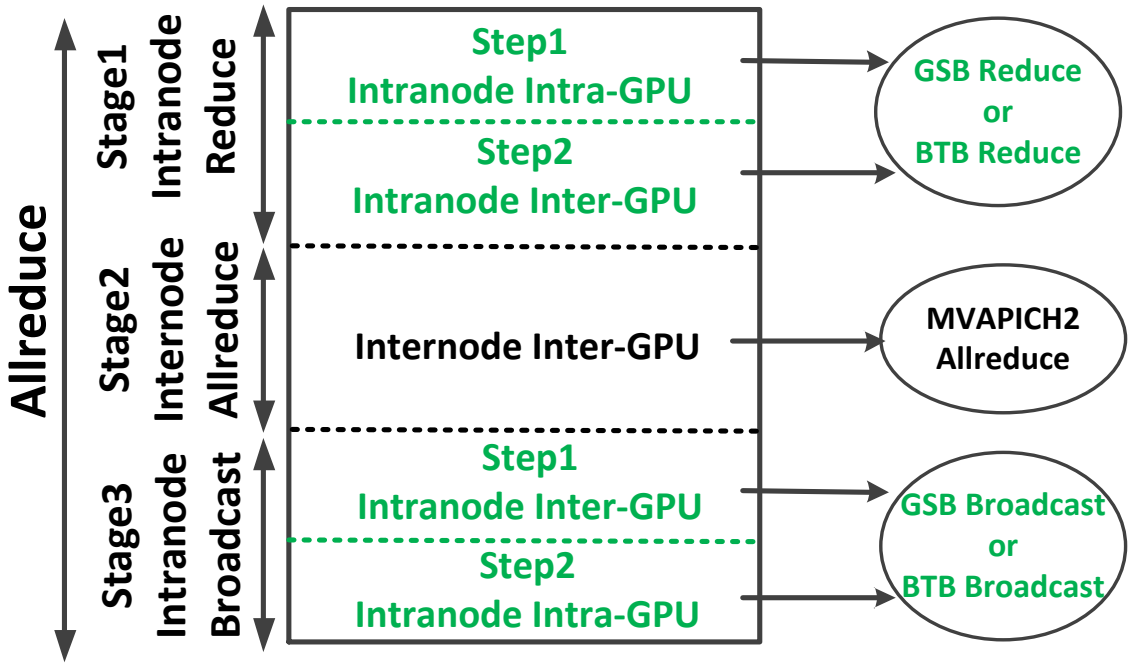


Figure 4.5: Steps of the hierarchical GPU-Aware MPI_Allreduce on GPU clusters with multi-GPU nodes

4.3 Experimental Results and Analysis

In this section, we evaluate our proposed hierarchical framework on a multi-GPU node and across the cluster. In our tests, we consider the GSB and BTB designs in different intranode hierarchy levels. In all of these figures, we compare two different design scenarios and provide the relative performance improvement.

4.3.1 Experimental Platform

We conduct our experiments on a cluster with multi-GPU nodes. In this regard, we use System B (Helios cluster) with 4 nodes and 8 K80 GPUs in each node, as described in Chapter 3. The System B nodes use QDR InfiniBand as their interconnect. We compare our proposed designs against the existing design in MVAPICH2-2.1 and present the results for various message sizes (256KB to 16MB)[1]. We consider up to 24 processes per node (a total of 96 processes) in our tests that are uniformly distributed among different GPUs. Our results are reported for 256 KB to 16 MB message sizes, as for smaller message sizes our designs provide limited or no improvement. In the rest of this section, we discuss our results on a single multi-GPU node and then across the cluster.

4.3.2 Results on a Single Multi-GPU Node

Fig. 4.6 and Fig. 4.7 evaluate our designs on a single 8-GPU Helios node (System B). Fig. 4.6 compares the GSB-GSB design with the MVAPICH2 MPI_Allreduce. According to the figure, the GSB-GSB design is superior over MVAPICH2 in all test cases. We can also observe that the GSB-GSB design provides higher improvement

[1]Note that MVAPICH2-GDR was not available on System B.

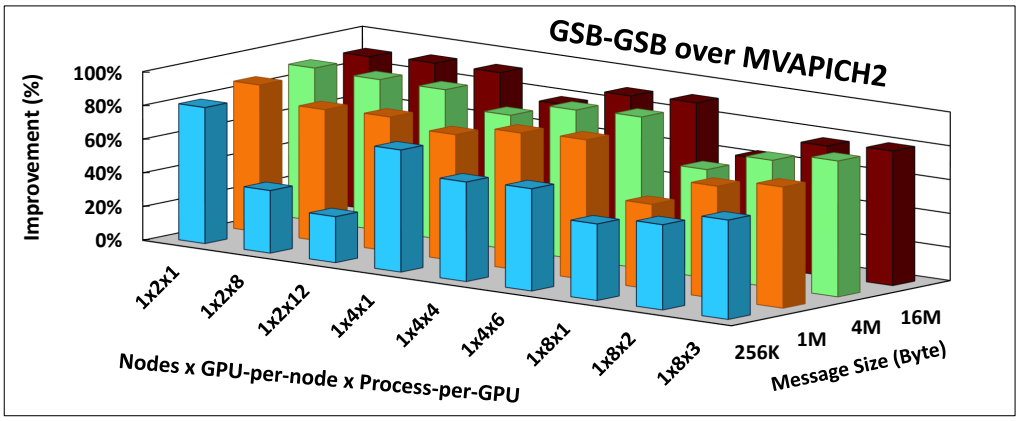
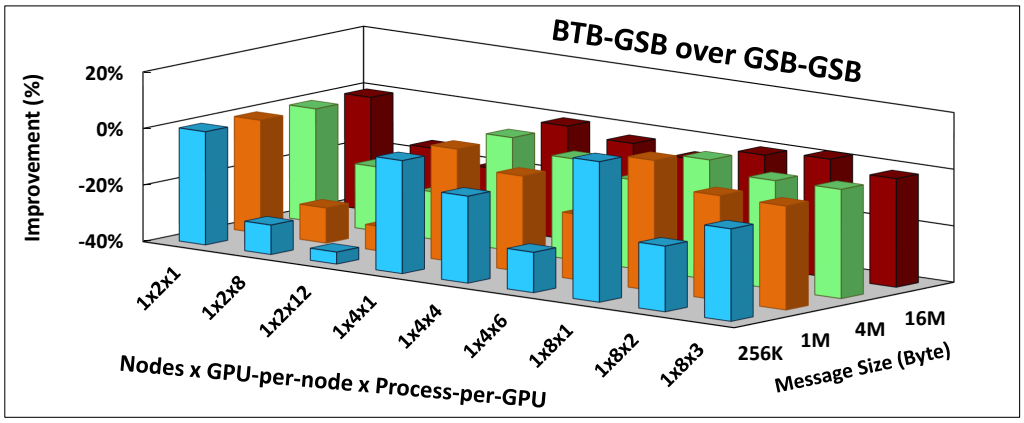


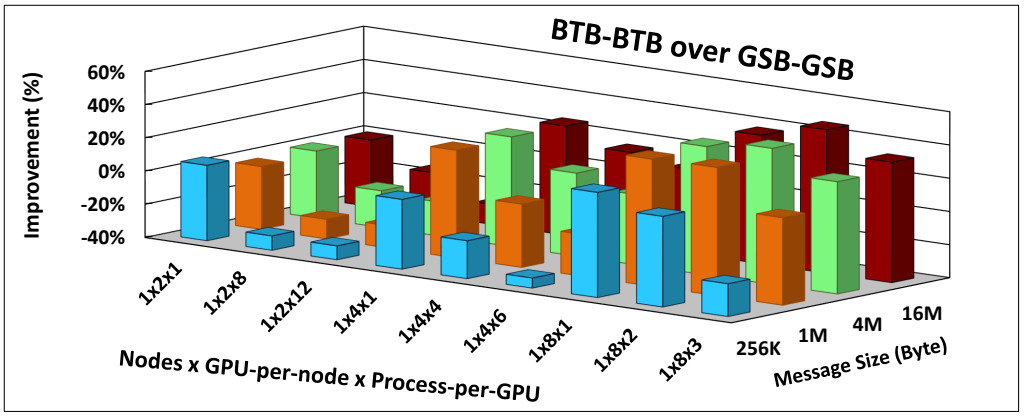
Figure 4.6: GPU hierarchical MPI_Allreduce with GSB for Intranode Intra-GPU and GSB for Intranode Inter-GPU steps over MVAPCIH2 on a single Helios K80 node with multiple GPUs

for cases with larger message size and higher number of processes per GPU.

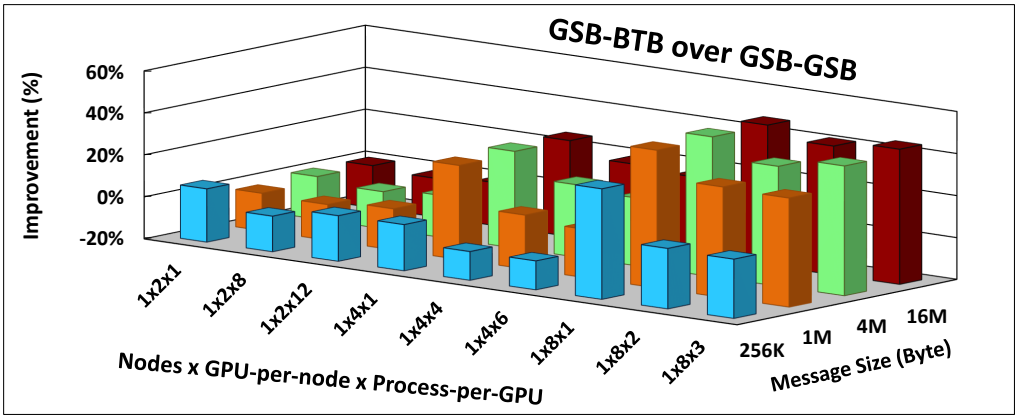
To investigate the choice of algorithms in our framework, we compare the GSB-GSB case (Case1) against the BTB-GSB (Case2), BTB-BTB (Case3), and GSB-BTB (Case4) in Fig. 4.7(a), 4.7(b), and 4.7(c), respectively. According to Fig. 4.7(a), the BTB-GSB design underperforms the GSB-GSB design in all test cases. These results comply with the results in Fig. 3.4, showing that the BTB design is not preferable in the intranode intra-GPU step. Using the BTB-BTB design (Fig. 4.7(b)) can lead to some performance improvement in cases of 4 and 8 GPUs. Interestingly, with the BTB-BTB design, performance improves as the number of GPUs per node increases, and degrades as the number of processes per GPUs increases. This implies that despite the inefficiency of using the BTB design in the intranode intra-GPU step, it can lead to performance improvement when used in the intranode inter-GPU step. Finally, the GSB-BTB design outperforms the GSB-GSB design in all test cases (Fig. 4.7(c)). This verifies that the GSB and the BTB approach should be the algorithm of choice for the intranode intra-GPU and intranode inter-GPU step, respectively.



(a) Case2: Hierarchical design - Intra-GPU: BTB, Inter-GPU: GSB over Case1: Hierarchical design - Intra-GPU: GSB, Inter-GPU: GSB



(b) Case3: Hierarchical design - Intra-GPU: BTB, Inter-GPU: BTB over Case1: Hierarchical design - Intra-GPU: GSB, Inter-GPU: GSB



(c) Case4: Hierarchical design - Intra-GPU: GSB, Inter-GPU: BTB over Case1: Hierarchical design - Intra-GPU: GSB, Inter-GPU: GSB

Figure 4.7: Evaluating the effect of using different algorithms in the GPU hierarchical MPI_Allreduce on a a single Helios K80 node with multiple GPUs per node

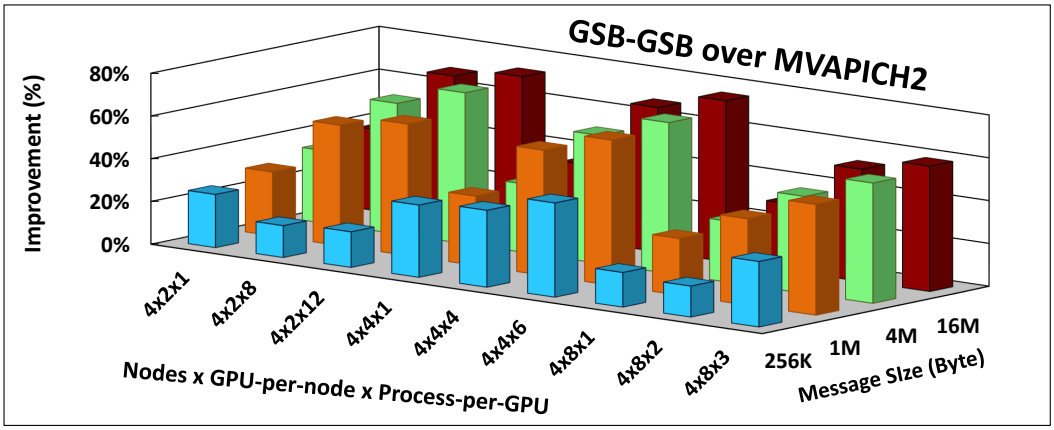
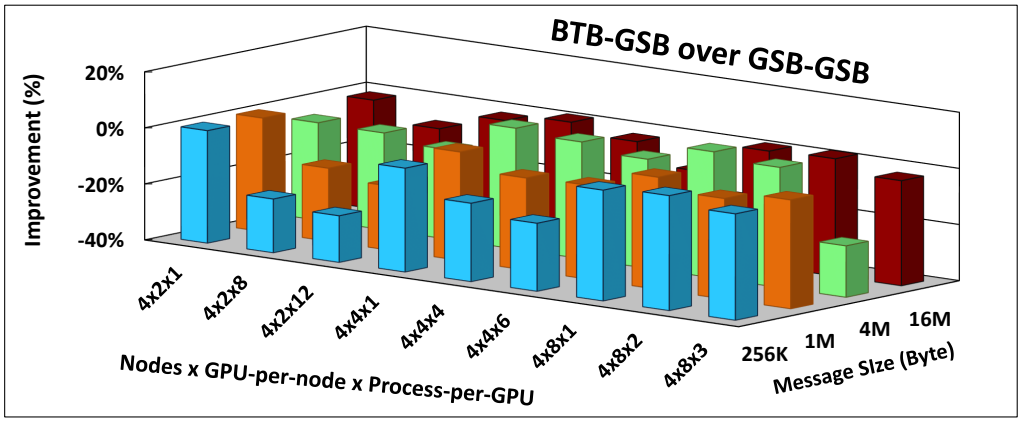


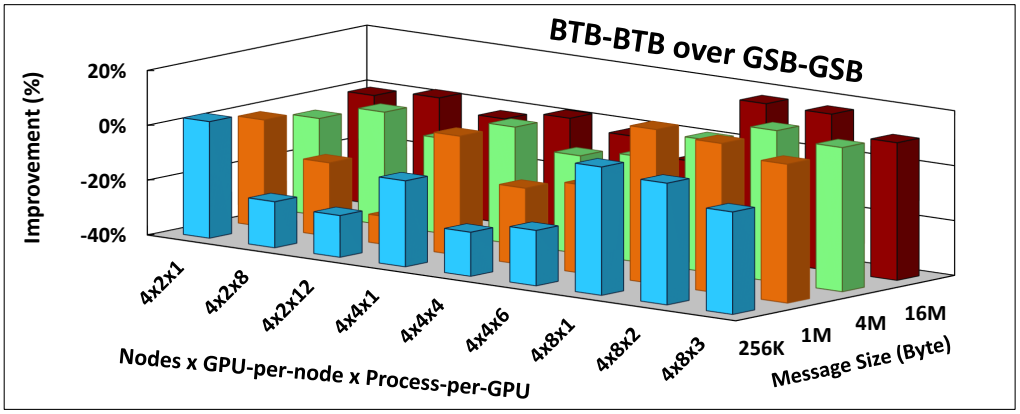
Figure 4.8: GPU Hierarchical MPI_Allreduce with GSB for Intranode Intra-GPU and GSB for Intranode Inter-GPU steps over MVAPCIH2 MPI_Allreduce on four Helios K80 nodes with multiple GPUs per node

4.3.3 Results on a Cluster of Multi-GPU Nodes

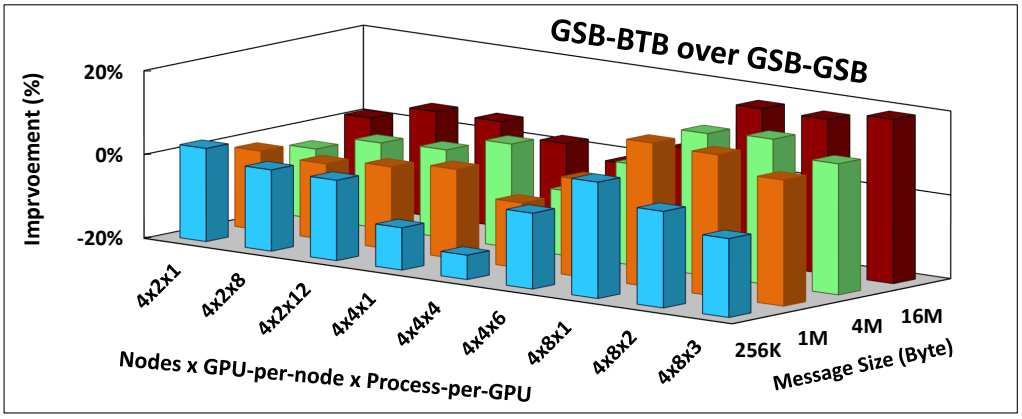
Fig. 4.8 and Fig. 4.9 provide our cluster-wide results on a cluster of four 8-GPU Helios nodes (System B). Similar to our single-node analysis, our cluster-wide experiments indicate that the GSB-GSB design can outperform the MVAPICH2 MPI_Allreduce in all test cases (Fig. 4.8), although with a lower improvement compared to the single-node results (Fig. 4.6). This is an expected behavior as we are using a fixed internode algorithm in the internode step of our framework. Thus, the higher the share of the intranode step (the more processes per node) in the collective operation, the higher the improvement potential in our proposals. In Fig. 4.9, we also evaluate the choice of different algorithms for the intranode step of a cluster-wide MPI_Allreduce operation. According to this figure and similar to our intranode results in Fig. 4.7, the GSB-BTB approach is showing to be the algorithm of choice for the intranode step of the cluster-wide MPI_Allreduce operation as well.



(a) Case2: Hierarchical design - Intra-GPU: BTB, Inter-GPU: GSB over Case1: Hierarchical design - Intra-GPU: GSB, Inter-GPU: GSB



(b) Case3: Hierarchical design - Intra-GPU: BTB, Inter-GPU: BTB over Case1: Hierarchical design - Intra-GPU: GSB, Inter-GPU: GSB



(c) Case4: Hierarchical design - Intra-GPU: GSB, Inter-GPU: BTB over Case1: Hierarchical design - Intra-GPU: GSB, Inter-GPU: GSB

Figure 4.9: Evaluating the effect of using different algorithms in the GPU hierarchical MPI_Allreduce on four Helios K80 nodes with multiple GPUs per node

4.4 Summary

Multi-GPU nodes have become a prevailing choice for HPC clusters to reach higher compute power. In such clusters, GPU inter-process communication can take place at different hierarchy levels, such as within a single GPU, across intranode GPUs, or among GPUs on different nodes. Taking this structure into account, in this chapter we proposed a hierarchical framework for collective operations for both single Multi-GPU nodes as well as cluster of multi-GPU nodes.

On a multi-GPU node, we applied our proposed hierarchical framework to MPI_Allreduce, while our framework can also be applied to other collective operations. We performed MPI_Allreduce in two stages: Stage1) Reduce; and Stage2) Broadcast. Our hierarchical framework is then applied to both stages by breaking them into intranode intra-GPU and intranode inter-GPU steps. By studying various algorithms within these steps, we showed the importance of choosing the right algorithm for different hierarchy levels of the GPU collective operations. We also evaluated our framework on MPI_Allreduce across the clusters and showed promising performance results. Our experimental results showed to highly benefit MPI_Allreduce with large message sizes which are highly used in deep learning and big data applications.

Our hierarchical collective designs in this chapter complements our GPU-aware collective communication algorithms in Chapter 3. In Chapter 3, we leveraged different algorithms and hardware features for GPU collectives targeting single GPU nodes. In this chapter, on the other hand, we utilized the hierarchical structure of GPU clusters with multi-GPU nodes to propose hierarchy-aware GPU collectives. Our proposals in these chapters show to mainly benefit large message sizes. We investigate this behavior in Chapter 5, and propose various designs to intelligently select

the right data copy mechanism in collective operations targeting single-GPU nodes.

Chapter 5

Efficient GPU Communications through Smart Data Copy Mechanism Selection

GPU accelerators have tremendously evolved over the past decade. During this period, the main and probably the most noticeable improvement in these accelerators is the increase in their floating-point operations per second (FLOPS) and power efficiency. Apart from this, GPUs have also evolved by introducing advanced features, such as GPUDirect, Dynamic Parallelism, UVA, UM, and Hyper-Q (discussed in Chapter 2) which can help to further harness the potential of the GPU resources. While such advances provide a vehicle for increasing the performance, GPU applications can find themselves constrained by potentially costly inter-process GPU communication performance. Thus, efficiently leveraging the GPU resources as well as the latest GPU features are of paramount importance in improving the performance of GPU applications.

Modern GPUs have the ability to leverage different data copy mechanisms and communication channels for inter-process communications. For such communications,

a single data copy mechanism is selected that well fits the communication characteristics, such as the message size and the communication channel between GPU processes. Taking this into consideration, researchers have utilized an efficient single GPU data copy mechanism to propose GPU-aware point-to-point and collective designs [73], and as proposed in Chapter 3 and 4of this dissertation. The fruit of these work is showing the high impact of using the right single data copy mechanism in point-to-point and collective operations.

While it is most efficient to use a specific data copy mechanism for a single GPU inter-process communication, in this chapter we show the benefit of using multiple data copy mechanisms to perform multiple inter-process communications. To this aim, we propose collective designs that use different data copy mechanisms in conjunction with each other to perform GPU collective operations. The rationale behind our proposals in this chapter is to overlap the use of different data copy mechanisms for different inter-process communications, thus improving the total collective communication runtime. Therefore, our goal is to propose designs that can assess the performance and availability of multiple data copy mechanisms and cooperatively exploit them to perform GPU inter-process communications of the collective operations. We first present our designs for a single-GPU node and next provide the scalability of our designs to across the cluster.

In this chapter we make the following key contributions:

- We first provide evidence of using multiple data copy mechanisms in intranode inter-process GPU communications. We also demonstrate that the benefit of using multiple data copy mechanisms can be accentuated with the GPU feature, called Nvidia MPS service. For intranode GPU communications, we utilize two

inter-process data copy mechanisms: 1) CUDA IPC; and 2) host-staged. We observe that, not only does the MPS service can improve the performance of each of these data copy mechanisms, it can also further overlap them as well. We also show that for multiple inter-process communications, the most efficient combination of data copy mechanisms can vary based on the message size and process count.

- We propose a *Static* algorithm and an alternative *Dynamic* algorithm for intra-node MPI_Allgather and MPI_Allreduce operations. Both designs are tuned to leverage the MPS service and use a combination of different data copy mechanisms to perform their collective operations. The *Static* algorithm decides the number and the mechanism of the copy for inter-process communications based on a tuning table that is provided prior to the runtime. The *Dynamic* algorithm, on the other hand, chooses the right number and mechanism of the copy based on the information that it gathers at runtime [24, 22].

- We provide a node-wide and cluster-wide analysis of our proposed designs. We compare our proposals against the existing GPU-aware collectives, including our proposed GSB and BTB algorithms in Chapter 3. We show that the *Static* and the *Dynamic* approaches, by intelligently selecting the right number and mechanism of the copies, can outperform the existing collective designs, specifically for cases that an inefficient data copy mechanism is in-use.

- We shed some light on the effect of the MPS service on our proposed designs by profiling and performing further experiments. We conclude that efficient design decisions are indeed required to utilize the MPS service, as otherwise this

service can only provide limited improvement or even impose overhead. With our collective designs being aware of the Hyper-Q feature, further improvement compared to the other alternative designs can be realized.

5.1 Motivation

In this section, we provide some motivational results that serve as the backbone of our designs. With these results, we show the benefit of using the Nvidia Multi-Process Service (MPS) [64] on the inter-process GPU communications using different data copy mechanisms. Our goal in this section is twofold. First, we show that there are certain message ranges in which a single specific data copy mechanism is most favored for inter-process communications. With the MPS service this trend will not change, even though the communication performance can potentially improve. Second, we provide evidence that for some message sizes utilizing multiple data copy mechanisms in conjunction with the MPS service is the best way to improve multiple inter-process communications.

5.1.1 Impact of MPS and Hyper-Q on Communication

The Nvidia Hyper-Q feature, as discussed in Chapter 2, provides potential concurrency among different tasks on a single process. The MPS service [64], on the other hand, allows the Hyper-Q feature to take effect among multiple processes and allow them to potentially run their tasks on single GPU resources. The benefit of using the Hyper-Q feature through the MPS service is already evaluated on various applications and offloaded computational kernels [102, 13, 71, 9]. In this section, on the other hand, we specifically evaluate the impact of the Hyper-Q feature and the MPS

service on the point-to-point intranode communications. We consider four point-to-point communicating pairs that are first synchronized and then perform pair-wise communications with either the host-staged (HS) or CUDA IPC data copy method. We consider three scenarios: 1) all communications are performed with only host-staged copy; 2) half of the communications are performed using host-staged and the other half are performed using CUDA IPC; and 3) all communications are performed using CUDA IPC.

Fig. 5.1 presents our microbenchmark results with and without the MPS service. According to the figure, three main observations can be made. The most apparent observation is that both host-staged and IPC data copy mechanisms can benefit from the MPS service mainly for small and medium message sizes. For 128KB messages and above, however, the MPS service imposes overhead on both data copy mechanisms, with a larger impact on the host-staged data copy mechanism. This is due to the fact that MPS allows multiple memory copies to be issued faster and reduce their initialization overhead. However, for large message sizes this overhead is negligible compared to the data copy time. MPS service imposes more overhead on the host-staged copies on large message sizes which we believe is mainly implementation dependent. We also observed that the MPS service provides the possibility of multiple CUDA IPC copies to overlap with each other while this overlap possibility already exists among multiple host-staged copies without the MPS service. Secondly, it can be seen that the host-staged copies (with or without MPS) are faster than the CUDA IPC copies for small and medium message sizes. The opposite trend can be seen for large message sizes, in which the CUDA IPC copies (with or without MPS) are superior to the host-staged copies. Finally, we can also observe that in some message

sizes (32KB and 64KB) with MPS enabled, the communication using both data copy mechanisms is superior to the communication with a single data copy mechanism. This implies that the Hyper-Q feature through the MPS service is providing some overlap between the host-staged and CUDA IPC communications.

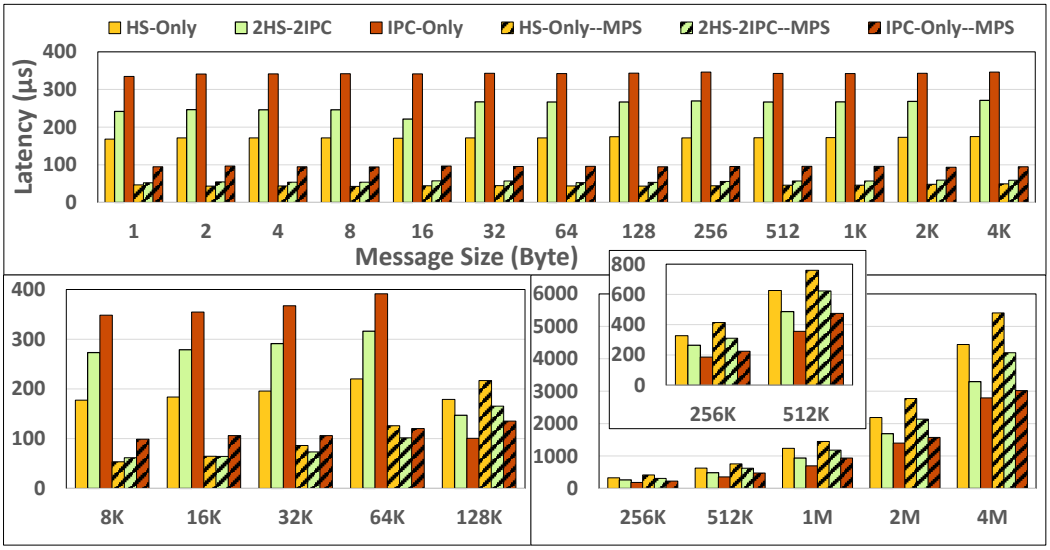


Figure 5.1: Hyper-Q effect on intranode point-to-point communication with and without MPS

While Fig. 5.1 shows the potential benefit of the Hyper-Q feature in intranode communications with different data copy mechanisms, it only considers three out of five possible combinations. For further investigations, we repeated the tests with the MPS service enabled, but this time considering all possible combinations. As shown in Fig. 5.2, the Hyper-Q feature through the MPS service provides faster communication when using both data copy mechanisms for a larger message range (8KB to 256KB). It can be argued that there is no silver bullet combination working efficiently across all message sizes; thus, the best approach is to leverage different combinations of data copy mechanisms across different message sizes.

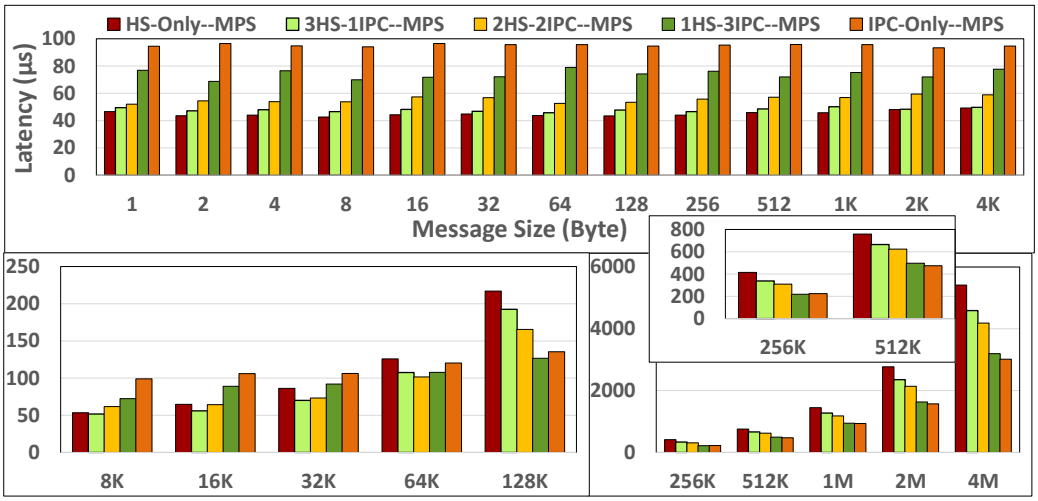


Figure 5.2: Hyper-Q effect on intranode point-to-point communication with MPS enabled

In Chapter 3 and Chapter 4, we addressed the following question: what should be the algorithm of choice in GPU collective operations. In this chapter, on the other hand, considering the findings in the motivational study, we raise the following question: how different GPU data copy mechanisms can be used in conjunction with each other to improve the GPU collective communication?

5.2 Related Work

Since the introduction of the Nvidia Fermi GPUs, researchers have studied various ways to avoid GPU underutilization. Guevara et al. [30] introduced concurrent kernel execution for Nvidia GT200 GPUs. The authors proposed an approach to allow multiple kernels to share the same GPU resources by merging them into a large kernel. Wang et al. evaluated different approaches for concurrent kernel execution [98, 100, 99]. The authors evaluated sharing the CUDA context feature, which became available with CUDA 4, among multiple processes. Their experimental results

showed up to 90% performance improvement by using context sharing. Wende et al. [101] proposed an approach to reduce the false-serialization effect of running multiple independent GPU kernels on Fermi GPUs. The false serialization effect in Fermi is due to the use of a single task queue by the CUDA scheduler. The authors proposed a GPU kernel re-ordering mechanism to mitigate this effect. This inefficiency, however, was addressed by Nvidia in the next generation GPUs followed by Fermi (i.e., Nvidia Kepler). The Kepler GPUs by using 32 task queues reduced the false serialization effect significantly. NVIIDA has also introduced the MPS service to allow multiple processes share the same GPU resources. The benefit of the MPS service has been evaluated on various applications and computational kernels [102, 13, 71, 9]. Wende et al. [102] performed a detailed analysis of the MPS service on multiple offloaded computational kernels. This work showed that the MPS service can efficiently allow multiple compute kernels to share resources of a single GPU and improve its utilization. On the other hand, to the best of our knowledge, this chapter for the first time evaluates the effect of the MPS service and the Hyper-Q feature on the intranode GPU inter-process data copy mechanisms. Accordingly, we propose two different algorithms that can efficiently use these features to improve the GPU inter-process communication performance.

In GPU clusters, various GPU-aware solutions for MPI operations have been proposed. All of these designs unanimously utilize a single data copy mechanism to perform their GPU inter-process communication [85, 84, 39, 73, 72]. Singh et al. [85] optimized internode MPI_Alltoall utilizing host-staged copy in their designs. To improve the performance of this costly operation, they overlapped the device-to-host and host-to-device CUDA memory copies with the network communications. Singh also

used host-staged copy to optimize internode MPI_Allgather using a store-and-forward approach [84]. The CUDA IPC data copy mechanism was studied in [39, 73] and used for one-sided and point-to-point communications. In this chapter, we leverage a combination of both data copy mechanisms for intranode inter-process communications. We propose algorithms for GPU collective operations that are capable of deciding the number and mechanism of the inter-process copies. Depending on the algorithm, we opt to make this decision based on the information that is gathered either during or prior to the runtime.

5.3 GPU Collective Designs with Efficient Data Copy Mechanism Selection

In this section, we exploit the Nvidia MPS and the Hyper-Q feature to propose a *Static* algorithm and an alternative *Dynamic* algorithm for intranode GPU collectives. Both of the *Static* and *Dynamic* approaches decide the number and mechanism (host-staged or IPC) of the GPU inter-process communications that are involved in the collective operation. The *Static* algorithm makes this decision based on a priori information that it extracts from a tuning table. The *Dynamic* algorithm, on the other hand, dynamically decides the number and mechanism of the copies at runtime. Our proposed algorithms can be applied to any collective operation; however, in this chapter we will consider MPI_Allgather and MPI_Allreduce operations as our case studies. While both the *Static* and *Dynamic* designs go through the same general steps (i.e., Gather, Kernel Function, and Broadcast), each step has a different algorithm.

5.3.1 Static Hyper-Q Aware Algorithm

Tuning collective operations can be performed by conducting experiments on the underlying system and exploiting the gathered information; this has been extensively studied and shown to highly improve the collective performance [95]. In our work, the tuning table for each collective associates the most efficient combination of the data copy mechanisms to each message size and process count. For example, the configuration of using 10 host-staged and 5 CUDA IPC copies has shown to be the most efficient combination to perform MPI_Allgather on 16 processes (that are bound to the 16 CPU cores) and 16KB of data[1]. The main steps of the *Static* algorithm are described below.

Stage1: Intranode Intra-GPU Gather

All processes copy their share of data into the GPU shared buffer area in a first-come first-serve order, using a data copy mechanism that is assigned to them by the leader process (without loss of generality, process with `rank 0` is considered as the leader process). The leader retrieves the most efficient combination of the data copy mechanisms from the tuning table, assigns a particular data copy mechanism to each process, and then queries their completion.

Stage 2: Kernel Function

In this step, a kernel function is called by the leader process on the aggregated data in its GPU shared buffer. This step is only required for some collective operations. In our test cases, only MPI_Allreduce goes through this step and performs an element-wise reduction on the aggregated data in the GPU shared buffer.

Stage 3: Intranode Intra-GPU Broadcast

[1]note that the leader process in our algorithms does not use any of the host-staged or IPC data copy mechanisms

The collective result is now available in the GPU shared buffer and is copied out into the destination buffer of the participating processes. Similar to the `Gather` step, the leader process assigns a particular data copy mechanism for each of the inter-process copies on the information that is extracted from the tuning table.

5.3.2 Dynamic Hyper-Q Aware Algorithm

The *Static* algorithm is dependent on tuning parameters that must be available prior to the runtime. On top of that, the tuning parameters for a particular platform may not necessarily be useful on another platform. We propose a *Dynamic* algorithm that is independent of any tuning parameters and is capable of determining the data copy mechanism for each process by solely exploiting the runtime information. The idea behind this approach is to decide the data copy mechanisms based on their availability and efficiency. We acquire this information by querying the responsiveness of these data copy mechanisms. In other words, the *Dynamic* algorithm tends to choose the slower and less available data copy mechanism less frequently, while the faster and more responsive data copy mechanism is more frequently selected. The main steps of this algorithm are discussed below:

Stage 1: Intranode Intra-GPU Gather

All of the participating processes copy their share of data into the GPU shared buffer of the leader process. In this step, the leader process uses an algorithm that decides the mechanism of its inter-process copy based on the responsiveness of the data copy mechanisms; this algorithm goes through the following phases:

Phase1. Initialization: Considering that at this point no prior knowledge about

the data copy mechanisms exists, assessing the responsiveness of a data copy mechanism can only be done by actually assigning a host-staged and a CUDA IPC data copy mechanism to the first two (non-leader) processes arriving at the collective call and then querying their completion.

Phase2. Progress: The leader process queries the pending copies and waits until one completes. Our intuition is that it is more efficient to issue multiple copies on a faster data copy mechanism all at once. In this regard, once an inter-process communication using a specific data copy mechanism completes, we calculate the difference between the number of completed communications using the host-staged and the CUDA IPC data copy mechanisms. A zero or a negative difference indicates that the currently completed data copy mechanism is not as fast as the other data copy mechanism, thus only a single copy is issued with this slow but yet available data copy mechanism. A positive difference, on the other hand, indicates that the completed data copy mechanism is faster than the other mechanism and thus multiple copies should be assigned to the next available processes. We determine the number of the issued copies to be two to the power of this difference; this way, we ensure quick assignment of the inter-process communications to the faster data copy mechanism and thus using it more frequently. This procedure continues until the last copy is issued.

Phase3. Final Copy Completion: For the final pending copy, the leader takes a different approach. The rationale behind this is that the last pending copy could potentially linger for a long time and also there is no pending copy using the other mechanism. At this point, the leader checks if there has been any successful completion of this data copy mechanism before. If this is not the case, the leader considers

this data copy mechanism to be extremely slow and assigns the final copy to be re-sent with the other data copy mechanism. If the slow copy turns out to be of the host-staged mechanism, the remaining portion of the host-staged copy (if any) will be discarded. This can be supported by packetizing the host-staged copies into large chunks and sending them back to back. Once a process receives a re-send assignment with the IPC data copy mechanism, the remaining packets of the host-staged copy will be discarded. However, this approach cannot be applied to the slow IPC data copy mechanism; therefore, we overlap the slow IPC copy with the next steps of the collective operation.

Stage2: Kernel Function

This step is similar to the *Static* algorithm.

Stage 3: Intranode Intra-GPU Broadcast

Both data copy mechanisms can be potentially used to broadcast the available result in the GPU shared buffer among the participating processes. However, unlike `Step 1`, the leader now has some knowledge about the efficiency of the data copy mechanisms. Based on this information, this step goes through the following phases:

Phase1. Initialization: If there has been no successful completion of a data copy mechanism since the beginning of the collective operation, the leader tags it as a slow data copy mechanism and avoids using it in the broadcast step. Otherwise, both data copy mechanisms can potentially be used in this step.

Phase2. Progress: If all copies are initiated using a single mechanism, the completion of that mechanism is only required to be queried. Otherwise, the leader monitors the progress of both data copy mechanisms and uses the faster data copy mechanism more frequently and waits for all of the pending copies in the current or previous

step(s) to complete before returning form the collective operation.

Implementation Details

Both *Static* and *Dynamic* algorithms use pre-allocated CPU and GPU shared buffers. These buffers are allocated during `MPI_Init()`, with the CPU shared buffer also being registered to prevent it from being swapped out. During `MPI_Init()`, we also check the status of the MPS service in order to choose the right tuning parameters. After the leader allocates its GPU shared buffer, it broadcasts its memory handle to the other processes. Buffer allocation and broadcasting the handle are expensive operations and thus are only performed once to mitigate their high cost.

Fig. 5.3 illustrates different components of the Dynamic algorithm and shows how processes can communicate with each other through the CPU and GPU shared buffers. The GPU shared buffer is used to gather the pertinent data from all participating processes. The CPU shared buffer is used for staging the data in the host-staged mechanism of copy; it also serves as a directory to track the communications between the processes. The directory is composed of two parts, the `Completion Flag` and the `Copy Status Flags`; setting the `Completion Flag` indicates that the result of the collective operation is available in the GPU shared buffer and can be copied out. The `Copy Status Flags` show the status of the copy operations. In the *Static* algorithm, these flags indicate the initiation and completion of the copies; in the *Dynamic* algorithm these flags are also used by the leader process to assign the data copy mechanism to the other processes. Given the use of directory in the *Static* algorithm is a simplified version of the *Dynamic* algorithm, in the following we will only discuss the implementation details of the *Dynamic* algorithm.

The `Copy Status Flags` in the directory of the *Dynamic* algorithm for MPI_Allreduce can take one of the following eight states: `INIT`, `RTS`, `IPC_ASGN`, `HS_ASGN`, `IPC_INIT`, `HS_INIT`, `IPC_CMP`, and `HS_CMP`. The `INIT` state represents the initial state, and the `Copy Status Flags` are set to this state before entering and exiting the collective operation. The `RTS` (Ready To Send) flag indicates a process arrival to the collective operation. The `IPC_ASGN` and the `HS_ASGN` are the assignment flags, set by the leader process to assign IPC and host-staged data copy mechanisms for inter-process communication, respectively. The `IPC_INIT` and `HS_INIT` indicate the initiation of the IPC and host-staged copy, respectively. The `IPC_CMP` and the `HS_CMP` indicate completion of the IPC and host-staged copy, respectively.

Fig. 5.3 illustrates the different steps of the *Dynamic* algorithm for MPI_Allreduce and shows how processes can communicate with each other through the directory. Fig 5.3(a) shows the initial state, in which the `Copy Status Flags` are all set to the `INIT` state and the `Completion Flag` is set to zero. Fig. 5.3(b) shows a snapshot of the Gather step of the *Dynamic* algorithm. In this step, the leader process first queries the `Copy Status Flags`, looking for an `RTS` state. As can be seen in the figure, P_1 (process with `rank 1`) has arrived at the collective operation and is ready to send its data, while P_{n-3} has not yet arrived at the operation. All processes (except the leader) initiate their copy once their data copy mechanism is determined. In Fig. 5.3(b), the leader has assigned the host-staged data copy mechanism to P_{n-1}. The status of the P_{n-4} indicates that it has initiated its IPC copy. Depending on the initiated data copy mechanism, different course of actions is required to guarantee its completion. The completion of the CUDA IPC copy requires synchronization between the sender and the receiver sides. In this regard, a CUDA inter-process event is recorded right

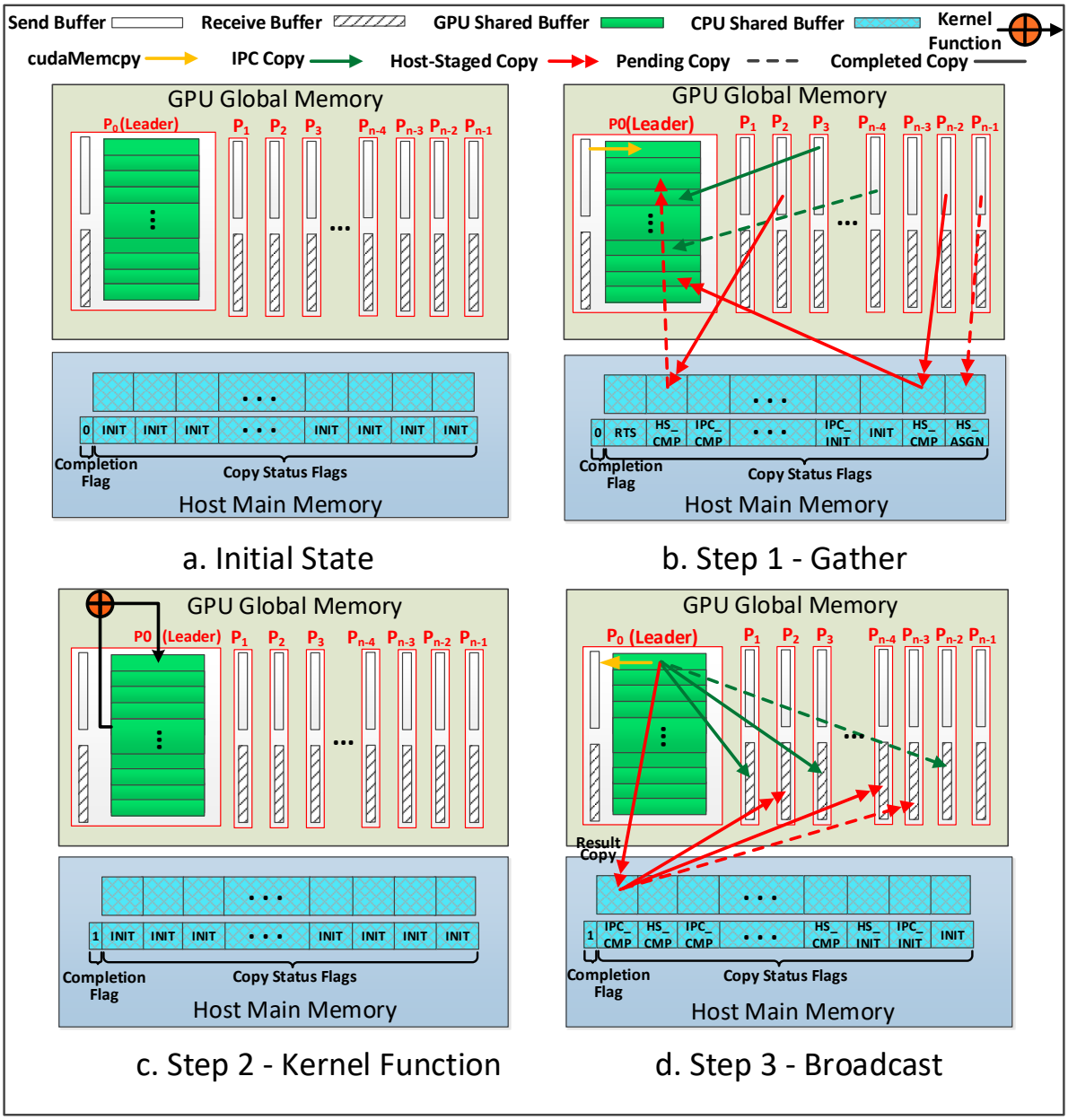


Figure 5.3: Different steps of the node-wide *Dynamic* algorithm for MPI_Allreduce

after the sender process starts its CUDA IPC copy and before it sets its associated flag in the directory to `IPC_INIT`. The leader (the receiving process), once observes this flag, queries the inter-process event to check the IPC copy completion; once the

copy completes, the leader sets the associated entry in the `Copy Status Flags` to the `IPC_CMP` state. In the case of the host-staged copy, the process initiating the copy is also responsible for querying its completion and setting its associated flag to the `HS_CMP`. Note that IPC and host-staged copies are initiated asynchronously, thus allowing the sending process to query the potential re-send assignment from the leader. According to the figure, P_2 and P_{n-2} have completed their host-staged copies and their share of data are available in the GPU shared buffer. The leader is also copying the staged data from P_2 and has already completed the copy from P_{n-2}. The state of P_3 shows that its IPC copy is completed. The dotted copies in the figure resemble pending copies and have the potential to be overlapped with each other using the Hyper-Q feature.

Fig. 5.3(c) shows the Kernel Function step. This step can be skipped in some collectives (MPI_Allgather in our case) by immediately setting the `Completion Flag` and resetting all of the `Copy Status Flags` back to the `INIT` state once the Gather step completes. In some collectives (MPI_Allreduce in our case), a kernel function is called on the aggregated data and the result is stored in the GPU shared buffer. Once the kernel function completes, the leader sets the `Completion Flag` to inform other processes that the result is available.

Fig. 5.3(d) shows a snapshot of the Broadcast step of the *Dynamic* algorithm. In this step, the collective result is broadcast to all participating processes. The steps shown in the figure applies to MPI_Allreduce in which the reduced result is broadcast from the shared buffer to all processes. For other collective operations this step may require some modifications; for example, in MPI_Allgather, the entire gathered data in the GPU shared buffer is broadcast to all participating processes. According to

Fig. 5.3(d), using the *Dynamic* algorithm, the leader assigns IPC or host-staged data copy mechanism to other processes through the `Copy Status Flags` to notify them how and from which shared buffer (CPU or GPU) they can read their collective results. If the *Dynamic* algorithm opts to use the host-staged data copy mechanism, the leader requires to first copy the result from the GPU shared buffer to the CPU shared buffer.

5.3.3 Cluster-wide Extension of the Static and the Dynamic Algorithms

To extend the node-wide Static and Dynamic approaches to across the cluster, we propose a general three-level hierarchical framework similar to those presented in Chapter 4. A node-wide collective algorithm, reduce for MPI_Allreduce or gather for MPI_Allgather, is first performed among processes that share the same GPU using the static or dynamic algorithm. A cluster-wide collective operation, allreduce for MPI_Allreduce or allgather for MPI_Allgather, is then performed among GPU leader processes on each node using the MVAPICH library. Finally, a node-wide collective algorithm, broadcast for both MPI_Allreduce and MPI_Allgather, is performed among processes that share the same GPU using the static or dynamic approach. In the following, we discuss the general steps involved in extending the *Static* and *Dynamic* approaches to across the cluster for MPI_Allreduce and MPI_Allgather.

MPI_Allreduce

The MPI_Allreduce with *Static* and *Dynamic* approach is performed in three stages as follows:

Stage 1: Intranode Intra-GPU Reduce Intranode intra-GPU processes use the *Static* or *Dynamic* approach to reduce the data into their predefined GPU leader

process.

Stage 2: Internode Inter-GPU Allreduce The GPU leader processes call MPI_Allreduce using the existing internode MVPAICH2 algorithm.

Stage 3: Intranode Intra-GPU Broadcast The GPU leader processes use the *Static* or *Dynamic* approach to broadcast the data among the intranode intra-GPU processes.

In order to extend and use the MPS service across the cluster, an instance of this service is required to be running on each node of the cluster. Fig. 5.4 shows how the MPS service can be used with the *Static* and *Dynamic* algorithms on MPI_Reduce. As shown in the figure, the MPS server on each node allocates one instance of the GPU storage and scheduling resources that can be shared by all intranode MPI processes, which are also called MPS clients. This way, all intranode processes can use their own instance of the MPS service to share the GPU within their node; therefore, they can concurrently access and share the GPU resources.

MPI_Allgather

To extend the *Static* and *Dynamic* algorithm to across the cluster in MPI_Allgather, this operation is performed in three stages as follows:

Stage1: Intranode Intra-GPU Gather Intranode intra-GPU processes use the *Static* or *Dynamic* approach to gather the data into their predefined GPU leader process.

Stage2: Internode Inter-GPU Allgather The GPU leader processes call MPI_Allgather using the default MVPIACH2 algorithm.

Stage3: Intranode Intra-GPU Broadcast The GPU leader processes use the *Static*

or *Dynamic* approach to broadcast the data among the intranode intra-GPU processes.

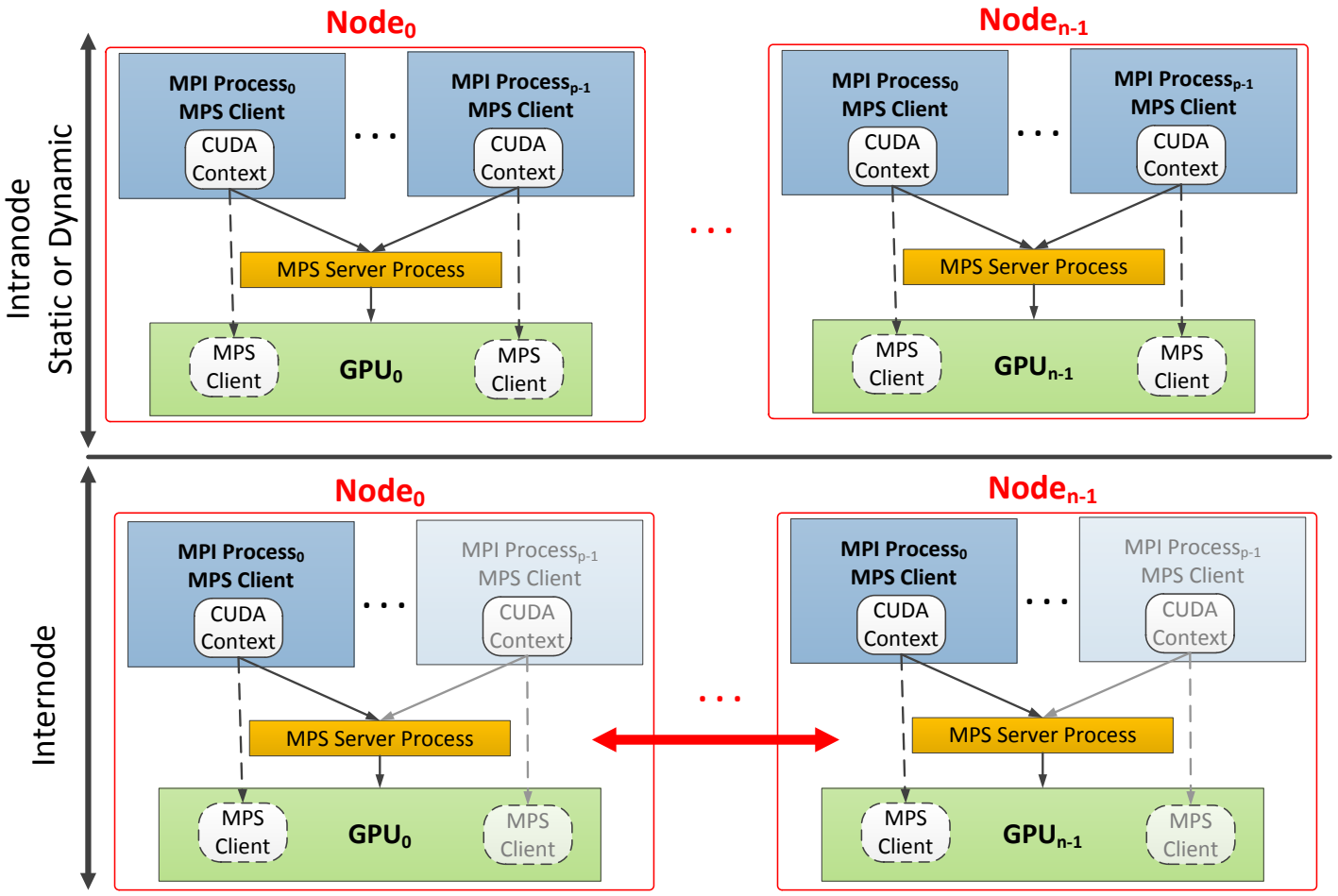


Figure 5.4: *Static* and *Dynamic* algorithms across the cluster for MPI_Reduce

5.4 Experimental Results and Analysis

In this section, we compare our Hyper-Q aware collective designs against MVAPICH2 and MVAPICH2-GDR and evaluate the effect of the MPS service on them. While our proposed algorithms can be applied to all collective operations, we consider

MPI_Allgather and MPI_Allreduce as our test cases. It is worth noting that MVAPICH2 and MVAPICH2-GDR fail to use the Nvidia MPS to perform MPI_Allgather for some message sizes and process counts. We speculate this failure to be rooted in using the FGP algorithm [84] in MVAPICH2. So for these test cases, we do not have any results to report.

In the rest of this section, we first provide our experimental platform; then, we discuss our results on a single-GPU platform and provide some profiling results to show how our Hyper-Q aware algorithms can successfully overlap different data copy mechanisms and improve the total communication performance. Finally, we will discuss the results on the GPU cluster.

5.4.1 Experimental Platform

Our experiments in this section are conducted on a 4-node GPU cluster (System C), called Odin, at the HPC Advisory Council. Each of the Odin nodes is equipped with 3 (or 4) K80 GPUs, 64 GB of memory, and two Intel Xeon E5-2697 processors. Each Xeon processor operates at 2.6 GHz and provides 14 cores; thus, each Odin node has 28 cores. Each node runs a 64-bit RHEL 7.2 as the operating system and utilizes the CUDA Toolkit 7.5. In our experiments, we compare our collective designs with the existing collectives in MVAPICH2-2.1 and MVAPICH2-GDR-2.0.

Evaluation of our Hyper-Q aware proposals require to frequently change the GPU mode in order to stop and start the MPS service. These actions require an interactive access to the GPU cluster and cannot be scheduled in a PBS script. Taking this into consideration, we solely provide our results on the Odin cluster to which we had dedicated access, unlike the other clusters that were available to us.

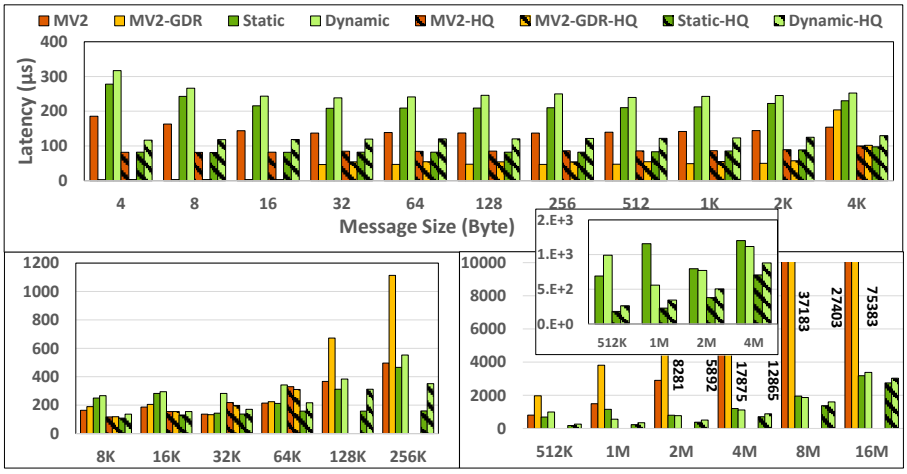
We conduct our experiments using the OSU microbenchmark that is configured to support GPUs [11]. We also provide some profiling results for our Hyper-Q aware designs with the OSU microbenchmark. We use the Nvidia Profiler (`nvprof`) [65] in conjunction with the Nvidia Tools Extension (`NVTX`) as our profiling tool. The `nvprof` presents an overview of the instructions launched by the CUDA runtime or driver APIs, whereas we used `NVTX` to annotate MPI routines and assign MPI ranks to their associated process ids and GPU contexts on the profiler timeline. Note that while we use the modified version of the OSU benchmark to get our profiling results, the original (unmodified) version of this benchmark is used to report the performance results.

5.4.2 Node-wide Experimental Results

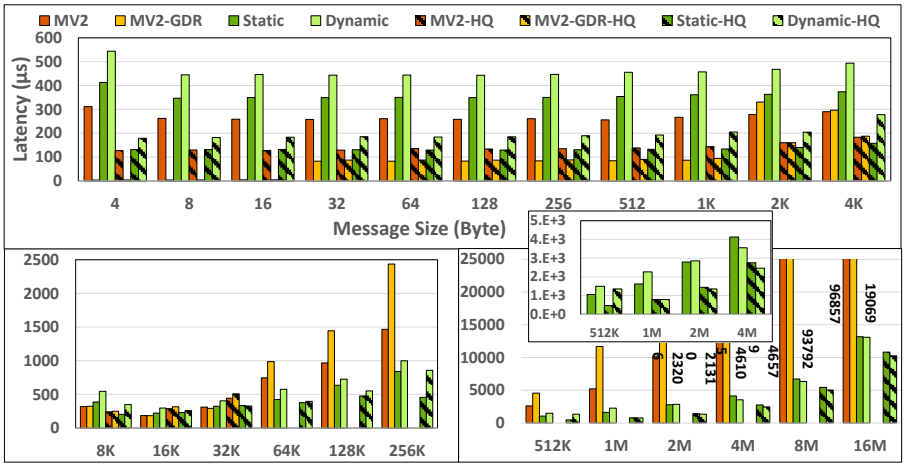
MPI_Allgather

In Fig. 5.5, we compare the *Static* and *Dynamic* approach against MVAPICH2 and MVAPICH2-GDR on MPI_Allgather. We also evaluate the effect of the MPS service on our results. According to Fig. 5.5, the benefit of the *Static* and *Dynamic* approach on 16 processes mainly starts at 4 KB and 8 KB, respectively. For message sizes below these thresholds, both the *Static* and *Dynamic* approach also provide competitive results with MVAPICH2 and MVAPICH2-GDR. The only exception to this is on the very short message sizes (less than 16 Bytes) of the MVAPICH2-GDR. For these message sizes, we speculate some features, such as gdrcopy, are the reasons behind the better performance of the MVAPICH2-GDR.

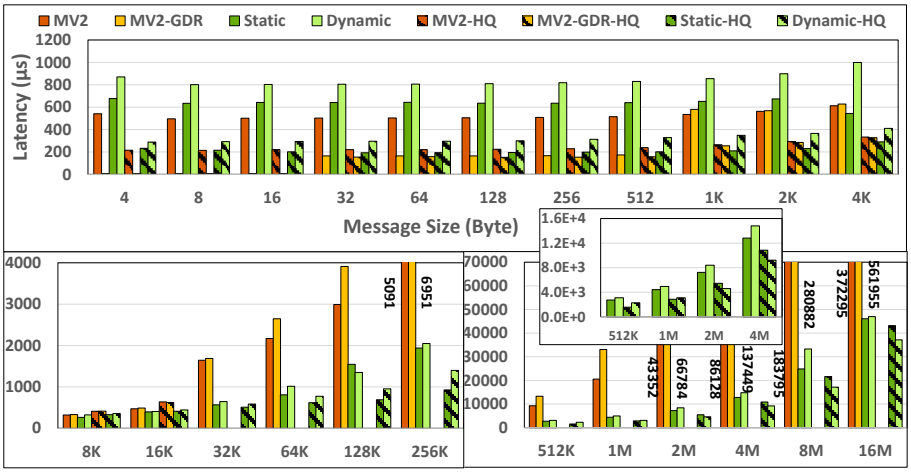
In Fig. 5.5, we can also observe that the performance of the *Dynamic* algorithm in most cases is comparable with the *Static* algorithm and there are a few cases in which



(a) MPI_Allgather on 4 processes



(b) MPI_Allgather on 8 processes



(c) MPI_Allgather on 16 processes

Figure 5.5: *Static* and *Dynamic* vs. MVAPICH2 and MVAPICH2-GDR MPI_Allgather w and w/o the MPS on a single node of Odin cluster with a single GPU per node

the *Dynamic* algorithm can outperform the *Static* algorithm. We associate this to the way that the tuning table is constructed for the *Static* algorithm. This table stores integer values for different configurations of the collective operation. These integer values represent the number of data copy mechanisms to be used in a collective operation and are the rounded average of a thousand runs. For instance, we use 11 (10.8 rounded up) host-staged copies and 4 (4.2 rounded down) CUDA IPC copies for 64KB message size and 16 processes. While the *Static* algorithm in different runs always stick to these rounded numbers, the *Dynamic* algorithm decides the number and mechanism of the copy within each run. Consequently, the *Dynamic* approach has the potential to be more accurate in choosing the right number and mechanism of the copies across multiple runs. We can also observe that all approaches in most cases are benefiting from the MPS service. The only exception to this is the case of MVAPICH2-GDR with small message sizes. In this case, the MPS service provides no improvement or even sometimes adversely affect the performance. In general, the *Static* and *Dynamic* approach benefit the most from the MPS service and on average achieve 2.17× and 2× speedup, respectively.

MPI_Allreduce

Fig. 5.6 compares our *Static* and *Dynamic* approach against MVAPICH2 and MVAPICH2-GDR for MPI_Allreduce. The benefit of the *Static* and *Dynamic* approach starts at 32 Bytes and 4 KB on 16 processes, respectively. For message sizes below these thresholds, (similar to our results for MPI_Allgather) both the *Static* and *Dynamic* approach provide competitive results with the MVAPICH2 and MVAPICH2-GDR. The only exception to this is on the very short message sizes (less than 16 Bytes) of the MVAPICH2-GDR. In general, the *Dynamic* approach provides

competitive results compared to the *Static* approach. The overhead of the *Dynamic* approach also decreases as the message size increases. There are also a few cases in which the *Dynamic* approach can outperform the *Static* approach.

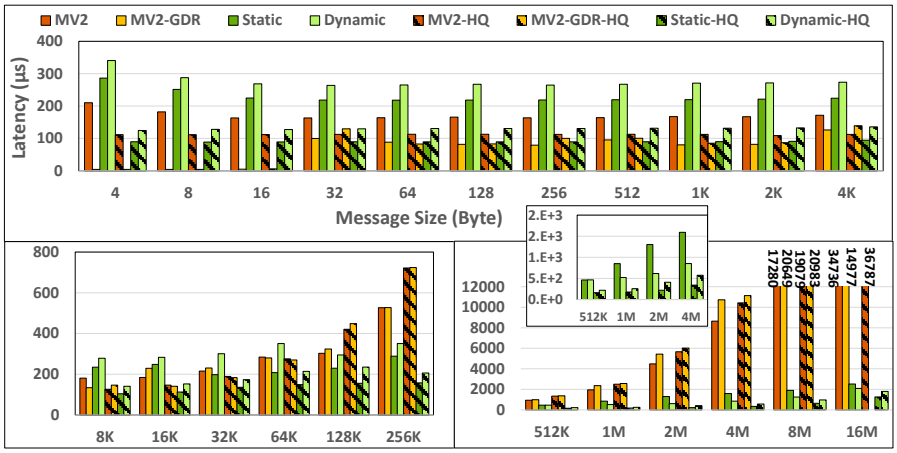
According to Fig. 5.6, all approaches in most cases are benefiting from the MPS service. The *Dynamic* approach on average benefits more from the MPS service (2.62×), compared to the *Static* approach (2.49×).

Profiling results

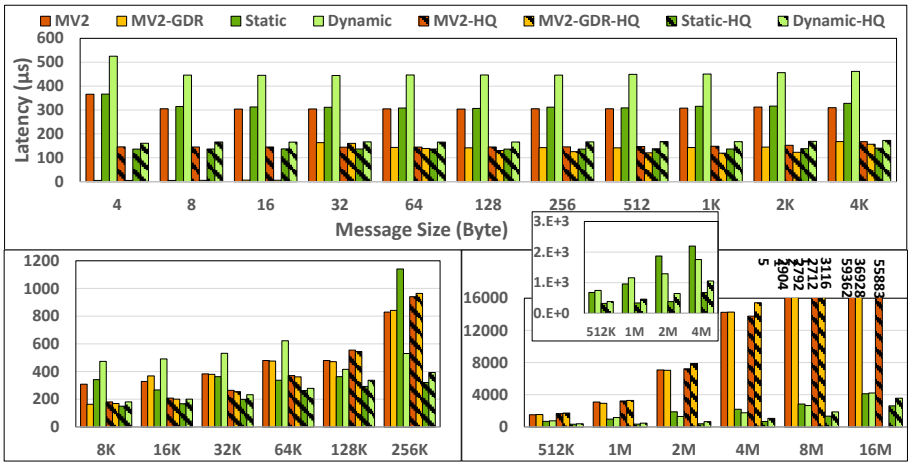
In this section, we discuss the main reasons behind the benefit of our Hyper-Q aware algorithms. In this regard, we shed some light on how our Hyper-Q aware algorithms work with the MPS service by providing some profiling results.

The GPU computational kernels and memory operations are performed using the compute and memory engines, respectively. Without the MPS service, each engine can be assigned to a single process at a time and cannot be shared among them. A time sliced scheduler is used on the GPU to handle the requests from different processes to these engines. With the MPS service, however, requests for accessing the GPU engines are funneled through the MPS server through the only available context on the GPU (i.e., the MPS context); consequently, there will be no need for any context switching.

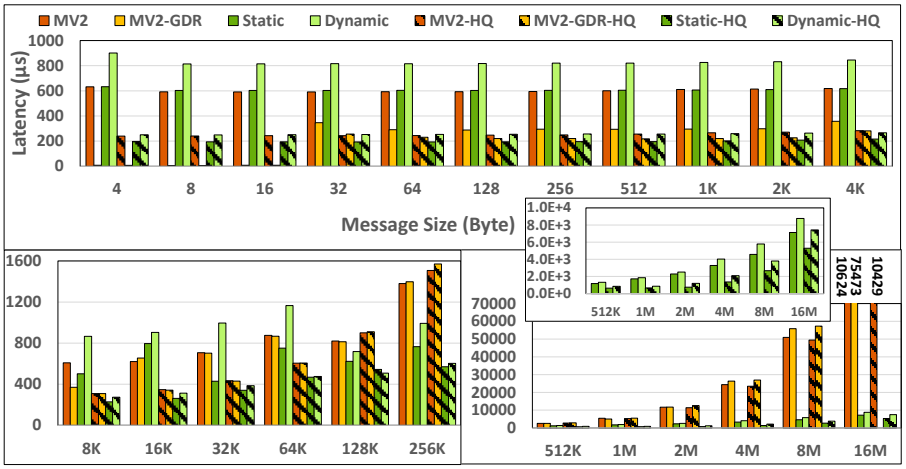
Profiling MPS service on multiple processes communicating with the host-staged copies reveals that not only is the context switching overhead eliminated, device-to-host and host-to-device copies from different processes can also further overlap with each other. While the MPS server allows computational kernels from different MPI processes to share the compute engine on the GPU and overlap, we observed that



(a) MPI_Allreduce on 4 processes



(b) MPI_Allreduce on 8 processes



(c) MPI_Allreduce on 16 processes

Figure 5.6: *Static* and *Dynamic* vs. MVAPICH2 and MVAPICH2-GDR MPI_Allreduce w and w/o the MPS on a single node of Odin cluster with a single GPU per node

with this service, among the three data copy mechanisms (i.e., host-to-device, device-to-host, and device-to-device), device-to-host and host-to-device engines cannot be shared among different MPI processes.

The MPS service highly improves the IPC copies by reducing the context-switching overhead. It also allows various CUDA IPC copies to share their local memory bandwidth and overlap their device-to-device communications. The IPC copies can also overlap with the host-staged copies. Our Hyper-Q aware designs select the right number and mechanism of the inter-process copies to provide the maximal overlap among them through the MPS service. In Fig. 5.7, we reflect this behavior by profiling the MPI_Allreduce operation (on 16 processes and 128 KB of data) that is implemented by our *Dynamic* algorithm. This figure is the output of the `nvvp` visual profiler that provides a runtime snapshot of the MPI_Allreduce operation with the MPS service. The profiling information in this figure is gathered using `nvprof` and `nvtx` profiling tools. According to the figure, the *Dynamic* algorithm selects 2 host-staged and 13 IPC copies in the Gather step, and 3 host-staged and 12 IPC copies in the Broadcast step of the MPI_Allreduce operation, respectively. We can also observe that with the MPS service, different inter-process copies with the same or different data copy mechanisms can overlap with each other.

5.4.3 Cluster-wide Experimental Results

MPI_Allgather

Fig. 5.8 depicts cluster-wide comparative results for the *Static* and *Dynamic* algorithm against the MVAPICH2 and MVAPICH2-GDR using MPI_Allgather. According to the figure, the *Static* approach in most cases provide comparable results

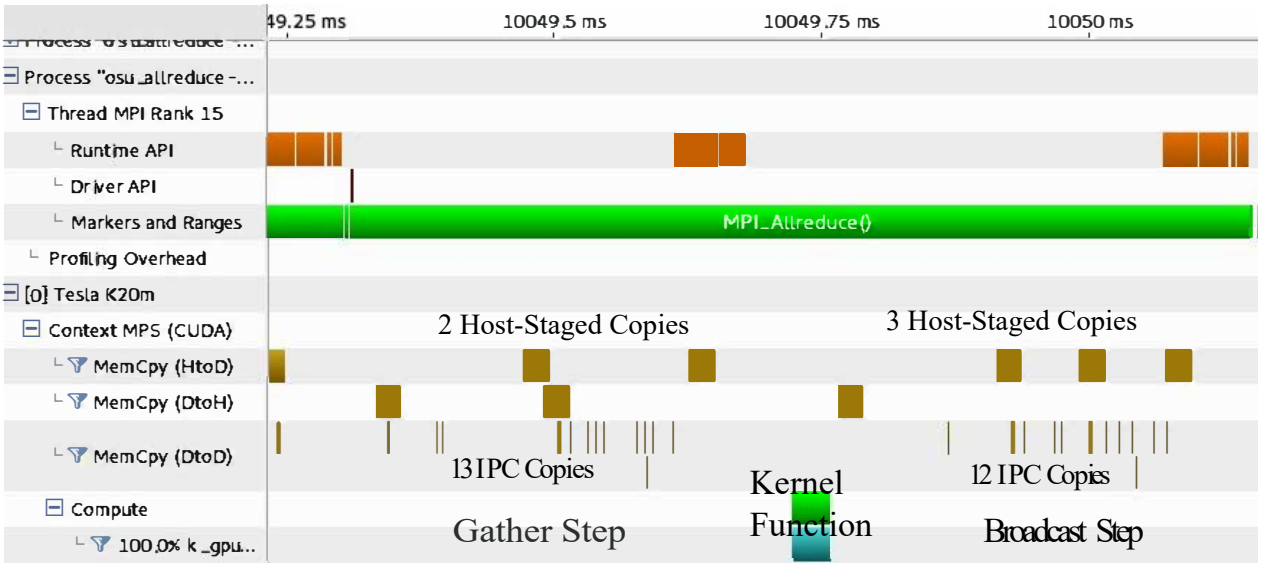
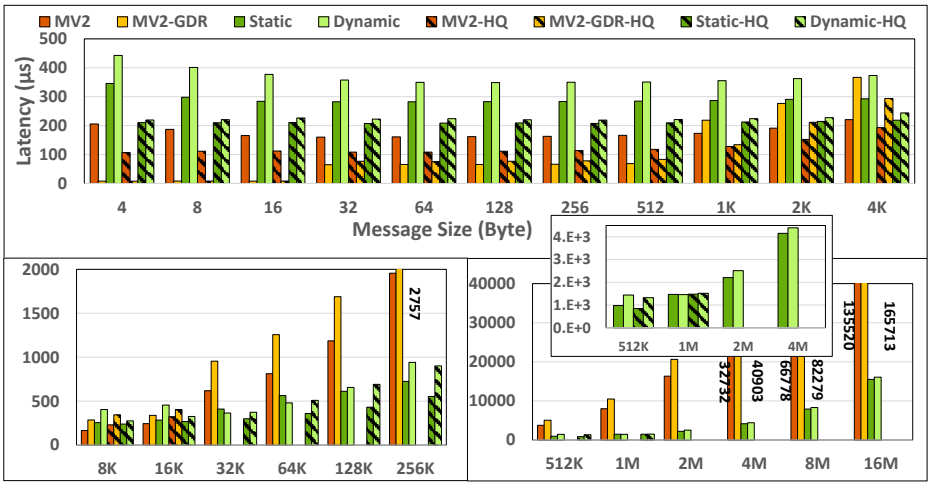


Figure 5.7: Profiling snapshot of the *Dynamic* algorithm in MPI_Allreduce with MPS

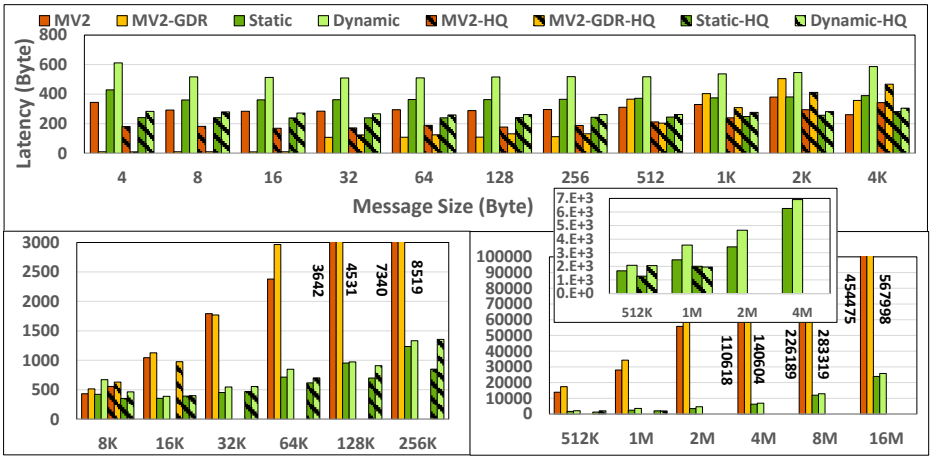
or outperform MVAPICH2 and MVPAICH2-GDR (except for message sizes less than 16 Bytes on MVAPICH2-GDR). The benefit of the *Static* and *Dynamic* approach can be better realized as either the message size or the number of processes per node increases. More specifically, the benefit of the *Static* approach starts at 16KB, 8KB, and 512 Bytes for 4, 8, and 16 processes, respectively. The *Dynamic* approach in most cases also provides comparable results with the *Static* approach.

MPI_Allreduce

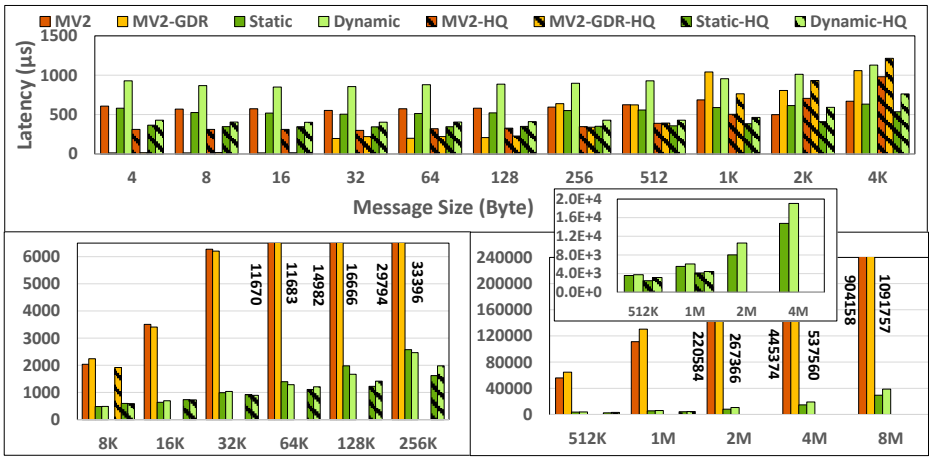
Fig. 5.9 illustrates the cluster-wide comparaitve results for the *Static* and *Dynamic* algorithms against MVAPICH2 and MVAPICH2-GDR using MPI_Allreduce. According to the figure, the *Static* approach in most cases outperforms the MVAPICH2 and MVPAICH2-GDR and for other cases provide comparable results (except for message sizes less than 16 Bytes on MVAPICH2-GDR). The *Dynamic* approach also in most cases provide a comparable result with the *Static* approach. In general, we can observe that the extent of the improvement of our Hyper-Q aware approaches



(a) MPI_Allgather on 16 processes - 4 processes per node



(b) MPI_Allgather on 32 processes - 8 processes per node



(c) MPI_Allgather on 64 processes - 16 processes per node

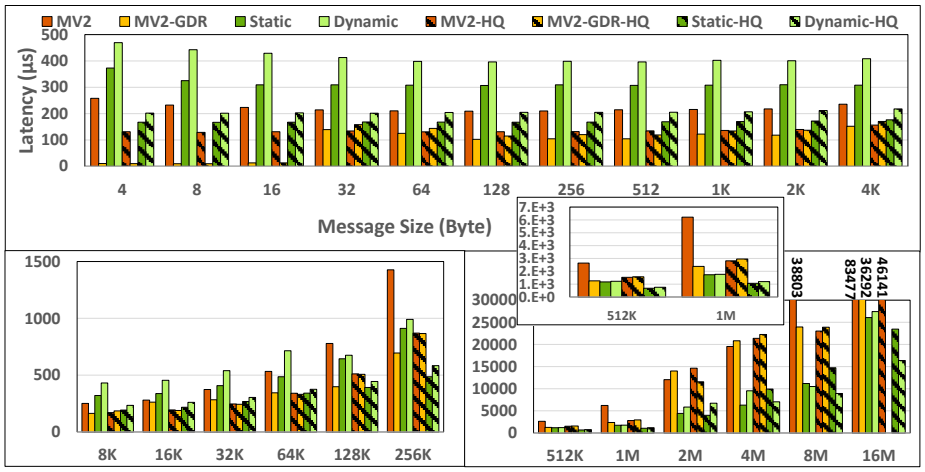
Figure 5.8: Comparison of *Static*, *Dynamic*, MVAPICH2, and MVAPICH2-GDR using MPI_Allgather w and w/o the MPS on 4 nodes with a single GPU per node

is higher for MPI_Allreduce, compared to MPI_Allgather. We associate this to the benefit that is provided by using GPU kernel functions in performing reduction operations in MPI_Allreduce.

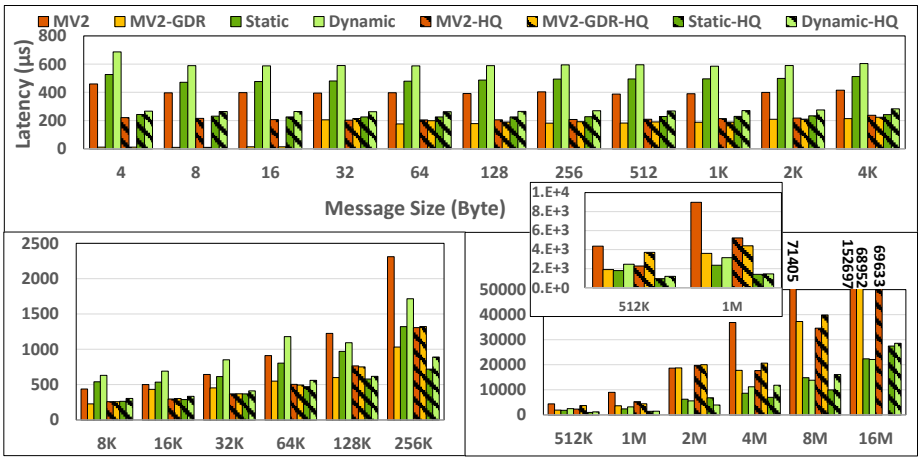
5.4.4 Comparative Analysis of Hyper-Q Aware Algorithms against GSB/BTB Algorithms

To further investigate our Hyper-Q aware proposals and evaluate their efficiency with the presence of the MPS service, we compare them against our Hyper-Q agnostic collective algorithms in Chapter 3 (i.e., the GSB and the BTB). In this regard, Fig. 5.10 provides the speedup achieved by using the *Static* design over MVAPICH2, MVAPICH2-GDR, GSB, BTB and *Dynamic* design in MPI_Allreduce. This experiment is conducted on 64 processes that are evenly distributed among 4 single-GPU nodes of the Odin cluster. As shown in the figure, the *Static* approach in most cases outperforms the rest of the designs, with few exceptions.

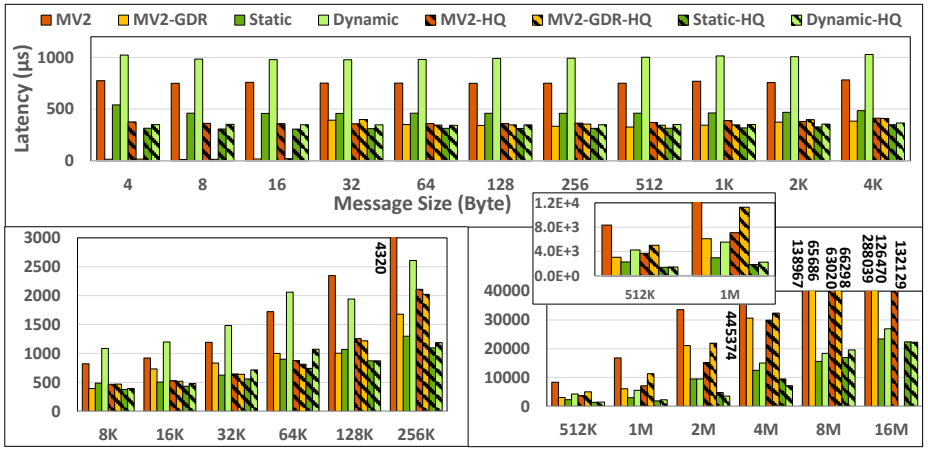
The results in Fig. 5.10 confirm our findings in Fig. 3.6, that while our GPU-aware collective communication algorithms (GSB and BTB) are capable of outperforming the conventional MVAPICH2 design for large message sizes, they fall behind for small and medium message sizes. We attribute this to the high startup overhead of the IPC copies for small and medium message sizes. However, our Hyper-Q aware designs can rectify this problem by selecting the right number and mechanism of the copies for different message sizes and process counts. In addition, the *Dynamic* approach, by dynamically selecting the right number and mechanism of the copies, is capable of providing comparable results with the *Static* approach in most cases.



(a) MPI_Allreduce on 16 processes - 4 processes per node



(b) MPI_Allreduce on 32 processes - 8 processes per node



(c) MPI_Allreduce on 64 processes - 16 processes per node

Figure 5.9: Comparison of *Static*, *Dynamic*, MVAPICH2, and MVAPICH2-GDR using MPI_Allreduce w and w/o the MPS on 4 nodes with a single GPU per node

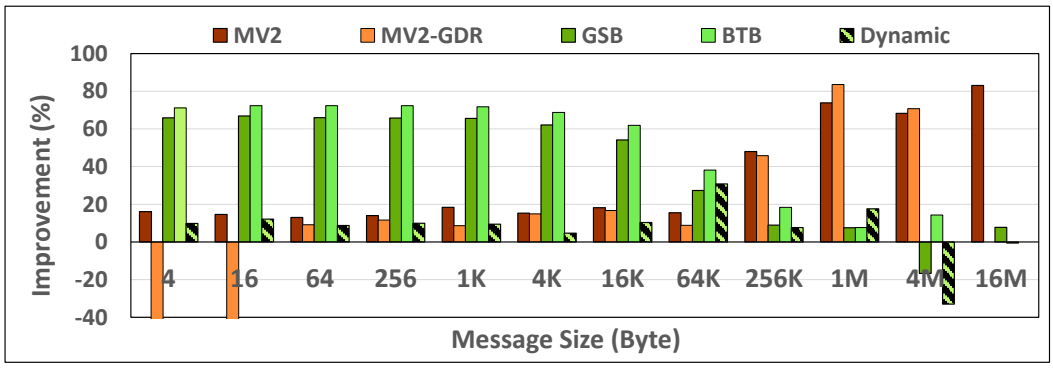


Figure 5.10: Improvement percentage of the *Static* approach over MVAPCIH2, MVAPICH2-GDR, GSB, and BTB for MPI_Allreduce with 64 processes using MPS - System C with 4 nodes and a single GPU per node

5.5 Provision of Using Our Proposals with Future GPU Accelerators

In this chapter, we showed the benefit of cooperatively using different data copy mechanisms in performing multiple inter-process communications. Accordingly, we proposed Hyper-Q aware designs for GPU collective operations that cooperatively use different data copy mechanisms to perform inter-process communications. We also analyzed the MPS service on various collective designs and showed that our Hyper-Q aware proposals can benefit the most from this service.

Our Hyper-Q aware designs in this chapter utilize two of the existing intranode inter-process data copy mechanisms (i.e., CUDA IPC and host-staged copy). However, our designs are not dependent to any specific data copy mechanism. In general, our proposals can be applied to any combination of different copy mechanisms as long as they provide different means for inter-process communications and have the potential to overlap.

As an example, in some multi-GPU nodes, QPI is used for GPU inter-socket inter-process communications. However, given the slow nature of this communication

channel in some architectures, network-assisted loopback copy may be preferred. In the loopback copy, the intranode communication goes through network interface cards and loopbacks to the remote GPU on the same node, thus avoiding potentially slow inter-socket connections. Our Hyper-Q aware designs can potentially use the loopback and QPI data copy mechanisms in conjunction with each other when multiple inter-socket GPU communications are on the fly, leading to improvements in the total communication performance.

The next generation of the Nvidia GPUs (codenamed Pascal) benefit from a high-bandwidth interconnect, called NVLink. Fig. 5.11 (adapted from the NVLINK configuration of the Pascal architecture [66]) illustrates an example multi-GPU architecture that is equipped with NVLink interconnects. The four GPUs in this figure are interconnected to the CPU and also together using the existing PCIe communication channel (with 16 GB/s peak uni-directional bandwidth). Moreover, the GPUs are also interconnected together with NVLink connections. According to the figure, the GPUs can use one or two NVLink connections (each NVLink connection has 20 GB/s peak uni-directional bandwidth) to reach each other. In such node, inter-process communication using the conventional designs would only consider a single communication channel path between the GPU peers (for example, *PATH1* will be only used for inter-process communications between GPU0 and GPU1). On the other hand, using our proposed Hyper-Q aware designs, all of the available communication channels will be assessed and potentially used in multiple inter-process communications. For instance, using our Hyper-Q aware proposals multiple inter-process communications between GPU0 and GPU1 can be potentially routed through *PATH1*, *PATH2*, and *PATH3*.

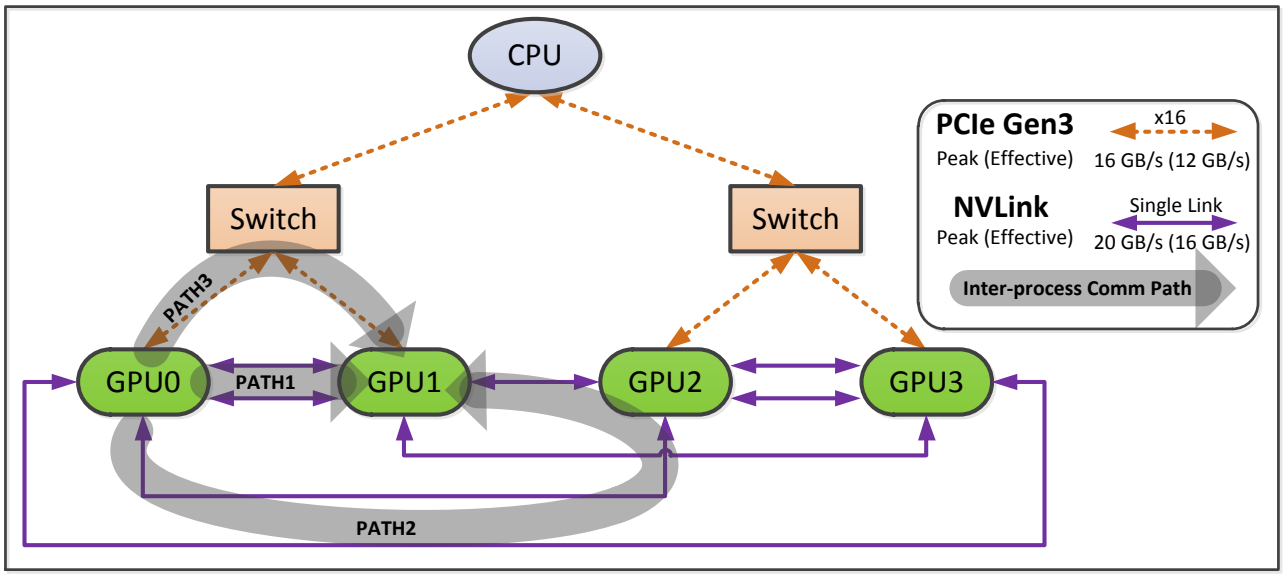


Figure 5.11: Different intranode communication channels of a 4-GPU node with NVLink and PCIe (adapted from [66])

As of today, NVIDIA provides the *CUDA_DISABLE_NVLINK_MAPPINGS* environment variable to enable/disable the use of NVLINK communication channel; using this environment variable, it is possible to force processes to either use NVLINK or PCIe communication channel prior to the runtime; however, forcing processes to avoid a communication channel is not yet supported during runtime. While the potential for overlapping inter-process GPU communications between NVLINK and PCIe communication channels exists, the use *Static* or *Dynamic* algorithms require support from the runtime libraries. In summary, with our proposed *Static* and *Dynamic* designs, no matter the mechanism of the GPU and the architecture of the node, with the right support from the runtime libraries, different communication channels can be assessed and used cooperatively to perform multiple GPU inter-process communications and speed up the total communication performance.

We have also observed the benefit of using the MPS service with our Hyper-Q

aware designs. It is noteworthy to mention that as the GPU architecture evolves, it often inherits the GPU features from its previous generations. In particular, the MPS service or a similar services should exist in the upcoming GPU generation, as it allows multiple processes to share the GPU resources and tackle the GPU underutilization. Whether the MPS service remains available or gets replaced with another technology, the ability to share a single GPU by multiple processes will remain an important asset in future GPU generations.

5.6 Summary

For GPU inter-process communications different data copy mechanisms with different performance characteristics can be used. Different data copy mechanisms are usually favored for different message sizes. However, we observed the benefit of jointly using them when performing multiple inter-process communications. This way, different data copy mechanisms can overlap with each other and speed up the total inter-process communications. Accordingly, we proposed two Hyper-Q aware MPI_Allreduce algorithms for GPU collectives: 1) *Static Hyper-Q aware*; and 2) *Dynamic Hyper-Q aware*. Both designs jointly utilize CUDA IPC and host-staged data copy mechanisms in their collective operation. However, they use different mechanism of information to decide their data copy mechanisms to perform inter-process communications.

We evaluated the effect of the MPS service on our proposed algorithms. The MPS service can allow different MPI processes to further overlap with each other and more efficiently share single GPU resources. Most of the collective designs showed to benefit from this service, however we achieved the highest improvement with our

Hyper-Q aware algorithms.

We analyzed our designs using MPI_Allgather and MPI_Allreduce. Unlike our collective designs in Chapter 3, our designs in this chapter by selecting the right data copy mechanism can provide improvement across all message sizes. In general, the *Static* approach provides higher improvement compared to the *Dynamic* approach. However, the *Dynamic* approach in most cases provide comparable performance improvement. The *Dynamic* approach also has the advantage of being independent of any tuning parameter and thus can be portable across different platforms.

In Chapter 3, we introduced our GPU-aware algorithms for collective communications targeting single-GPU nodes and clusters. In Chapter 4, we proposed a hierarchical framework for GPU collectives and evaluate the sensitivity of different algorithms to different hierarchy levels in multi-GPU nodes and clusters. In this chapter, we proposed various designs to intelligently select the right data copy mechanism in collective operations targeting single-GPU nodes. In all of these chapters, our proposals have targeted improving the performance of specific GPU communication operations. In Chapter 6, on the other hand, we propose topology-aware designs to improve the total communication efficiency of HPC applications running on multi-GPU nodes and clusters.

Chapter 6

Topology-aware GPU Communications

The node architecture of modern GPU clusters usually consist of multi-core processors and multiple GPU devices interconnected by a hierarchy of different communication channels. In such clusters, not only will the intranode and internode inter-process communications traverse different paths, but also different intranode inter-process communications may have different traversal paths. We show that in a multi-GPU node, not only should one expect heterogeneity in terms of the computational power and characteristics of different processing units (i.e., CPUs and GPUs), but also an added heterogeneity in terms of the topology and performance of different communication channels used to interconnect them.

Our goal in this chapter is to propose designs that can ultimately lead to efficient utilization of different communication channels to improve the performance of inter-process communications in GPU clusters. To do so, we propose designs to lead MPI processes to intelligently select GPUs (among the available GPUs that are visible to them) in a way that more intensive inter-process GPU communications take place on the more efficient communication channels. In this regard, in this chapter we make the following contributions:

- We first report the latency and bandwidth results of different intranode GPU communication channels in a multi-GPU node and show that they can considerably vary from each other. We also provide similar analysis for different GPU and CPU communication channels across the GPU cluster. Our results show that the performance and the number of various CPU and GPU communication channels can be significantly different as well.

- We propose a GPU assignment policy that would take into account the GPU communication pattern and the physical topology of the multi-GPU node to improve the communication performance among the GPUs [25, 26]. We have integrated the proposed GPU scheme into the Open MPI library. To the best of our knowledge, this is the first application of topology awareness for GPU to MPI process assignment on multi-GPU nodes. We use three metrics (latency, bandwidth, and communication distance) to reflect the performance of different intranode GPU communication channels in a multi-GPU node.

- We propose a GPU assignment scheme which considers different computational entities within a heterogeneous cluster [55]. More specifically, we argue that in order to extend our GPU assignment scheme to across the cluster, one should consider both the CPU and GPU communications, as well as their physical topologies. In this regard, we propose a three-level hierarchical mapping scheme. Our proposed scheme breaks down the mapping into three distinct phases: 1) internode process-to-node mapping; 2) intranode process-to-CPU-core binding; and 3) intranode process-to-GPU assignment.

- To evaluate our GPU assignment scheme on multi-GPU nodes, we develop

three microbenchmarks and use one real application. The microbenchmarks model three of the commonly used communication patterns in parallel applications. Namely, they model 2D and 3D stencil patterns, as well as alltoall communications over sub-communicators. For application evaluations, we use HOOMD-Blue [3, 27] which is a general-purpose toolkit used for molecular dynamics simulations. The results show that our topology-aware GPU selection scheme can highly outperform the default selection scheme. We can achieve up to 72% and 21% improvement in performance at the microbenchmark and application levels, respectively.

- We provide a comprehensive analysis of our experimental results. In this regard, we evaluate both the single- and double-precision versions of HOOMD-Blue with different input benchmarks and particle sizes. We also provide detailed profiling analysis of our single-node results. In this regard, we extend the FPMPI library [29] and build a tool for profiling both CPU and GPU communications. We apply our profiling tool into HOOMD-Blue in order to provide further insights into the extent of improvements that can be achieved with our proposed topology-aware GPU selection scheme.

- We also use both microbenchmarks and application analysis in our cluster-wide experiments. In this regard, we develop a benchmark suite with a set of microbenchmarks that represent different communication patterns used in scientific applications. We designed this microbenchmark suite in a way to allow concurrent inter-process communications among CPUs and GPUs with different communication patterns. Our experimental results show that our GPU selection scheme can highly improve the communication performance of these

microbenchmarks. We also show the effect of our GPU selection schemes on an end-point application: HOOMD-Blue. According to our microbenchmark results, our proposed GPU selection scheme can improve the total benchmark runtime by 90% across the GPU cluster. Our application results also show up to 8% performance improvement across the GPU cluster.

6.1 Motivation

In this section, we discuss various topology levels with different communication channels in a multi-GPU node and across the GPU cluster, and also evaluate their corresponding latency and bandwidth.

6.1.1 Impact of CPU/GPU Topology Levels on CPU/GPU Communication Performance in a Cluster with multi-GPU nodes

In a multi-GPU node, various traversal paths may exist among different GPUs. Therefore, communication between different GPU pairs may go through different communication channels. Fig. 6.1 depicts an example configuration of a multi-GPU node which we also use as our experimental testbed (Section 6.4). As shown in the figure, GPUs can communicate with each other through different paths. In general, communication path between different GPU pairs can traverse four topology levels which we refer to as **Level 0**, **Level 1**, **Level 2**, and **Level 3**.

At **Level 0**, the communication path between GPU pairs traverses a PCIe internal switch; this path exists among the on-board GPUs of a single GPU accelerator (e.g., path between **GPU 0** and **GPU 1**). Communication path at **Level 1** goes through multiple PCIe switches (e.g., path between **GPU 0** and **GPU 2**). At **Level**

2, communication path crosses a root complex[1] (*RC*) device (e.g., path between **GPU 0** and **GPU 4**); *RC* connects the PCIe switch fabric to the socket. Communications at **Level 3** goes through an inter-socket (*IS*) link such as Intel QPI (e.g., path between **GPU 0** and **GPU 8**) [33]. Such a variety of communication channels at different topology levels can result in different GPU communication latency and bandwidth. To evaluate the latency and bandwidth characteristics of various intranode GPU and CPU topology levels, we perform intranode ping-pong latency and uni-directional bandwidth tests. More specifically, the ping-pong latency provides the half-way round-trip data transfer latency between two GPUs. In uni-directional bandwidth test, we measure the data transfer rate between two processes in number of Bytes per second.

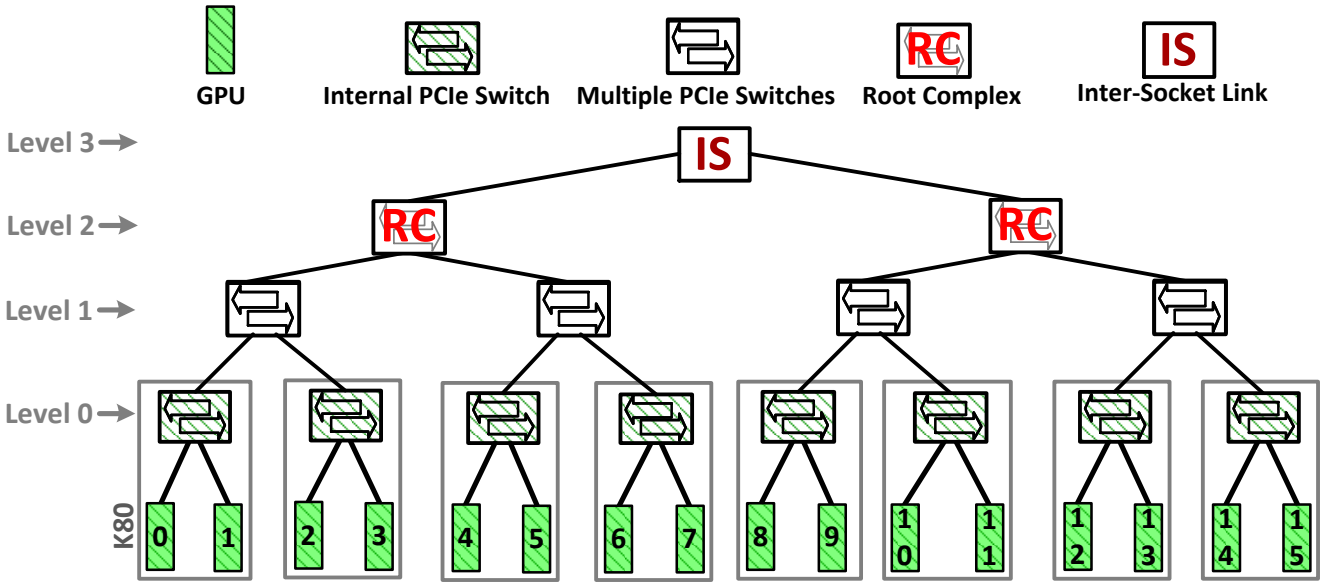


Figure 6.1: Different intranode GPU pair levels. This is also the topology of the K80 GPU node used in our experiments

[1]The root complex is a part of the hostbridge logic; in the sequel, we use the root complex and hostbridge terms interchangeably.

In Fig. 6.2 and 6.3, we provide the latency and bandwidth results on different CPU and GPU topology levels, respectively. We perform our experiments both within and across the cluster with multi-GPU nodes using two Helios nodes (System B). The GPU latency and bandwidth analyses are conducted by first performing the latency and bandwidth tests for all possible GPU pairs in our experimental platform. Each test is performed one at a time and measures the latency (or bandwidth) across various message sizes for a single-GPU pair. Next, depending on the GPU topology level, we categorize the latency and bandwidth results into five GPU topology levels, and report the average values in Fig. 6.2 and Fig. 6.3, respectively. Similar to the GPU experiments, the CPU latency and bandwidth experiments are conducted and reported; however, unlike the GPU latency and bandwidth tests, the CPU tests are conducted among the CPU core pairs and categorized into three CPU topology levels.

According to Fig. 6.2 and 6.3, the internode topology level is shown to have different latency and bandwidth characteristics compared to the intranode topology levels. We can observe that the number and performance of the CPU topology levels vary from the GPU topology levels. The only exception is the bandwidth of the internode CPU and GPU topology levels that provide similar results. At this level, the same interconnection network (i.e., Mellanox QDR InfiniBand) is used for data network transfer both between the CPUs, and between the GPUs. However, the latency of these topology levels are still different. In general, the GPU communications have higher initialization cost compared to the CPU communications.

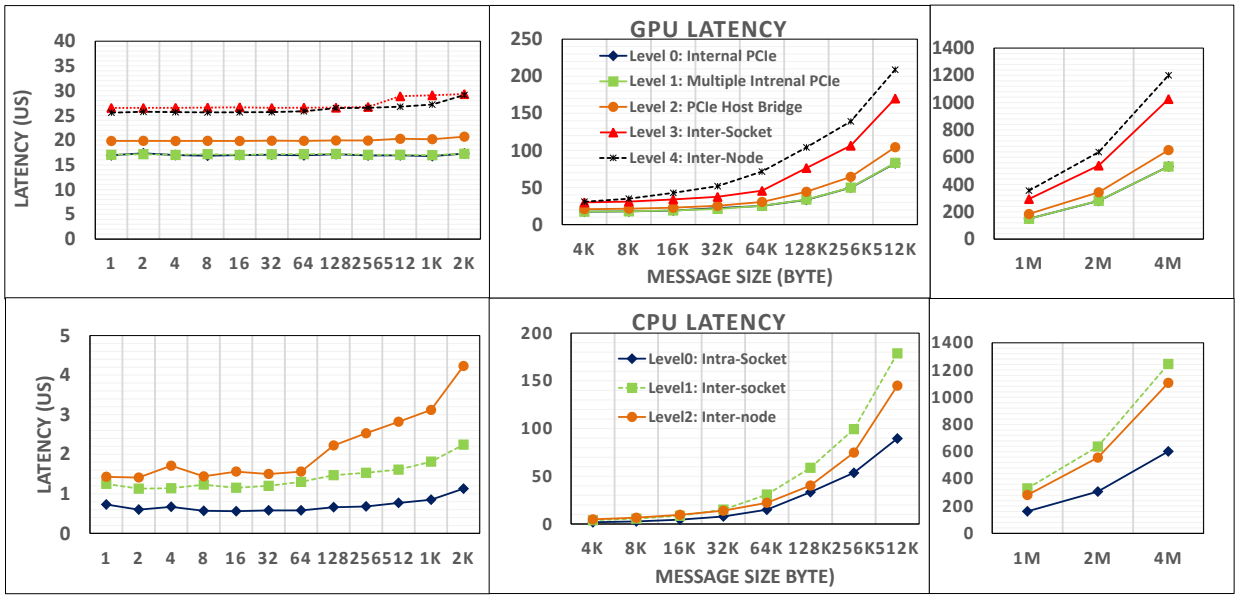


Figure 6.2: Impact of internode and intranode CPU and GPU topology level on ping-ping latency in a GPU cluster

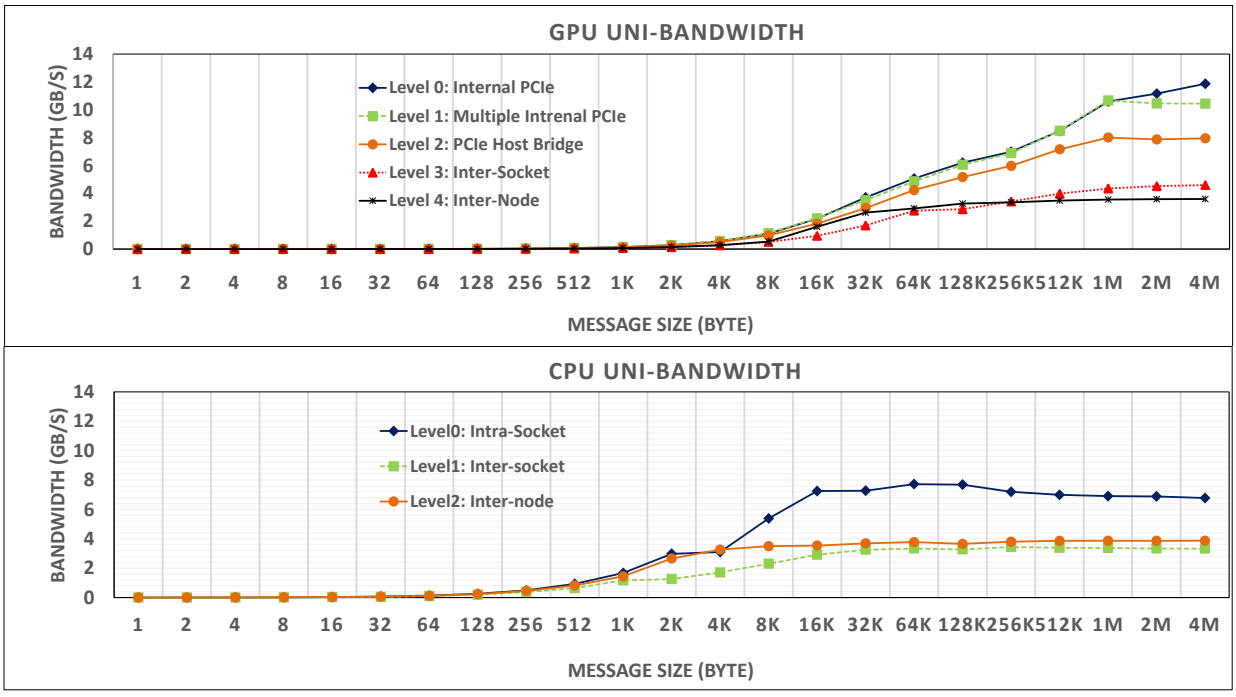


Figure 6.3: Impact of internode and intranode CPU and GPU topology level on uni-directional bandwidth in a GPU cluster

6.2 Related Work

In GPU clusters, GPU communications play a crucial role in the performance of MPI applications. In this regard, researchers have studied various GPU-aware point-to-point and collective operations to improve the GPU communication performance [39, 73, 85]. The CUDA IPC data copy mechanism is used in [39, 73] to improve one-sided and point-to-point communications. Singh et al. proposed designs to improve the performance of the MPI_Alltoall operation on the GPU. While these studies target to improve the efficiency of specific MPI routines, our goal in this chapter is to improve the overall MPI communication runtime among GPUs by efficiently assigning them to MPI processes.

Martinasso et al. [50] provide a detailed analysis of the congestion behavior associated 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 intranode GPU communications. Lutz, et al. [48] propose an autotuning framework for distribution of stencil computations across multiple GPUs. They show that various PCIe layouts could have adverse effects on the performance, thereby utilizing all GPUs might not be necessarily a better choice in all cases. Similarly, we show that different communication channels among GPUs can have different performance characteristics, however we exploit this to propose a topology-aware GPU assignment for MPI processes.

Topology-aware mapping has been extensively studied in the context of CPU-to-CPU communications. Several experiments carried out on large-scale systems such

as IBM BG/P and Cray XT supercomputers have verified the adverse effects of contention and hop-count on message latencies [6]. Another study [4] shows that different mappings of an application on large-scale IBM BG/P systems can significantly affect the overall performance. Rashti, et al. [77] proposed a topology-aware mapping mechanism for two of the MPI topology functions, i.e., `MPI_Graph_create` and `MPI_Cart_create`; the authors used the network distance to model the physical topology. Ito, et al. [36] proposed a similar mapping algorithm, but uses actual bandwidth measurements to model the physical topology. Mércier and Jeannot [53] modified the implementation of the `MPI_Dist_graph_create` function in MPICH2 to provide it with a topology-aware reordering of processes. Mirsadeghi and Afsahi [54] proposed a parallel mapping approach that takes into account the underlying routing mechanism in addition to topology. Rodrigues, et al. [79] also used bandwidth measurements to model the physical topology of the target system. In this chapter, unlike these previous work, we exploit topology awareness and mapping concepts in the context of GPU communications, and show how a topology-aware GPU selection scheme can help to improve the performance of communications on a multi-GPU node and across the cluster.

6.3 Improving GPU Communication by Efficient GPU Assignment Schemes

As the number of GPUs within a multi-GPU node increases, the topology of the GPU interconnects becomes more hierarchical, effectively increasing the heterogeneity of the GPU communication channels. Taking this into consideration, in this section we first propose a topology-aware GPU selection scheme for a multi-GPU node. Using our scheme, GPUs can be efficiently assigned to the processes. Next, we discuss

the extension of our proposed scheme to across the cluster. With our cluster-wide scheme, processes can be efficiently mapped to CPU cores and GPUs assigned to the processes, thus improving the total inter-process communication performance.

6.3.1 GPU Assignment Scheme on a Multi-GPU Node

We model our proposed scheme in this section as a graph mapping problem where the GPU communication graph is mapped onto the GPU physical topology graph. A given solution of this mapping problem would designate a specific assignment of GPUs to MPI processes.

We extract the GPU communication pattern by profiling the application in an initial run. In this regard, we instrument the MPI library to gather the GPU inter-process communications, and save it into a square matrix. The matrix captures the total volume of GPU messages transferred between each pair of processes. Using this matrix, we construct a GPU virtual topology graph, representing the application's GPU communication pattern. In this graph, vertices stand for MPI processes, and the weighted edges represent the existence and significance of GPU communications among each pair of processes. Thus, the higher the edge weight, the higher the communication volume between the associated GPU peers.

We use three different metrics to reflect the impact of different topology levels in a multi-GPU node: 1) latency; 2) bandwidth; and 3) distance. For each metric, we perform a series of tests to extract the associated physical topology matrix file (note that none of the tests require root access). Using the generated files, we construct the GPU physical topology graph. Vertices in this graph represent the GPU device indices, and the edges represent the strength of the connection between two GPUs.

All intranode GPUs are capable of communicating with each other, thus the GPU physical topology graph will be a complete graph. A higher edge value represents a lower latency, higher bandwidth, and lower communication distance in the latency-based, bandwidth-based, and distance-based physical topology graphs, respectively.

In this work, we use the SCOTCH library [69] to map the constructed virtual topology graph onto the physical topology graph. SCOTCH is a graph mapping library in which a guest graph G is mapped onto a host graph H. The problem is known to be NP-hard, and SCOTCH library and other mapping libraries (such as METIS [41], and ParaMETIS [42]) provide sub-optimal solutions for that. SCOTCH, in particular is capable of mapping a given source graph onto a given target graph with any topology, and with weighted or non-weighted vertices and edges.

We also use the SCOTCH library to construct the GPU virtual and physical topology graphs out of the virtual and physical matrix files. These matrix files are required to be available prior to the application execution. The virtual topology matrix file is generated once in an initial profiling run. The physical topology matrix file is created for each target multi-GPU node.

Depending on the chosen metric, we use different tests to generate the physical matrix file. For the latency-based metric, we use the normalized GPU pair latency values from the ping-pong test results (Fig. 6.2). For each pair of GPUs, we calculate the ratio of the Level 3 communication latency to the latency of the topology level associated with each pair of GPUs. To this end, we consider all latency values for message sizes in the range 1B to 2KB and store the resulting average in the topology matrix file. Note that channel latency is the dominating factor in the communication performance of the small messages.

For the bandwidth-based metric, we use the normalized GPU pair bandwidth values from the uni-directional bandwidth test results (Fig. 6.3). For each GPU pair, we first calculate the ratio of the bandwidth corresponding to the topology level of each GPU pair to the bandwidth of the Level 3 topology level. To this end, we only consider the message sizes for which their bandwidth values varies (2KB to 16MB). For each GPU pair, we store the average ratio in the physical topology matrix file. Note that the channel bandwidth is the dominating factor (and the channel latency is negligible) in the communication performance of the large messages.

Finally, for the distance-based metric, we use a set of APIs from the NVIDIA management library (NVML) [62] to extract the communication distance of different GPU pairs that are available on the node. In this regard, we mainly use the NVML topology APIs such as *nvmlDeviceGetTopologyCommonAncestor*() with which we retrieve the common ancestor for all GPU pairs. For each pair of GPUs, the depth of their common ancestor can represent the physical distance between them. Based on the maximum number of detected topology levels (4 in our case), for each GPU pair, we store the difference between this maximum value and the topology level value of the pair into the physical topology matrix file.

We have integrated our proposed topology-aware GPU selection scheme into the MPI initialization phase of the Open MPI library [68]. During this phase, we use the SCOTCH library to map the GPU virtual topology graph onto the physical topology graph. The output mapping table determines the desired GPU-to-process assignments in terms of an array M. The element $M[i]$ designates the GPU_id to be assigned to rank i. Due to the nature of the SCOTCH mapping algorithms, which can lead to different mapping results in different runs, only one process[2] performs the mapping

[2]Without loss of generality, process with rank 0

and scatters the results M to other processes. Upon receiving the GPU_id, each MPI process calls the CUDA device selection function $cudaSetDevice(GPU\_id)$ to select its assigned GPU.

6.3.2 GPU Assignment Scheme Across the GPU Cluster

Our GPU selection approach for a single node can be extended to across the cluster by considering all GPU units within a cluster. This way, processes can select any of the GPUs within the cluster. However, this approach has multiple obstacles that would make it less practical. First, by default the only GPUs that are visible to a process are those that reside on the same node to which the process is mapped. Consequently, accessing the GPUs on a remote node requires a middleware software framework (such as rCUDA [17]). Second, applications would require to move the data back and forth between the GPU global memory and the host CPU main memory; by selecting GPUs across the node, such interactions would have to go through the network communication channels that are generally more costly compared to the intranode communication channels. Third, in some GPU-aware MPI designs [85, 84], CPU-assisted mechanisms are used to perform the GPU inter-process communications; in such communications, data transfer is pipelined through the CPU host buffers; such designs would suffer from inter-process GPU assignment as pipelining would require costly internode data staging.

Taking these into consideration, we model our cluster-wide topology-aware GPU selection scheme as a joint problem of CPU and GPU mapping. In this regard, we

break down the mapping into three distinct phases: Phase1) internode process-to-node mapping; Phase2) intranode process-to-CPU-core binding; and Phase3) intranode GPU-to-process assignment.

Phase 1: Internode Process-to-Node Mapping

In this phase, we determine the cluster node that each process should be mapped to. This phase takes care of the internode CPU and GPU mapping. The communication pattern that is considered in this phase is the combination (summation) of the CPU and GPU communication patterns; the rationale behind this is that the internode CPU pairs and GPU pairs share the same communication channel; Fig. 6.2 also verifies and shows identical performance results for CPU and GPU inter-process communications across the nodes. Consequently, the aggregated CPU and GPU communications would represent the network activity with respect to both CPU and GPU inter-process communications. The physical topology in this phase is constructed based on the network topology of the cluster. The physical and virtual network topologies are then passed as the inputs to the SCOTCH mapping algorithm. The mapping result would lead processes with intensive network communications to potentially share the same node, thus minimizing the costly network activities.

Phase 2: Intranode Process-to-Core Binding

In this phase, each process that is now mapped to a node is bound to an individual core within that node. To perform the mapping, the required virtual and physical topologies have to be passed to the SCOTCH mapping algorithm. The virtual topology on each node is constructed based on the CPU communication pattern on that node; the physical topology, on the other hand, represents different intranode CPU communication channels.

Phase 3: Intranode GPU-to-Process Assignment

In this phase, processes that are mapped to a single node select a GPU device within that node. In this regard, we use our proposed node-wide GPU selection scheme from Section 6.3.1. Our scheme utilizes the intranode GPU communication pattern and intranode GPU topology that is constructed based on the bandwidth metric.

In all three phases, the SCOTCH [69] mapping library with the default strategy is used. Other mapping algorithms can also be used in different phases; these mapping algorithms do not have to be necessarily the same and can be tuned for each phase. Mirsadeghi, et al. [55] provide a through analysis of this 3-phase mapping framework with different mapping algorithms.

Implementation details

To extract the network topology level, we used the InfiniBand [32] subnet discovery command, `ibnetdiscover`. At the intranode level, the CPU and GPU topology levels are determined by the HWLOC (Hardware Locality) [8] and NVML [63] library, respectively. HWLOC [8] is a well-known library that provides a set of portable APIs to query the attributes of a node including cores, sockets, caches as well as I/O devices such as InfiniBand network interfaces [32] or GPUs. The NVML library includes a set of C-based APIs that can be used for extracting various information about the NVIDIA GPU devices, including the topology information. We use the *nvmlDeviceGetTopologyCommonAncestor*() API in the NVML library to retrieve the common ancestor of each pair of GPUs. The retrieved common ancestor represents the highest level of hierarchy that the communication path between two GPUs will pass through.

6.4 Experimental Results and Analysis

6.4.1 Experimental Platform

We conduct our experiments on four Helios nodes of System B as described in Chapter 3. For single-node experiments, we use 16 MPI processes with each MPI rank assigned to a single CPU core and a GPU device. In the cluster-wide experiments 64 processes are evenly distributed among the four nodes of the cluster. We also use Open MPI 1.10.2, CUDA 7.5, and SCOTCH 6.0 in our experiments. Our proposed designs in this chapter is not dependent to any MPI library. In this Chapter, we opt to use Open MPI library to compare against and implement our design. This library provides close to peak inter-process communication performance on various GPU communications channels, thus the importance of efficiently utilizing these communication channels can be better realized. In the rest of this section, we first provide our results and analysis on a single multi-GPU node and then discuss our cluster-wide evaluations.

6.4.2 Multi-GPU Node Results and Analysis

All our experiments in this section are running on one node of the Helios cluster (System B) with 16 MPI processes (one process per GPU) that are evenly distributed among the cores on the two sockets. We analyze our GPU selection scheme at both microbenchmark and application levels.

Microbenchmark studies

For our microbenchmark studies, we develop and use the following three microbenchmarks that model communication patterns commonly used in parallel applications:

1. 2D Stencil (*2D*),

2. 3D Stencil (*3D*),

3. Sub-communicator collective (*COLL*).

The 2D and the 3D microbenchmarks model a 2-dimensional 5-point and a 3-dimensional 7-point Stencil patterns, respectively. The processes are first organized into a logical 2-dimensional (3-dimensional) grid, and then each process communicates with its immediate neighbors along each of the dimensions of the grid. With 16 processes, the *2D* and *3D* microbenchmarks organize the processes into a 4×4 and a $2 \times 2 \times 4$ grid, respectively. In addition, we consider the options of having *wraparound* and *weighted* connections for these microbenchmarks. In this regard, the microbenchmarks with wraparound will include an extra link for connecting the processes at the two edges of each dimension together. Microbenchmarks without wraparound do not have any such links. On the other hand, in the weighted case, a higher weight is assigned to the communications along a specific dimension of the grid. In other words, each process sends and receives larger messages (3 times larger) to its neighbors that fall along the first dimension of the grid. In the non-weighted case, the same message volume is communicated along all dimensions. Thus, we will have the following four cases for the *2D* and *3D* microbenchmarks:

1. Without weight and without wraparound,

2. Without weight, but with wraparound,

3. With weight, but without wraparound,

4. With weight and with wraparound.

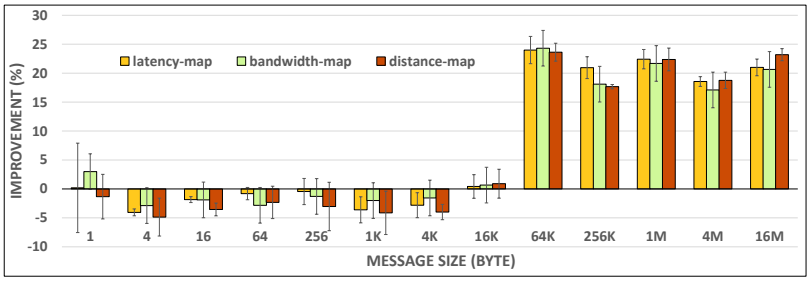
In the sub-communicator collective microbenchmark (*COLL*), the processes are first organized into a 3-dimensional grid ($4 \times 2 \times 2$), and an MPI sub-communicator is created for each group of processes that fall along the first dimension of the grid. Next, an MPI collective (i.e., MPI_Alltoall in our tests) is called over each sub-communicator.

Microbenchmark performance results and analysis

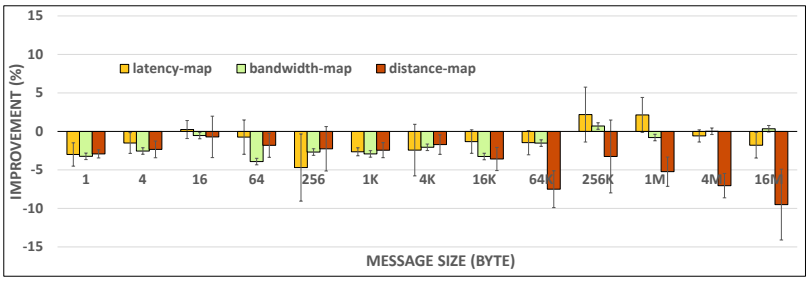
Fig. 6.4 through Fig. 6.6 show the results in terms of the improvements in the communication time of each microbenchmark. We report the improvements achieved by our topology-aware scheme over the default (naive) GPU selection. In the default scheme, each process selects a GPU based on its associated rank number; for instance, a common approach is for process with rank i to select the GPU with index i. Our experimental results in this section represent the average of four runs and also include standard deviation.

According to Fig. 6.4, for the non-weighted *2D* and *3D* microbenchmarks with no wraparound, performance improvement is achieved mainly for message sizes larger than 64KB. With wraparound connection, in the *2D* case (Fig. 6.4(b)), no performance improvement can be achieved for any of the metrics. We can also observe some performance degradation using the distance metric. For the *3D* case (Fig. 6.4(d)), all metrics provide the same amount of improvement across all message sizes.

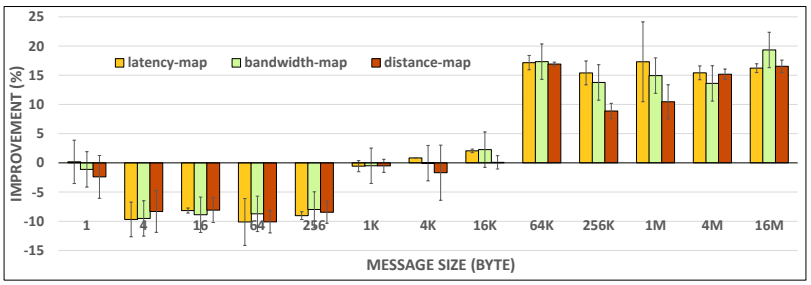
Fig. 6.5 shows that for the weighted *2D* and *3D* microbenchmarks, we can achieve up to 65% performance improvement by using the topology-aware GPU selection scheme. In addition, we can also see that the bandwidth metric consistently provides an equal or higher improvement compared to the latency and distance metrics. Fig. 6.6 shows that for the *COLL* microbenchmark, all three metrics can improve the



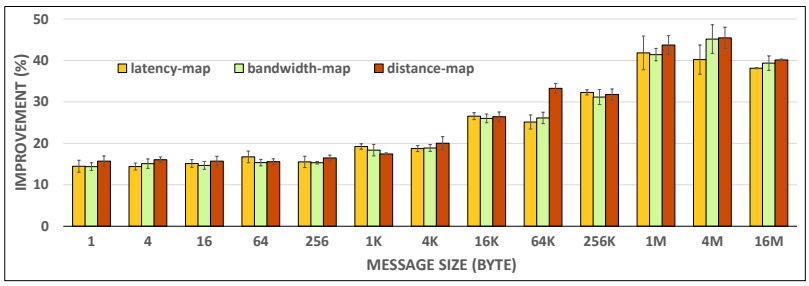
(a) 2D without wraparound without weight



(b) 2D with wraparound without weight

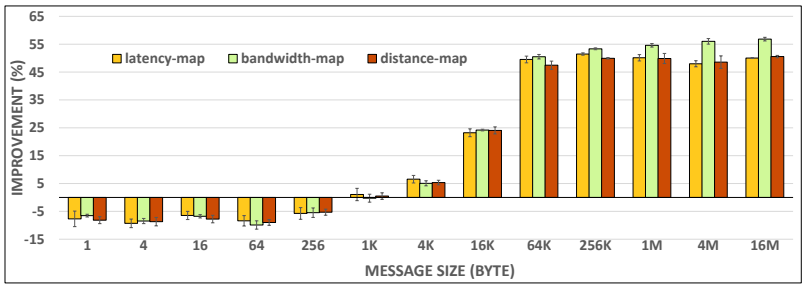


(c) 3D without wraparound without weight

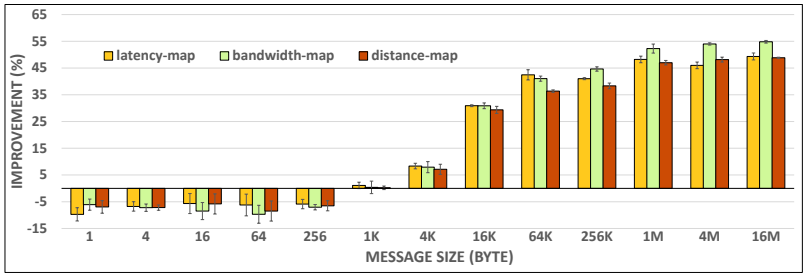


(d) 3D with wraparound without weight

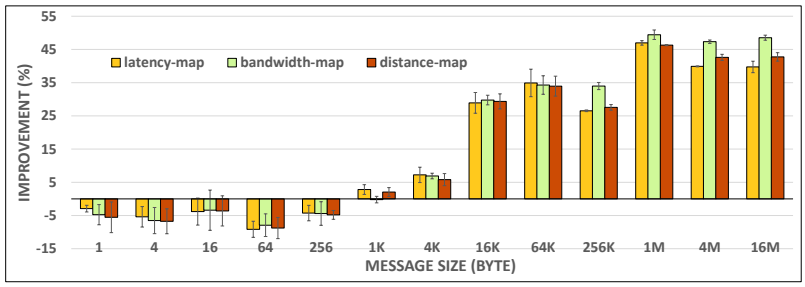
Figure 6.4: Communication time improvements achieved by topology-aware GPU selection over the default selection scheme for the non-weighted 2D and 3D microbenchmarks



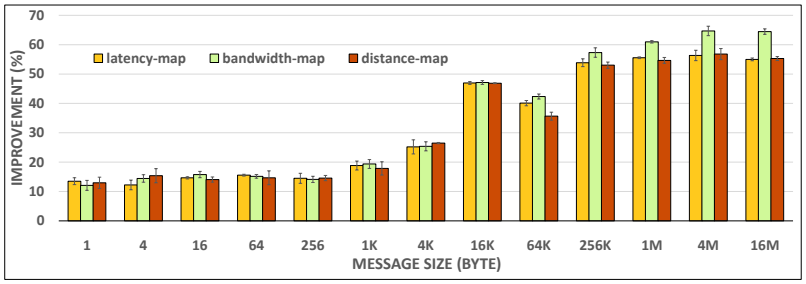
(a) 2D without wraparound with weight



(b) 2D with wraparound with weight



(c) 3D without wraparound with weight



(d) 3D with wraparound with weight

Figure 6.5: Communication time improvements achieved by topology-aware GPU selection over the default selection scheme for the weighted 2D and 3D microbenchmarks

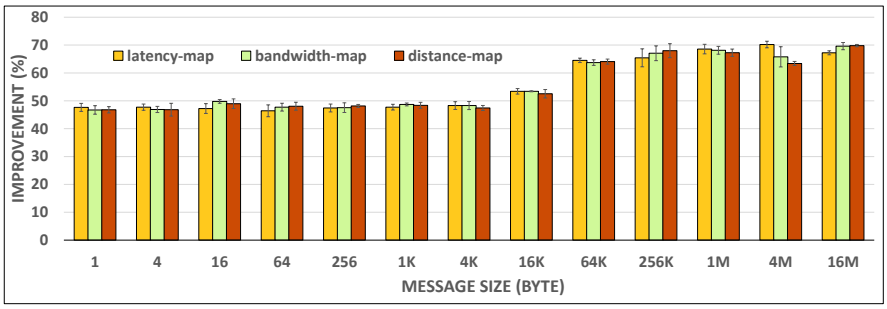


Figure 6.6: Communication time improvements achieved by topology-aware GPU selection over the default selection scheme for the sub-communicator collective microbenchmark, COLL

performance by up to 70% across all messages.

We also observe that the weighted *2D* and *3D* microbenchmarks can benefit more from our topology-aware GPU selection scheme when compared to their non-weighted counterparts. This is an expected behavior because in the weighted cases, the GPUs should be assigned to the processes in a way that the heavier-communicating processes end up using the GPU pairs with stronger physical connections. This would in turn provide more opportunity for performance optimizations through topology awareness.

On the contrary, we do not see the same behavior in presence of the wraparound connections. In fact, while adding wraparound connections would lead to further improvements for the *3D* microbenchmark, we do not observe any improvements for the *2D* microbenchmark; we can even see some performance degradation. To investigate this further, we analyzed the communication pattern of these microbenchmark with and without the wraparound connections. We noticed that wraparound connections in the *3D* microbenchmark will result in communications between processes that are far from each other (in terms of the GPUs assigned to them by the default approach). As shown in Fig. 6.2 and Fig. 6.3, the farther two GPUs are from each other, the lower their communication performance would be. In the *2D* case on the

other hand, adding wraparound connections will cause most of the communications to take place among the GPUs with stronger connections among them. In conclusion, adding wraparound connections makes the default GPU assignment an already-good match for the *2D* microbenchmark communication pattern, whereas it is the opposite for the *3D* microbenchmark.

In general, we can observe that our topology-aware schemes are more beneficial for large messages (greater than 16KB). The reason, as shown in Fig. 6.2 and Fig. 6.3, is that the difference in the latency of the small messages at different levels is not as much as the difference in the bandwidth of the large messages. More specifically, the latency ratio of **Level 1**, **Level 2**, and **Level 3** to **Level 0** for small messages is 1.01, 1.17, and 1.57, respectively. On the other hand, the bandwidth ratio of **Level 0** to **Level 1**, **Level 2**, and **Level 3** for large messages is 2.47, 4.22, and 4.47, respectively. So, the main difference between the channels at different levels is in their bandwidth. As a result, the performance of different topology levels are more diverse for large message sizes as they are mostly affected by the bandwidth characteristics of the underlying communication channels. For small messages, on the other hand, there is much less room for improvement (1.57 latency ratio compared to 4.47 bandwidth ratio). In addition, small-message communications are highly affected by the startup latencies which are not affected by the topology awareness and the specific strategy used for GPU assignment. Taking this into consideration, there are still three microbenchmarks for which we can get performance improvement for small messages. There are also some cases where we see slight performance degradation for small messages. We leave further investigations on this trend for small messages to a future work. Designing a scheme that is finely tuned for small-message communications falls

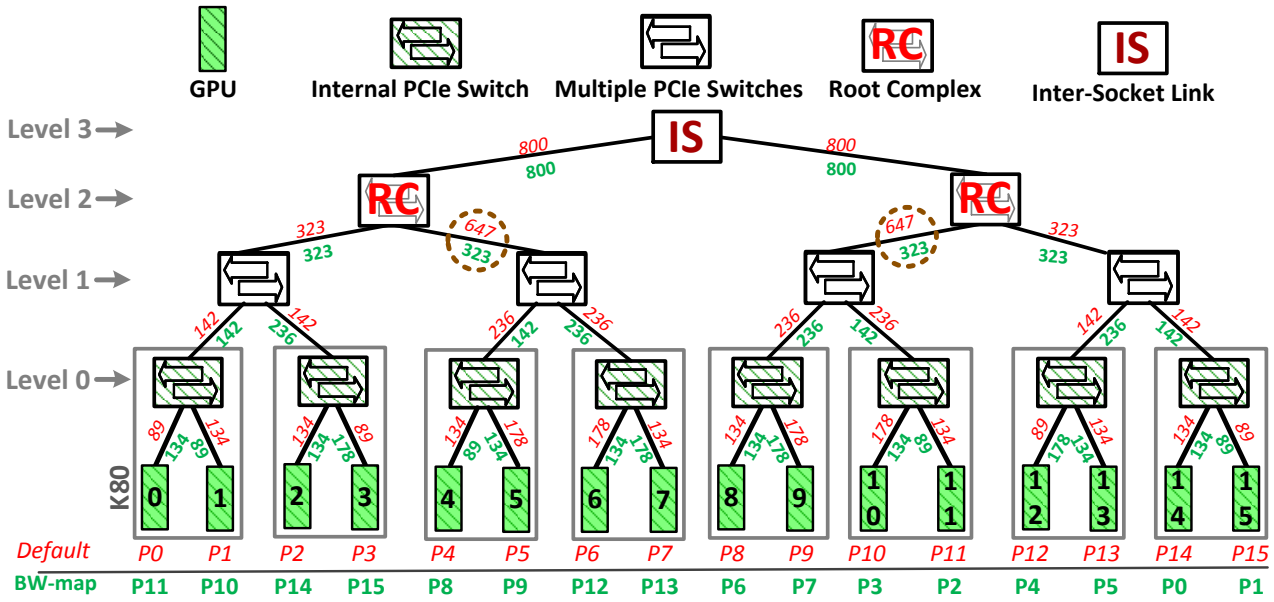
within the scope of our future work.

Congestion analysis

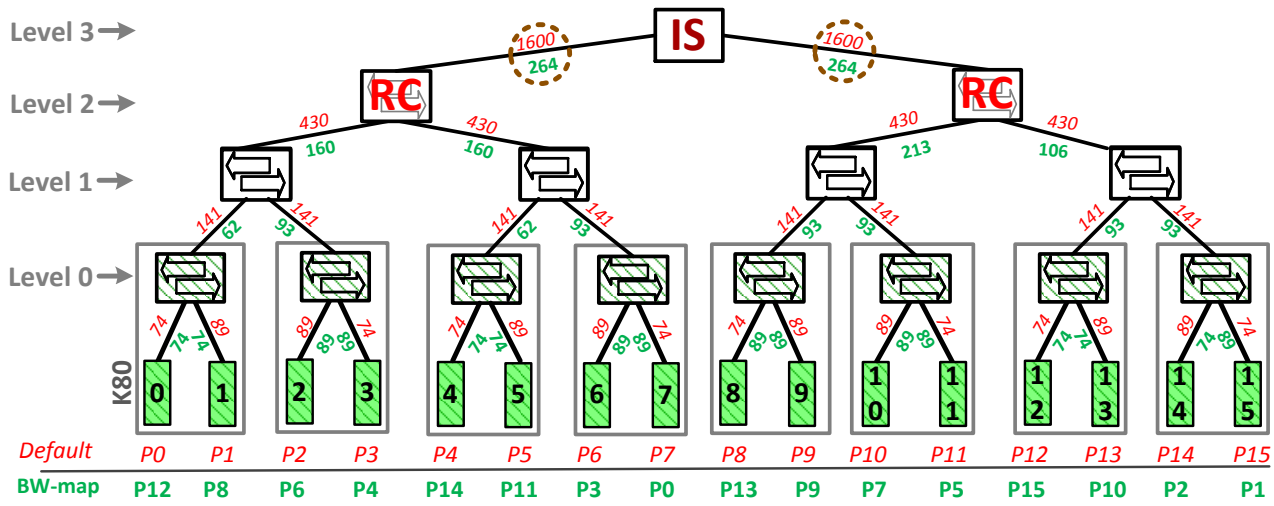
We further investigate our results by analyzing how our topology-aware scheme affects the congestion across the GPU communication channels. We define congestion as the total volume of the traffic that passes through a link divided by the bandwidth of that link. Different mappings will result in different congestion across the links that interconnect GPUs. We would like to find a mapping that leads to lower congestion across the communication channels.

In the following, we compare our topology-aware scheme against the default mapping with respect to their resulting congestion for two of our microbenchmarks: 2D non-weighted and 3D weighted (both without wraparound). Figure 6.7 shows the congestion values for the 2D and 3D microbenchmarks. The numbers above each link (shown in red) represent the congestion values for the default mapping, while the numbers below each link (shown in green) represent the congestion values for our topology-aware mapping using the bandwidth metric. Moreover, the numbers below each GPU denotes the rank of the process mapped onto that GPU by the default and topology-aware schemes, respectively.

As shown in Fig. 6.7(a), for the 2D non-weighted benchmark the congestion values for both mappings are similar to each other except at **Level 2** (encircled area), where our topology-aware mapping leads to a lower congestion. On the other hand, as shown by Fig. 6.7(b), for the 3D weighted benchmark the topology-aware mapping can significantly decrease the congestion values across the links at different levels of the tree. In particular, the maximum congestion at **Level 3** is decreased from 1600 to 264 (encircled area). This also correlates well with our microbenchmark performance



(a) 2D without wraparound without weight.



(b) 3D without wraparound with weight

Figure 6.7: Congestion values for the default and topology-aware GPU assignment

results for which we achieve higher communication time improvements for the 3D weighted benchmark (≈ 50% in Fig. 6.5(c)) compared to the 2D non-weighted (≈ 20% in Fig. 6.4(a)).

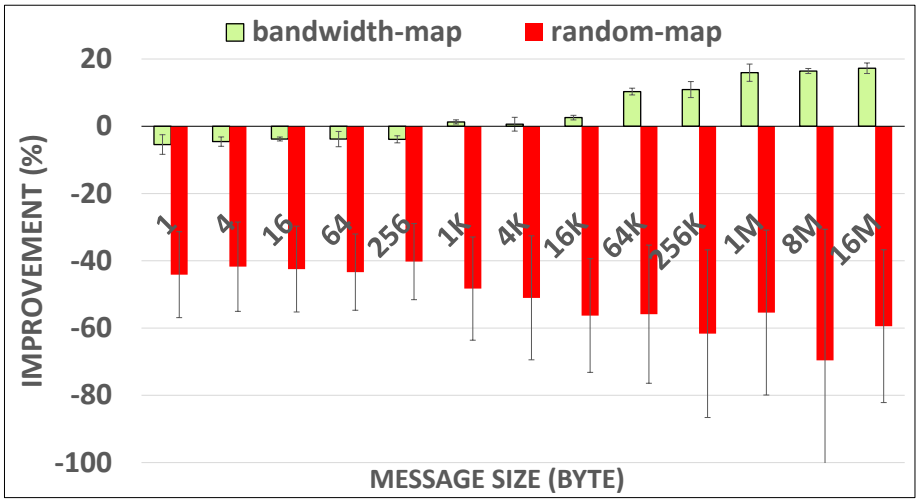
Comparison with Random Mapping

To further show the benefits of the topology-aware GPU selection, we compare our approach (using bandwidth metric) against four different random mappings. These mappings use a random sequence of numbers that is generated for each run. We change the random seed across different runs to avoid getting the same results. We provide our results as the average of four runs (with standard deviation) in Fig. 6.8. According to the figure, the random mapping for both benchmarks underperforms the default mapping. We can also see a higher standard deviation for the random results compared to our scheme. Moreover, we did not observe any performance improvement with any of the four random mappings over the default mapping. This highlights the importance of using a non-trivial and topology-aware design for GPU assignment.

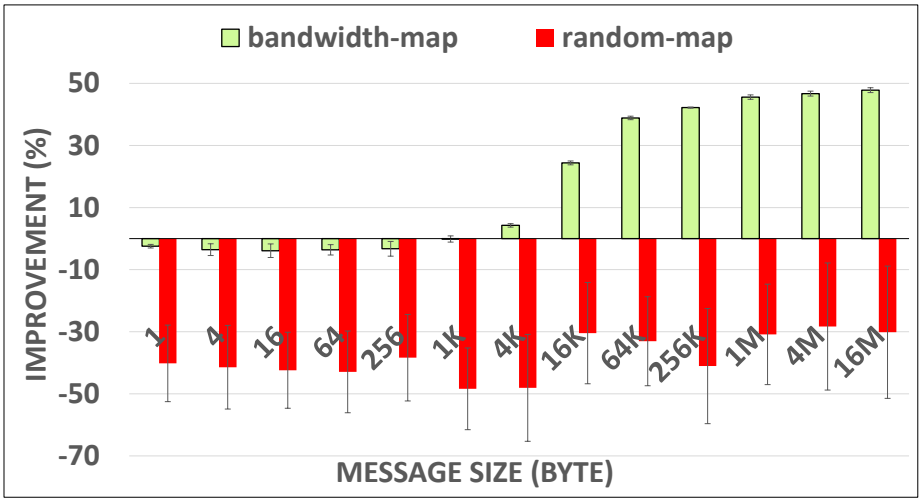
Application Study

In this section, we evaluate our proposed scheme across an end-point application: HOOMD-Blue [3, 27]. HOOMD-Blue (Highly Optimized Object-oriented Many-particle Dynamics-blue edition) is a general-purpose toolkit used for molecular dynamics simulations. It supports multi-CPU and multi-GPU simulations using MPI, with a one-to-one mapping between CPU cores (MPI ranks) and GPU devices. Each GPU is assigned a sub-domain of the multi-dimensional simulation box. The length of the sub-domains are calculated by dividing the lengths of the box by the number of processors per dimension.

We configure HOOMD-Blue for both single- and double-precision computations, and use two different benchmarks: 1) MicroSphere (MS), and 2) Lennard-Jones (LJ)



(a) 2D without wraparound without weight



(b) 3D without wraparound with weight

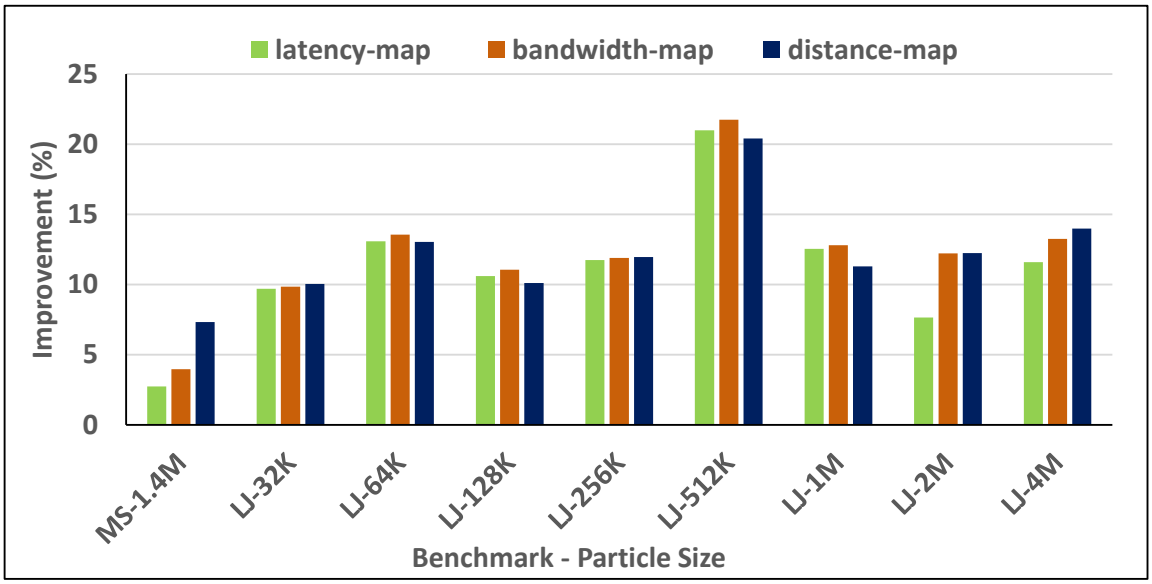
Figure 6.8: Communication time improvements achieved by topology-aware GPU selection and random mapping over the default selection scheme

liquid. The MS benchmark simulates 1,428,364 particles; it runs a system of star polymers in an explicit solvent which organize into a microspherical droplet. The LJ benchmark is a classic benchmark in general-purpose molecular dynamics simulations. It is representative of the performance that HOOMD-Blue can achieve with straight pair potential simulations. We use a range of particle sizes from 32,000 to 4,000,000 with this benchmark.

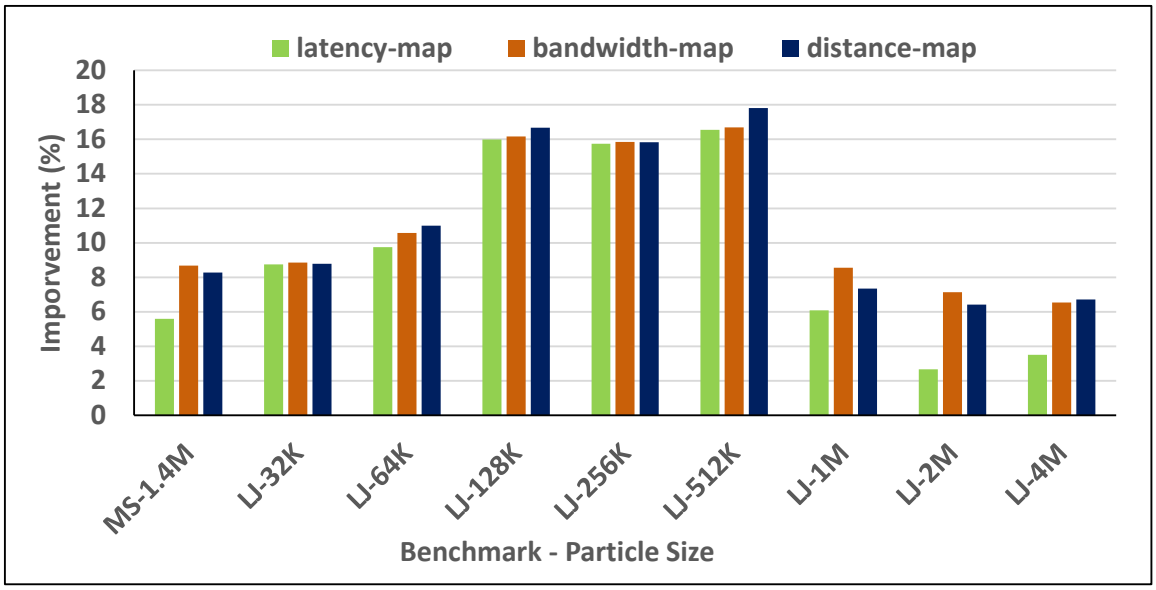
Application performance results and analysis

Fig. 6.9 reports the TPS (number of application Time Steps Per Second) improvements using our scheme over the default approach for the single- (Fig. 6.9(a)) and double-precision (Fig. 6.9(b)) versions of HOOMD-Blue. According to the figure, our scheme can provide up to 21.7% and 17.8% improvement for the single- and double-precision versions, respectively. We can also see that in most cases, the bandwidth and the distance metrics tend to outperform the latency metric.

The general trend in Fig. 6.9 shows that the improvement increases as the particle size grows up to 512K and then drops. In order to understand the reason behind such a drop in improvement, we need to evaluate the communication characteristics of the application in detail. To this end, we use the FPMPI [29] library to get more insights into the nature of the communications in HOOMD-Blue. FPMPI [29] is a profiling library which provides various information about the underlying MPI communications of an application. However, FPMPI does not distinguish between the CPU and GPU communications. In this regard, we have extended the FPMPI library to provide profiling support for both CPU and GPU communications. The extended profiler allows us to separately extract the CPU and GPU communication characteristics of an application. To this end, we leverage various CUDA APIs to analyze the buffer(s)



(a) Single-precision



(b) Double-precision

Figure 6.9: TPS improvement of topology-aware mappings over default mapping on a) single-precision and b) double-precision HOOMD-blue Application

in MPI routines. By analyzing the buffer(s), we can determine whether it is located on the host main memory or on the GPU global memory. We also instrument Open MPI to expose specific information that will be queried by the FPMPI library. For instance, we add the address type of the send/receive buffers of MPI routines to the MPI_Request object to distinguish among different types of communications (i.e., CPU versus GPU).

Fig. 6.10 demonstrates the GPU communication pattern of the double-precision HOOMD-Blue with LJ-512K benchmark. The pattern is represented by a square matrix, where each row and column corresponds to an MPI rank. The ijth element of the matrix denotes the total volume of messages transferred between the pair of GPU devices assigned to rank i and j in both directions[3]. As shown by Fig. 6.10, only a small portion (around 20%) of the GPU pairs are involved in heavy communications. More importantly, the communication pattern closely resembles the pattern induced by a 3D Stencil with wrap-around and non-weighted connections. On the other hand, Fig. 6.11 shows the distribution of GPU communications across various message sizes (averaged over all ranks) for three different particle sizes. We can observe that as the particle size increases, the share of the larger message sizes also increases. With such communication characteristics and in accordance to the result shown in Fig. 6.4.d, we expect to observe a non-decreasing improvement with the increase in particle size for HOOMD-blue. However, the highest improvement is achieved with the 512K particle size in Fig. 6.9.

To further investigate our application results, in Fig. 6.12 we provide the share of CPU and GPU communications in the total application runtime on LJ benchmark with different particle sizes. As shown in this figure, our mapping schemes can improve

[3]Note that the values have been normalized between 0 and 100.

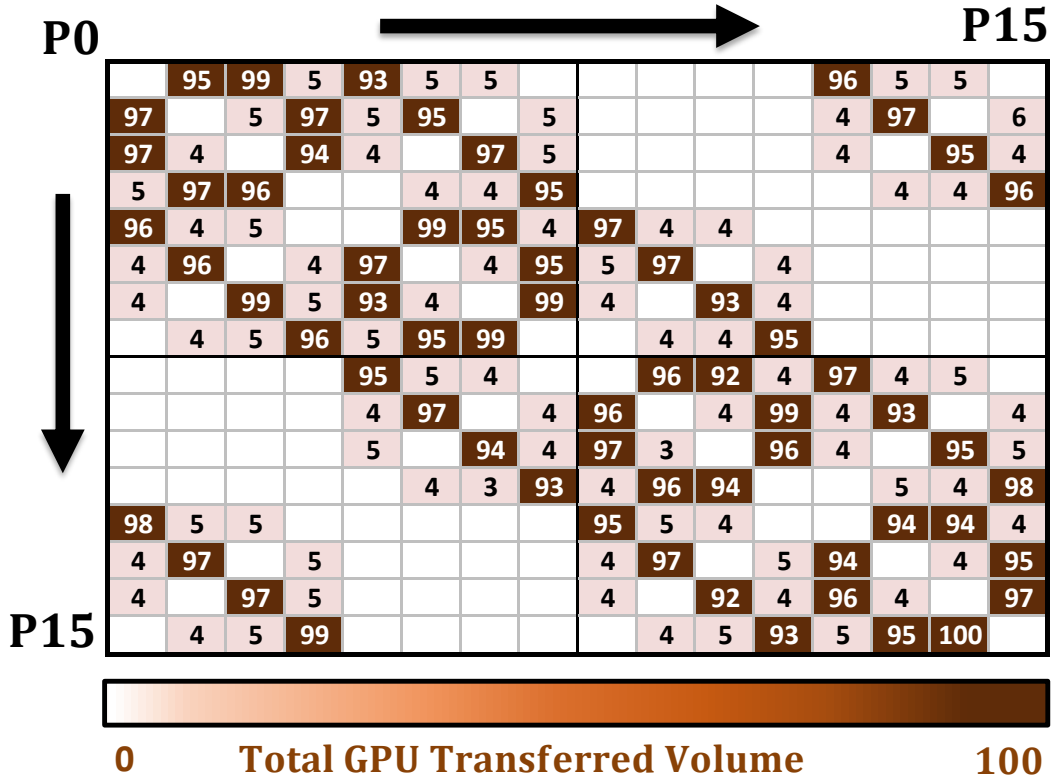


Figure 6.10: Normalized GPU communication pattern of double-precision HOOMD-Blue with LJ-512K benchmark

the total application runtime for all three benchmarks. We can also observe that our mapping schemes only improve the GPU communication portion of the application and the rest of the application runtime is almost left intact. Fig. 6.12 also shows that as the particle size increases, the share of GPU communications in the total application runtime decreases. Consequently, any GPU communication improvements would have a relatively lower impact on the total application performance. To verify this, we also measure the GPU communication time improvements achieved by our scheme for the HOOMD-Blue application. Fig. 6.13 shows the corresponding results. This time, we can see a steady improvement of about 20% for our topology-aware

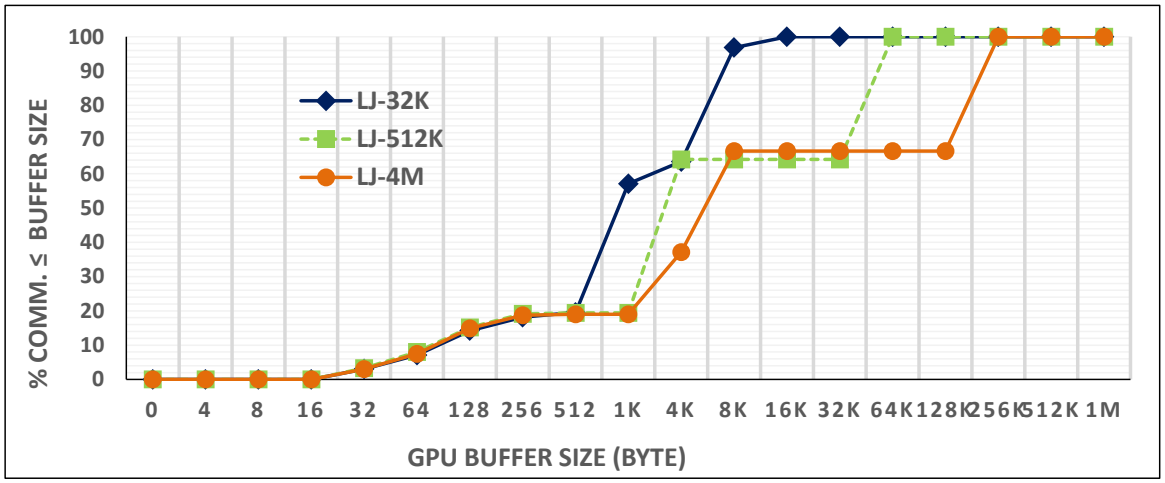


Figure 6.11: Distribution of different message sizes in GPU communications of double-precision HOOMD-Blue with LJ benchmark

scheme across all particle sizes. The only exception is with the latency metric and 4M particle size for which we see a reduction in the achieved improvement. This is expected to some extent, as latency is not a good representative of communication channel characteristics for large messages that this application mainly uses.

In conclusion, although larger message sizes provide more opportunity to improve communication performance thorough our topology-aware scheme, the lower share of GPU communications in the total runtime of the application leads to a lower reflection of such improvements. This is why we see a drop in the achieved performance improvements after the 512k particle size shown by Fig. 6.9. However, we see a generally increasing trend in the improvements up to (and including) 512K particle size. This implies that the 512K particle size is a turning point in terms of the impact of *GPU message size* versus *communication time share* on the overall improvements.

Mapping overhead

In this section, we analyze the overhead that is associated with our proposed

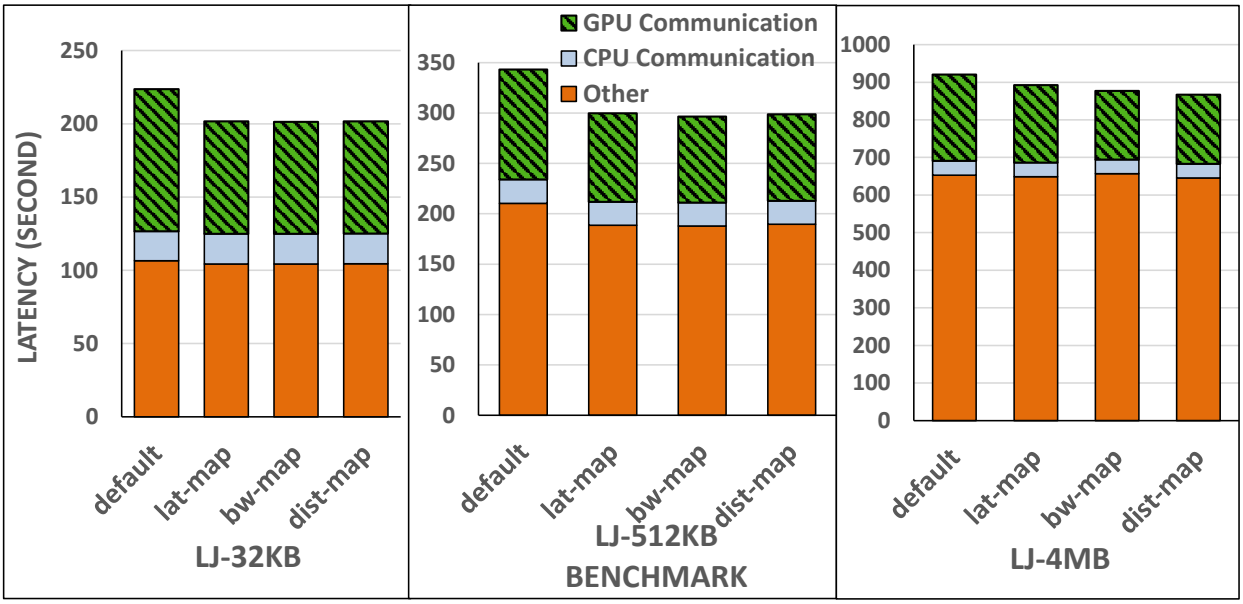


Figure 6.12: Share of GPU Communications in total HOOMD-Blue runtime

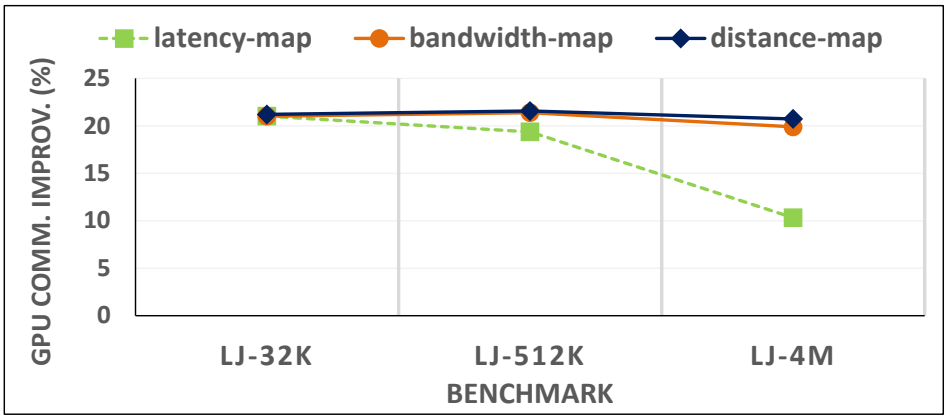


Figure 6.13: HOOMD-Blue GPU communication improvements

topology-aware GPU assignment scheme. To this end, we measure the time it takes to do the mapping and compare it against the application runtime. According to our results, regardless of the mapping metric, benchmark type, and the particle size that are used for the application, the mapping time is around 0.5 ms. This time is negligible compared to the total application runtime and contributes to at most

0.0003% of that. This mapping time is also considered to be a one-time overhead for each instance of the application. We note that the main purpose of our work in this chapter is to show the importance of topology-aware GPU selection and the mapping choice falls into the scope of future study. Thus, one may replace SCOTCH with another mapping heuristic. In particular, for small multi-GPU nodes (with 4 or 8 GPUs), one could use an exhaustive search to find the optimal mapping.

6.4.3 GPU Cluster Results and Analysis

In this section, we evaluate our cluster-wide mapping scheme over the default process mappings and GPU assignments. Our experiments are performed at both the microbenchmark and application levels and are averaged over four runs. Our experiments are performed on 4 nodes of the Helios cluster (System B in Chapter 3). We only consider and report results of our topology-aware scheme using the bandwidth metric in this section[4].

Microbenchmark Studies

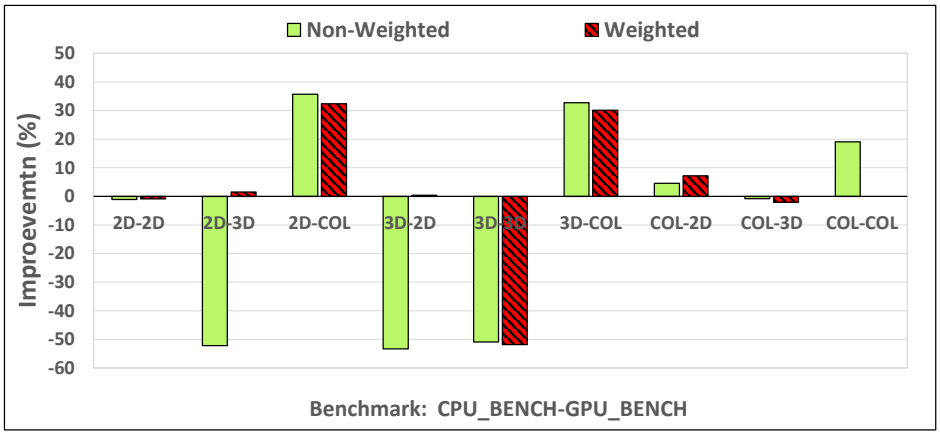
For our microbenchmark analysis, we have developed a microbenchmark suite, called *Accelerated MPI Benchmark* [21]. We use this benchmark suite to evaluate the performance of both CPU and GPU MPI communications. The *Accelerated MPI Benchmark* models various communication patterns among the CPU cores as well as among the GPU devices of a cluster. To better resemble the communication patterns of the real world applications, this benchmark is also capable of simultaneously modeling various CPU and/or GPU communication patterns. The current version of this suite

[4]Our preliminary experimental evaluations on other metrics showed identical or worse performance results

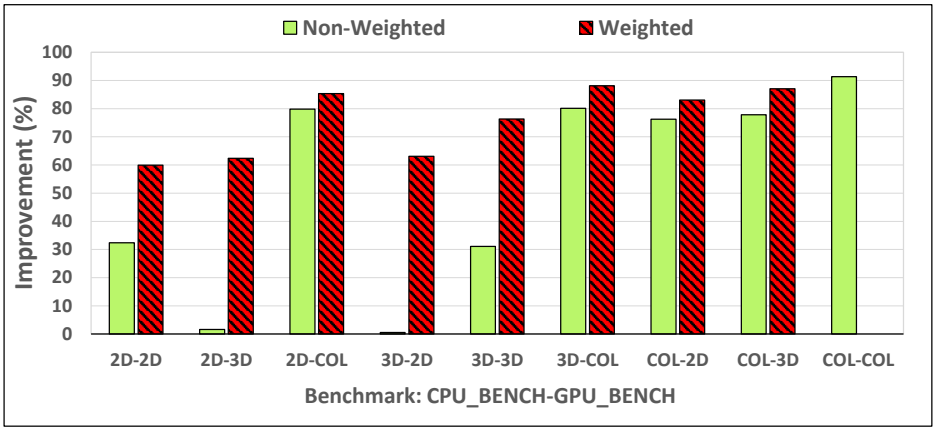
consists of three microbenchmarks: 1) 2D Stencil (2D), 2) 3D Stencil (3D), and 3) Sub-communicator collective (COL). The 2D and the 3D microbenchmarks model a 2-dimensional 5-point and a 3-dimensional 7-point Stencil patterns, respectively. The processes are organized into a 2D/3D mesh, and each process communicates with its two immediate neighbors along each dimension. For these two 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 (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. Table 6.1 summarizes the specifications of our developed microbenchmark.

Each of the microbenchmarks can be independently used as the communication pattern among CPU cores and among GPU devices. In this section, 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.

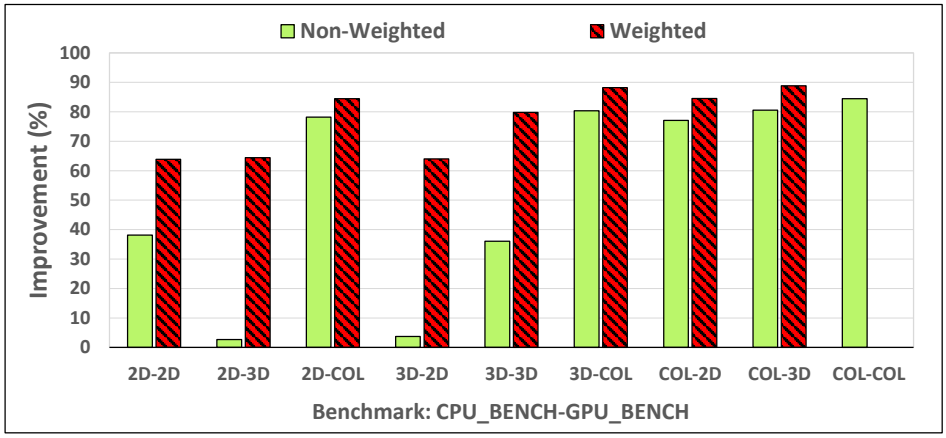
Fig. 6.14 shows the improvements achieved by using our proposed 3-phase mapping framework over the default mapping. We report performance improvement on



(a) Short message size: 256 Byte



(b) Medium message size: 64 KB



(c) Large message size: 1 MB

Figure 6.14: Microbenchmark runtime improvements using a 3-phase mapping framework on various message sizes over the default selection scheme

Name / Arguments	2D	3D	Col
Benchmark Description	2D Stencil	3D Stencil	Alltoall on a sub-comm
Message Weight	1 or 3	1 or 3	N/A
Wraparound	W or W/O	W or W/O	N/A
Dimensions on 64P	8x8	4x4x4	4x4x4

Table 6.1: Microbenchmark specification

three different message sizes on all nine combinations of our microbenchmarks. According to the figure, the improvement increases with the message size. Fig. 6.14(a) shows the result on short message sizes (256 Byte), where in most cases our topology-aware scheme does not provide any performance improvement and there are also cases with performance degradation. This is an expected behavior as we do not consider latency-based optimizations and metrics to determine the mapping algorithm. For medium and large message sizes, as shown in Fig. 6.14(b) and 6.14(c), we can observe consistent performance improvement (up to 91.4%) for all microbenchmarks. Fig. 6.14 also provides result for both of the weighted and non-wighted benchmark cases. Improvements for weighted microbenchmarks are generally higher than the non-weighted versions. This is because in the weighted microbenchmarks, larger message sizes communicate along one of the grid dimensions, providing more room for optimizations. While the default mapping and GPU assignment is oblivious to the communication volume and bandwidth among different processes, our design takes advantage of such information and improves performance by mapping intense communications on higher-bandwidth channels.

Application Level Analysis

In this section, we evaluate our mapping scheme on HOOMD-blue and consider two cases of single- and double-precision formats of the application. For the input,

we use the classic Lennard-Jones (LJ) liquid and MicroSphere (MS) benchmarks. In our experiments, we vary the particle size of LJ from 64 thousand to 4 million, and use the default 1.4 million particle size in MS.

Fig. 6.15 shows the TPS (number of application time steps per second) improvements achieved from our topology-awere scheme on various benchmarks. We can see up to 8.3% and 7.1% improvements on single- and double-precision version of the application, respectively. Moreover, for both clusters, the highest improvements are achieved with 512K particle size. We can make two general observations from the figure. First, while we can observe performance improvement in almost all cases, the extent of the improvement is lower compared to our single-node results in Fig. 6.9. The main reason behind this is that the number of processes in our cluster-wide experiment has quadrupled compared to our single-node experiments. This effectively lowers the size of the GPU messages leading to less improvement opportunity. Consequently, the expected trend should be higher improvement as the particle size increases. However, this trend cannot be observed in the figure. In order to further investigate this odd trend we profiled the application and investigated the results.

According to our profiling results, as the particle size increases, the total computation load of the application also increases. Our profiling results also show that by increasing the particle size the majority of the communicated messages still fall below 32KB on our platforms. Therefore, the messages are not large enough to consistently make the application bandwidth-bound. Moreover, 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 improvement opportunity through our topology-aware scheme. We expect to see higher improvements for applications that use larger messages and/or employ irregular communication patterns.

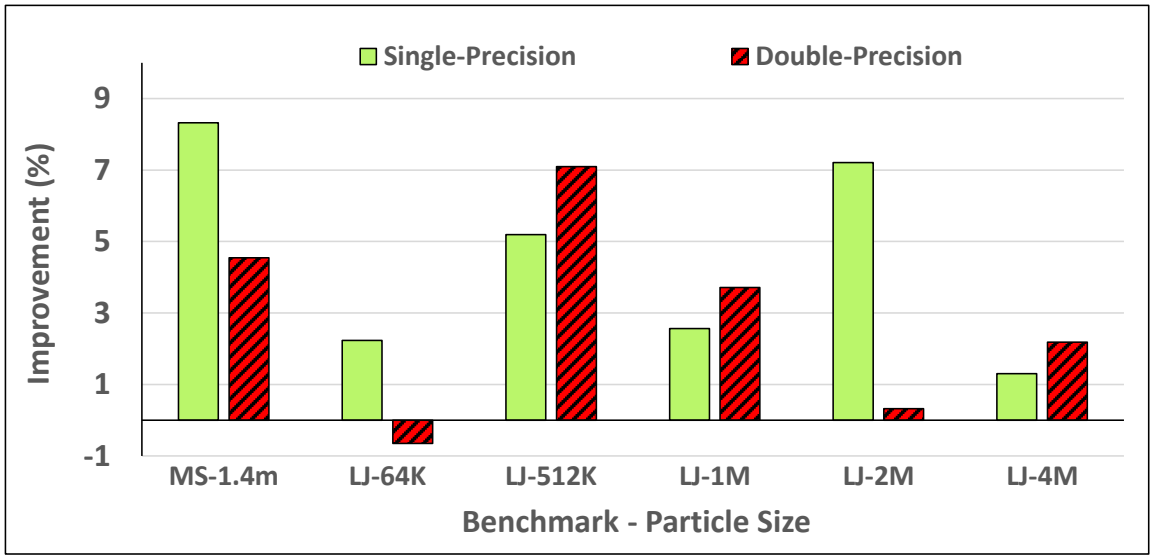


Figure 6.15: HOOMD-blue TPS (number of application time steps per second) improvements using a topology-aware scheme on various benchmarks

6.5 Provision of Using our Proposals with Future GPU Accelerators and Clusters

Future GPU clusters will be equipped with the next generation GPUs and will potentially use more advanced node architectures. These GPU clusters, however, should continue to benefit from our GPU selection schemes. On one hand, our schemes are oblivious to the GPU computational capability, and mainly target the efficiency of communications among them. On the other hand, our GPU selection proposals can

adapt to any node architecture and GPU clusters that consist of different communication channels. It does so by extracting the physical characteristics for each node and the cluster. Having said that, the more diverse the performance of these communication channels, the more opportunity for our schemes to improve the communication efficiency.

Fig 6.16 (repeated here from Fig. 5.11 for convenience) shows an example node architecture that is equipped with the latest GPU generation (i.e. Pascal P100) and the latest NVIDIA Nvlink intranode interconnect, as discussed in Chapter 5 as well. Table 6.2 lists the uni-directional communication bandwidth of all of the GPU pairs. According to the table, two of the GPU pairs provide 40 GB/s communication bandwidth, while the rest of the four pairs provide 20 GB/s. Such a node architecture would provide the opportunity for our proposed GPU selection scheme with the bandwidth metric to map more intensive communications on the stronger communication channels (i.e., channels with 40 GB/s bandwidth) and thus improving the communication efficiency.

GPU Pair	Uni-Directional Bandwidth
GPU0-GPU1	*40 GB/s*
GPU0-GPU2	*20 GB/s*
GPU0-GPU3	*20 GB/s*
GPU1-GPU2	*20 GB/s*
GPU1-GPU3	*20 GB/s*
GPU2-GPU3	*40 GB/s*

Table 6.2: Uni-directional bandwidth of different GPU pairs in a 4-GPU node with Pascal P100 and NVLink interconnect

In summary, our proposed GPU selection schemes will remain independent of future GPU types and node architectures. Moreover, as long as performance diversity

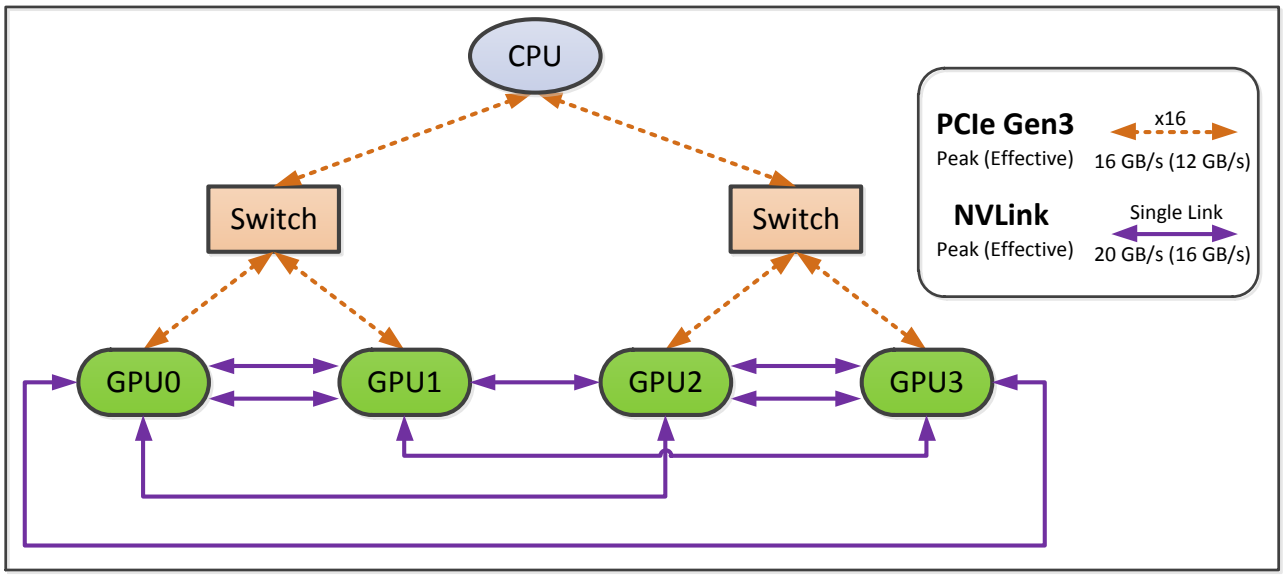


Figure 6.16: Node architecture of a 4-GPU node with Pascal GPUs and NVLink interconnect (adapted from [66])

among different communication channels exists, our schemes will continue to benefit the communication performance.

6.6 Summary

In this chapter, we showed that intranode GPU topology can have significant impact on the communication performance. We showed that in heterogeneous clusters with GPU accelerators, heterogeneity not only does exist in terms of the computational units, but also in terms of the communication channels that interconnect them. To this aim, we proposed topology-aware mapping solutions for both single and clustered multi-GPU nodes.

For single multi-GPU nodes, we proposed a non-trivial topology-aware GPU selection scheme that considers the application communication pattern and the physical

topology of the node. We modeled the problem as a graph mapping problem and used the SCOTCH library to solve it. We also used three metrics to represent the physical topology of the multi-GPU nodes: 1) latency; 2) bandwidth; and 3) communication distance. Our experimental results show that our proposed scheme can highly improve the communication performance at both the microbenchmark and application levels. We also observed that the microbenchmarks with more distant communications between the GPUs (with respect to their GPU ID) are more susceptible to improvement by our topology-aware scheme. Moreover, higher improvement is achieved with the weighted microbenchmarks compared to the non-weighted ones. On the application front, we observed more improvements for cases with larger messages and higher share of GPU communications in the total runtime. In general, our topology-aware scheme shows to efficiently utilize all three metrics and improve the communication performance. However, we observed the highest improvement with the bandwidth metric and with large message sizes.

We also extended our topology-aware GPU selection scheme from a multi-GPU node to across the GPU cluster. In this regard, we defined a cluster-wide topology-aware GPU selection as a joint problem of process-to-CPU-core mapping and GPU-to-process assignment. To address this problem, we proposed a mapping scheme which breaks down the mapping into three distinct phases: 1) internode process-to-node mapping; 2) intranode process-to-CPU-core binding; and 3) intranode process-to-GPU assignment. We evaluated our mapping scheme at both microbenchmark and application levels. For microbenchmark analysis, we developed a microbenchmark suite capable of modeling different communication patterns among the GPU buffers and/or the CPU buffers. Our microbenchmark and application results indicate that

by efficient process to core binding and GPU assignment, we can achieve performance improvement over the naive mapping schemes on GPU clusters.

Chapter 7

Conclusions and Future work

Today, many of the modern HPC clusters are equipped with GPU accelerators due to their high computational power and low energy consumption. Accordingly, many of the HPC applications are re-written or adjusted to exploit the massive computational power of GPUs. Performing compute-intensive portions of the application on GPUs seems to be a well justified approach. However, this has to be bundled with efficient GPU communications so the actual benefit of offloading can be realized. Optimizing GPU communication should not remain as an afterthought, and has to be efficiently addressed. While, some communication libraries provide support for GPUs, there are many communication routines within them that do not efficiently utilize GPU-aware designs and hardware features. For instance, many of the communication routines do not exploit GPU communication and computation features in conjunction with efficient algorithms to amortize their high overhead in application runtime. In addition, with the presence of different data copy mechanisms and communication channels for GPU inter-process communications, efficient usage of these resources is of paramount importance. The absence of hierarchical designs for GPU clusters is another one of important feature missing in GPU communication routines. While

GPU accelerators have added heterogeneity in terms of the computational units, they have also added heterogeneity in terms of the communication channels that are interconnecting them; this is another important factor that has been overlooked in optimizing GPU inter-process communications.

In this dissertation, we are focused on improving the GPU inter-process communication performance by utilizing innovative designs, efficient algorithms, topology-aware designs, and advanced hardware features. The main contributions of this dissertation are as follows:

(1) Efficient GPU Collective Communication Algorithms

In Chapter 3, we proposed two GPU-aware algorithms for collective communications: 1) GPU Shared Buffer-aware (GSB); and 2) Binomial Tree Based (BTB). In both designs, we used GPU-specific capabilities to accelerate communication and computation. As a test case scenario, these designs were applied to MPI_Allreduce. The designs in this chapter target clusters of single-GPU nodes. In Chapter 4, on the other hand, we provide how hierarchical designs can be applied to collective operations targeting clusters of multi-GPU nodes. We also evaluate the efficiency of different algorithms within different hierarchy levels of our designs. Our proposed collective designs provided up to 22 and 5 times performance improvement over the existing designs within a single-GPU node and across the cluster of single-GPU nodes, respectively.

(2) Hierarchical Framework for GPU Collective Communications

In HPC clusters with multi-GPU nodes, GPU inter-process communications can take place at different hierarchy levels. For instance, communications can take place

within a single GPU, across intranode GPUs, or over the network. However, previous research are either oblivious to this hierarchical structure or provide limited hierarchy-aware support through their transport layer. In this regard, we proposed a hierarchical framework for MPI collective operations on the GPU. Using our framework, we break down the collective operation into different hierarchical steps. We evaluated the effectiveness of our proposed framework by analyzing different algorithms in each hierarchy level, and showing the importance of choosing the right one. Our proposed framework is evaluated using MPI_Allreduce. Our evaluation showed promising results for large message sizes which are highly in-use in deep learning and big data applications. Our proposals provided up to 80% and 65% performance improvement on MPI_Allreduce over the existing flat designs within a multi-GPU node and across the cluster of multi-GPU nodes, respectively. However, for short and medium message sizes, similar to Chapter 3, our designs showed no improvement. This is mainly due to the high startup and synchronization costs that are associated with the data copy mechanism (i.e., CUDA IPC) that we used for GPU inter-process communications. In this regard, we investigated how to intelligently select the data copy mechanisms to efficiently perform GPU inter-process communications in Chapter 5.

(3) Efficient GPU Communications through Smart Data Copy Mechanism Selection

Inter-process communications among GPU buffers can be performed through different communication channels and data copy mechanisms. In Chapter 5, we provided a comprehensive analysis of two of these data copy mechanisms (i.e., CUDA IPC and Host-Staged). Our evaluation showed the benefit of jointly using different data copy

mechanisms to perform multiple inter-process communications. The benefit is mainly rooted in overlapping different data copy mechanisms that traverse different communication channels. Taking this observation into account, we proposed two algorithms for GPU collective operations that are capable of efficiently managing their copy mechanisms: 1) *Static* Hyper-Q aware; and 2) *Dynamic* Hyper-Q aware.

The *Static* algorithm decides what data copy mechanisms to use based on a priori information that it extracts from a tuning table. The *Dynamic* algorithm, on the other hand, dynamically decides the number and mechanism of the copies at runtime. These designs were evaluated on MPI_Allgather and MPI_Allreduce. The experimental results showed that the proposed designs outperform the native design across most of the message sizes. In general, the *Static* approach showed to provide higher improvement compared to the *Dynamic* approach. However, the *Dynamic* approach in most cases provide competitive results, and is also independent of any tuning parameter, thus having the portability advantage. We also evaluated the effect of the NVIDIA MPS service on the *Static* and *Dynamic* approach. The MPS service allowed different data copy mechanisms in our design to further overlap with each other and more efficiently share GPU resources. Our proposed designs showed up to 2.62 times speedup in the total GPU inter-process communications.

Our proposals in Chapter 3, 4, and 5 were targeted to improve the performance of GPU communications through efficient algorithms, novel designs, and modern features. In Chapter 6, on the other hand, we target to improve the communication and applications performance on multi-GPU nodes through topology-awareness.

(4) Topology-aware GPU Communications

In Chapter 6, we first provided a comprehensive analysis of different traversal

paths that interconnect different GPUs as well as different CPUs in a GPU cluster with multi-GPU nodes. The performance results showed that, depending on the physical topology level that interconnects these processing units, their performance can highly vary from each other. Taking this into consideration, we proposed topology-aware mapping solutions for both single and clustered multi-GPU nodes. For multi-GPU nodes, we used a non-trivial topology-aware GPU selection scheme that considers the application communication pattern and the physical topology of the node. We used three metrics for the GPU selection/mapping purpose: 1) latency; 2) bandwidth; and 3) communication distance. The bandwidth, among these metrics showed to provide the highest improvement in our experiments.

We also discussed the extension of our topology-aware GPU selection scheme to across the GPU cluster. We defined a cluster-wide topology-aware assignment as a joint problem of process-to-CPU-core mapping and GPU-to-process assignment. To address this problem, we proposed a mapping scheme that breaks down the mapping into three distinct phases: 1) process-to-node mapping; 2) intranode process-to-CPU-core binding; and 3) intranode GPU-to-process assignment.

Our comprehensive evaluation included both microbenchmark and application levels. We developed a microbenchmark suite that is capable of modeling different communication patterns among the GPUs and CPUs. Using our proposed topology-aware solutions, we observed considerable performance improvement at both the microbenchmark and application levels. More specifically, on a multi-GPU node, our topology-aware proposal provided up to 72% and 21% improvement in performance at the microbenchmark and application levels, respectively. Our proposals also improved the total benchmark runtime by 90% and showed up to 8% performance improvement

across the GPU cluster.

7.1 Future Work

Our future research plans in general revolve around developing designs that can tackle the major communication challenges in the HPC clusters with accelerators. In the following, we will outline some of the opportunities to extend the proposals discussed in this dissertation.

In Chapter 3, we proposed GPU-aware algorithms for MPI_Allreduce operation. We utilized CUDA reduction kernels and GPU communication features within the proposed designs to further accelerate this operation. An interesting avenue for future research would be to extend these designs to other collective operations, such as MPI_Alltoall. A CUDA transpose kernel or CUDA two-dimensional memory copies can be potentially applied in our designs to accelerate this operation.

In Chapter 4, we proposed a hierarchical framework for collective operations targeting clusters with multi-GPU nodes. We would like to extend this framework by studying more algorithms within different hierarchy levels. More specifically, we are interested to further tune our collective algorithms for the underlying hardware, for instance by utilizing the PCIe full duplex capability, minimizing slow inter-socket communications, and avoiding throttling the HCA in network transfers. Moreover, we would like to investigate hierarchical collectives that are tuned for certain message sizes. While our proposal in this chapter showed to highly improve the microbenchmark performance for large message sizes, we would also like to evaluate it using real-world deep learning and HPC applications.

In Chapter 5, we proposed designs that jointly use the CUDA IPC and Host-Staged data copy mechanisms for GPU collective operations. We also showed that the NVIDIA MPS service can be used to speedup the proposed designs. However, the CUDA IPC and MPS service showed some inconsistencies with each other that prevented us from extending our designs to multi-GPU nodes. While this inconsistency may be resolved with the next generation of this service, we would like to investigate alternative designs to avoid it in the first place.

We are interested to evaluate our topology-aware schemes in Chapter 6 with other GPU applications. More specifically, we would like to study applications with different communication patterns and study their behavior. Currently, we manually decide the metric (i.e., latency, bandwidth, and distance) in our topology-aware mapping schemes. An interesting avenue for the future work is to automate this process, allowing appropriate metric to be selected based on the profiled application characteristics. In general, our topology-aware schemes showed higher improvements for applications that use larger messages. For applications with small messages, we intend to study the latency characteristics of the physical topology and utilize latency-based mapping heuristics to perform efficient GPU assignment.

NVIDIA has recently released the NVIDIA Collective Communications Library (NCCL) [60] to facilitate the development effort for multi-GPU applications. This library mimics the MPI collective operations and has a familiar interface for HPC developers. The NCCL library implements ring-style collectives that are optimized for throughput. Designing other collective algorithms in this library that can minimize the slow inter-socket communications and exploit the PCIe full-duplex capability can be an interesting future work.

While heterogeneous clusters provide resources with massive computational power, efficiently utilizing them is of paramount importance. Inefficient utilization of these resources can lead to performance per watt degradation. Our investigation on various HPC applications shows that many of them highly underutilize the memory and computational GPU resources. In this regard, as a future work we would like to investigate designs that can improve resource utilization and consequently increase the performance per watt of the application. A potential solution would be to offload more work on the underutilized GPUs by allowing multiple processes to share their resources. For instance, the number of processes and GPU devices can be decided from a tuning table and prior to the application runtime. Another interesting alternative solution is to propose designs that can intelligently select a set of processes to share different underutilized GPUs. One way to tackle this problem is to use the Minimum Consistent Subset Cover algorithms to select the minimum subset of processes that would lead to highest GPU utilization by sharing the GPU resources. The expected outcome of this work would be improved performance per watt of multi-process applications running on multi-GPU nodes.

Improving communication in HPC clusters using the underlying hardware or software features is another topic of interest for our future work. With the latest GPU capabilities introduced in Pascal and Volta architectures, new doors for improvement are opened for our future research. For instance, leveraging the fast NVLink interconnect in conjunction with the Unified Virtual Memory (UVM) technology should fit well in Remote Memory Access operations. GPU kernel designs that are used for datatype processing should be revisited, specifically with the emergence of the new fast High-Bandwith Memory 2 (HBM2) technology.

Bibliography

[1] Advance Micro Device - AMD. `http://www.amd.com/`. [Online; last accessed 07/19/2017].

[2] D. Amodei, S. Ananthanarayanan, R. Anubhai, J. Bai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, Q. Cheng, G. Chen, et al. Deep speech 2: End-to-end speech recognition in English and Mandarin. In *International Conference on Machine Learning*, pages 173–182, 2016.

[3] 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.

[4] 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.

[5] M. Beck and M. Kagan. Performance evaluation of the RDMA over ethernet (RoCE) standard in enterprise data centers infrastructure. In *Proceedings of the 3rd Workshop on Data Center-Converged and Virtual Ethernet Switching*, pages 9–15. International Teletraffic Congress, 2011.

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

[7] M. S. Birrittella, M. Debbage, R. Huggahalli, J. Kunz, T. Lovett, T. Rimmer, K. D. Underwood, and R. C. Zak. Intel® Omni-path architecture: Enabling scalable, high performance fabrics. In *High-Performance Interconnects (HOTI), 2015 IEEE 23rd Annual Symposium on*, pages 1–9, 2015.

[8] 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. 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 180–186, 2010.

[9] W. M. Brown, J.-M. Y. Carrillo, N. Gavhane, F. M. Thakkar, and S. J. Plimpton. Optimizing legacy molecular dynamics software with directive-based offload. *Computer Physics Communications*, 195:95–101, 2015.

[10] J. Bruck, C. Ho, S. Kipnis, E. Upfal, and D. Weathersby. Efficient algorithms for all-to-all communications in multiport message-passing systems. *IEEE Transactions on parallel and distributed systems*, 8(11):1143–1156, 1997.

[11] D. Bureddy, H. Wang, A. Venkatesh, S. Potluri, and D. K. Panda. OMB-GPU: A Micro-benchmark suite for evaluating MPI libraries on GPU clusters. In *European MPI users' group meeting (EuroMPI)*, pages 110–120, 2012.

[12] C. Chu, K. Hamidouche, A. Venkatesh, A. Awan, D. Panda . CUDA kernel based collective reduction operations on large-scale GPU clusters. In *Cluster, Cloud and Grid Computing (CCGrid), 16th IEEE/ACM International Symposium on*, pages 726–735, 2016.

[13] Z. Chen, J. Xu, J. Tang, K. Kwiat, C. Kamhoua, and C. Wang. GPU-accelerated High-throughput Online Stream Data Processing. *IEEE Transactions on Big Data*, 2016.

[14] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew. Deep learning with COTS HPC systems. In *International Conference on Machine Learning*, pages 1337–1345, 2013.

[15] CORE-Direct The Most Advanced Technology for MPI/SHMEM Collectives Offloads. `http://www.mellanox.com/related-docs/whitepapers/TB_CORE-Direct.pdf`. [Online; last accessed 07/19/2017].

[16] DeppBench, benchmark tool for measuring basic operations involved in training deep neural network, https://svail.github.io/DeepBench/. [Online; last accessed 03/03/2017].

[17] 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. Int. Conf. on High Performance Computing and Simulation (HPCS)*, pages 224–231, 2010.

[18] H. C. Edwards, C. R. Trott, and D. Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202–3216, 2014.

[19] G. Faanes, A. Bataineh, D. Roweth, E. Froese, B. Alverson, T. Johnson, J. Kopnick, M. Higgins, J. Reinhard, et al. Cray cascade: a scalable HPC system based on a Dragonfly network. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012.

[20] Fact Sheet: Collaboration of Oak Ridge, Argonne, and Livermore National Labs. `http://energy.gov/downloads/fact-sheet-collaboration-oak-ridge-argonne-and-livermore-coral` - [Online; last accessed 06/13/2017].

[21] I. Faraji. Accelerated MPI benchmark, https://github.com/imanfaraji/MPI-ACC (Last updated 09/20/2016).

[22] I. Faraji and A. Afsahi. Design considerations for GPU-aware collective communications in MPI. *Concurrency and Computation: Practice and Experience - Accepted for publication.*

[23] I. Faraji and A. Afsahi. GPU-aware intranode MPI_Allreduce. In *Proceedings of the 21st European MPI users' group meeting*, pages 45–50. ACM, 2014.

[24] I. Faraji and A. Afsahi. Hyper-Q aware intranode MPI collectives on the GPU. In *Proceedings of the First International Workshop on Extreme Scale Programming Models and Middleware*, ESPM2, pages 47–50, 2015.

[25] I. Faraji, S. H. Mirsadeghi, and A. Afsahi. Topology-aware GPU selection on multi-GPU nodes. In *Proc. International Parallel and Distributed Processing Symposium - Accelerators and Hybrid Exascale Systems Workshop (AsHES)*, pages 712–720, 2016.

[26] I. Faraji, S. H. Mirsadeghi, and A. Afsahi. Exploiting heterogeneity of communication channels for efficient GPU selection on multi-GPU nodes. *Parallel Computing Journal - in press - 0167-8191 http://dx.doi.org/10.1016/j.parco.2017.07.001*, 2017.

[27] 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. *Computer Physics Communications*, 192:97–107, 2015.

[28] R. L. Graham and G. Shipman. MPI Support for Multi-core Architectures: Optimized Shared Memory Collectives. In *European Parallel Virtual Machine/Message Passing Interface (PVM/MPI) Users Group Meeting*, pages 130–140. 2008.

[29] W. Gropp and K. Buschelman. FPMPI-2 fast profiling library for MPI. [Online; last accessed 10/14/2016].

[30] M. Guevara, C. Gregg, K. Hazelwood, and K. Skadron. Enabling task parallelism in the CUDA scheduler. In *Workshop on Programming Models for Emerging Architectures*, volume 9, 2009.

[31] S. Habib, A. Pope, H. Finkel, N. Frontiere, K. Heitmann, D. Daniel, P. Fasel, V. Morozov, G. Zagaris, T. Peterka, et al. HACC: Simulating sky surveys on state-of-the-art supercomputing architectures. *New Astronomy*, 42:49–65, 2016.

[32] InfiniBand Trade Association (IBTA), http://www.infinibandta.org/.

[33] Intel, An Introduction to the Intel QuickPath Interconnect. Intel White Paper, January 2009.

[34] Intel InfiniBand. `http://www.intel.com/Infiniband`. [Online; last accessed 07/14/2017].

[35] Intel MPI. `http://software.intel.com/en-us/intel-mpi-library`. [Online; last accessed 10/24/2017].

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

[37] J. Jenkins, J. Dinan, P. Balaji, T. Peterka, N. F. Samatova, and R. Thakur. Processing MPI derived datatypes on noncontiguous GPU-resident data. *IEEE Transactions on Parallel and Distributed Systems*, 25(10):2627–2637, 2014.

[38] J. Jenkins, J. Dinan, P. Balaji, N. F. Samatova, and R. Thakur. Enabling fast, noncontiguous GPU data movement in hybrid MPI + GPU environments. In *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*, pages 468–476, 2012.

[39] F. Ji, A. M. Aji, J. Dinan, D. Buntinas, P. Balaji, R. Thakur, W. Feng, and X. Ma. DMA-assisted, intranode communication in GPU accelerated systems. In *Proc. 14th International Conference on High Performance Computing and Communication & 9th International Conference on Embedded Software and Systems (HPCC-ICESS)*, pages 461–468, 2012.

[40] N. T. Karonis, B. R. De Supinski, I. Foster, W. Gropp, E. Lusk, and J. Bresnahan. Exploiting hierarchy in parallel computer networks to optimize collective operation performance. In *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, pages 377–384. IEEE, 2000.

[41] 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.

[42] 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.

[43] T. Kielmann, R. F. Hofman, H. E. Bal, A. Plaat, and R. A. Bhoedjang. MagPIe: MPI's collective communication operations for clustered wide area systems. *ACM Sigplan Notices*, 34(8):131–140, 1999.

[44] D. B. Kirk and W. H. Wen-Mei. *Programming massively parallel processors: a hands-on approach, 3rd edition*. Morgan Kaufmann, 2016.

[45] X. Lapillonne, O. Fuhrer, P. Spörri, C. Osuna, A. Walser, A. Arteaga, T. Gysi, S. Rüdisühli, K. Osterried, and T. Schulthess. Operational numerical weather

prediction on a GPU-accelerated cluster supercomputer. In *EGU General Assembly Conference Abstracts*, volume 18, page 13554, 2016.

[46] S. Li, T. Hoefler, and M. Snir. NUMA-Aware Shared Memory Collective Communication for MPI. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing (HPDC)*, pages 85–96, 2013.

[47] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE micro*, 28(2), 2008.

[48] 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.

[49] A. R. Mamidala, R. Kumar, D. De, and D. K. Panda. MPI collectives on modern multicore clusters: Performance optimizations and communication characteristics. In *Cluster Computing and the Grid, 2008. CCGRID. 8th IEEE International Symposium on*, pages 130–137, 2008.

[50] M. Martinasso, G. Kwasniewski, S. Alam, T. Schulthess, and T. Hoefler. A PCIe congestion-aware performance model for densely populated accelerator servers. In *In Proc. International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2016.

[51] G. Martinez, M. Gardner, and W.-c. Feng. Cu2cl: A cuda-to-opencl translator for multi-and many-core architectures. In *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, pages 300–307. IEEE, 2011.

[52] Mellanox Technologies. `http://www.mellanox.com/index.php`. [Online; last accessed 07/14/2017].

[53] G. Mercier and E. Jeannot. Improving MPI applications performance on multicore clusters with rank reordering. In *Recent Advances in the Message Passing Interface*, pages 39–49, 2011.

[54] 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 ()*, pages 386–396, 2016.

[55] S. H. Mirsadeghi, I. Faraji, and A. Afsahi. MAGC: A mapping approach for GPU clusters. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2016 28th International Symposium on*, pages 50–58. IEEE, 2016.

[56] MPI3.1. `http://www.mpi-forum.org/docs/mpi-3.1/`. [Online; last accessed 03/03/2017].

[57] MPICH. `http://www.mpich.org/`. [Online; last accessed 10/14/2016].

[58] A. Munshi. The OpenCL specification. In *Hot Chips 21 Symposium (HCS), 2009 IEEE*, pages 1–314, 2009.

[59] MVAPICH2. `http://mvapich.cse.ohio-state.edu`. [Online; last accessed 10/14/2016].

[60] NCCL - NVIDIA collective communication library (NCCL) `https://github.com/NVIDIA/nccl` (Last updated 09/20/2016).

[61] NVIDIA compute unified device architecture programming. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/`. [Online; last accessed 06/12/2017].

[62] NVIDIA Corporation, http://www.nvidia.com.

[63] NVIDIA management library, https://developer.nvidia.com/nvidia-management-library-nvml (Last accessed 09/24/2017).

[64] NVIDIA, "MPS", Sharing a GPU between MPI processes: Multi-Process Service - vR352, 2015.

[65] NVIDIA profiler user's guide, http://docs.nvidia.com/cuda/profiler-users-guide/. [Online; last accessed 09/24/2017].

[66] NVIDIA Tesla P100 - The Most Advanced Datacenter Accelerator Ever Built, https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf. [Online; last accessed 03/13/2017].

[67] C. Obrecht, F. Kuznik, B. Tourancheau, and J. Roux. Scalable lattice Boltzmann solvers for CUDA GPU clusters. *Parallel Computing*, 39(6):259–270, 2013.

[68] Open MPI. `http://www.open-mpi.org/`. [Online; last accessed 09/24/2017].

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

[70] A. J. Pena and S. R. Alam. Evaluation of inter-and intra-node data transfer efficiencies between GPU devices and their impact on scalable applications. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 144–151, 2013.

[71] A. Pompili, A. Di Florio, and C. Collaboration. GPUs for statistical data analysis in HEP: a performance study of GooFit on GPUs vs. RooFit on CPUs. In *Journal of Physics: Conference Series*, volume 762, pages 012–044, 2016.

[72] S. Potluri, K. Hamidouche, A. Venkatesh, D. Bureddy, and D. K. Panda. Efficient Inter-node MPI Communication using GPUDirect RDMA for InfiniBand Clusters with NVIDIA GPUs. In *Parallel Processing (ICPP), 2013 42nd International Conference*, pages 80–89, 2013.

[73] 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. Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1848–1857, 2012.

[74] Y. Qian and A. Afsahi. Process arrival pattern aware alltoall and allgather on infiniband clusters. *International Journal of Parallel Programming*, 39(4):473–493, 2011.

[75] R. Rabenseifner. Automatic MPI counter profiling of all users: First results on a CRAY T3E 900-512. In *Proceedings of the message passing interface developers and users conference*, volume 1999, pages 77–85, 1999.

[76] R. Rabenseifner. Optimization of collective reduction operations. In *International Conference on Computational Science*, pages 1–9. Springer, 2004.

[77] M. J. Rashti, J. Green, P. Balaji, A. Afsahi, and W. Gropp. Multi-core and network aware MPI topology functions. In *Recent Advances in the Message Passing Interface*, pages 50–60, 2011.

[78] RDMA Consortium. [Online]. available: http://www.rdmaconsortium.org/ (Last accessed 06/13/2017).

[79] 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. Symposium on Computers and Communications*, pages 811–817, 2009.

[80] P. Sanders and J. Träff. The hierarchical factor algorithm for all-to-all communication. *Euro-Par 2002 Parallel Processing*, pages 17–51, 2002.

[81] R. Shi, X. Lu, S. Potluri, K. Hamidouche, J. Zhang, and D. K. Panda. HAND: A hybrid approach to accelerate non-contiguous data movement using MPI datatypes on GPU clusters. In *Parallel Processing (ICPP), 2014 43rd International Conference on*, pages 221–230. IEEE, 2014.

[82] R. Shi, S. Potluri, K. Hamidouche, J. Perkins, M. Li, D. Rossetti, and D. K. D. Panda. Designing efficient small message transfer mechanism for inter-node MPI

communication on InfiniBand GPU clusters. In *High Performance Computing (HiPC), 2014 21st International Conference on*, pages 1–10, 2014.

[83] "Sierra", Lawrence Livermore National Laboratory. `http://computation.llnl.gov/computers/sierra`. [Online; last accessed 06/13/2017].

[84] A. K. Singh. *Optimizing All-to-All and Allgather Communications on GPGPU Clusters*. PhD thesis, The Ohio State University, 2012.

[85] 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 *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pages 420–427, 2011.

[86] S. Sistare, R. Vaart, and E. Loh. Optimization of MPI collectives on clusters of large-scale SMPs. In *Supercomputing, ACM/IEEE 1999 Conference*, pages 23–23. IEEE, 1999.

[87] D. Slogsnat, A. Giese, M. Nüssle, and U. Brüning. An open-source hypertransport core. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 1(3):14, 2008.

[88] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66–73, 2010.

[89] "SUMMIT", Oak Ridge National Laboratory. `https://www.olcf.ornl.gov/summit/`. [Online; last accessed 06/13/2017].

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

[91] The TOP500 June 2017 List. `https://www.top500.org/lists/2017/06/`. [Online; last accessed 09/03/2017].

[92] V. Tipparaju, J. Nieplocha, and D. Panda. Fast collective operations using shared and remote memory access protocols on clusters. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, 2003.

[93] J. L. Träff. Improved MPI all-to-all communication on a Giganet SMP cluster. *Lecture notes in computer science*, pages 392–400, 2002.

[94] J. L. Traff and A. Rougier. MPI collectives and datatypes for hierarchical all-to-all communication. In *Proceedings of the 21st European MPI users' group meeting*, page 27. ACM, 2014.

[95] S. S. Vadhiyar, G. E. Fagg, and J. Dongarra. Automatically tuned collective communications. In *Super Computing, International Conference on*, 2000.

[96] H. Wang, S. Potluri, M. Luo, A. K. Singh, X. Ouyang, S. Sur, and D. K. Panda. Optimized non-contiguous MPI datatype communication for GPU clusters: Design, implementation and evaluation with MVAPICH2. In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pages 308–316, 2011.

[97] H. Wang, S. Potluri, M. Luo, A. K. Singh, S. Sur, and D. K. Panda. MVAPICH2-GPU: optimized GPU to GPU communication for InfiniBand clusters. *Computer Science-Research and Development*, 26(3-4):257, 2011.

[98] L. Wang, M. Huang, and T. El-Ghazawi. Exploiting concurrent kernel execution on graphic processing units. In *High performance computing and simulation (HPCS), 2011 international conference on*, pages 24–32, 2011.

[99] L. Wang, M. Huang, and T. El-Ghazawi. Towards efficient GPU sharing on multicore processors. *ACM SIGMETRICS Performance Evaluation Revision*, 40(2):119–124, 2012.

[100] L. Wang, M. Huang, V. K. Narayana, and T. El-Ghazawi. Scaling scientific applications on clusters of hybrid multicore/GPU nodes. In *Proceedings of the 8th ACM international conference on computing frontiers*, page 6. ACM, 2011.

[101] F. Wende, F. Cordes, and T. Steinke. On improving the performance of multi-threaded CUDA applications with concurrent kernel execution by kernel reordering. In *Application Accelerators in High Performance Computing (SAAHPC), 2012 Symposium on*, pages 74–83, 2012.

[102] F. Wende, F. Cordes, and T. Steinke. Multi-threaded Kernel Offloading to GPGPU using Hyper-Q on Kepler Architecture. In *Technical Report 14-19, ZIB, Takustr*, 2014.

[103] M. Wolfe. The OpenACC application programming interface, version 2.0, 2013.

[104] W. Wu, G. Bosilca, R. Vandevaart, S. Jeaugey, and J. Dongarra. GPU-Aware Non-contiguous Data Movement In Open MPI. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pages 231–242, 2016.

Publication List

Journal Publications

- **Iman Faraji**, Seyed H. Mirsadeghi, and Ahmad Afsahi. Exploiting Heterogeneity of Communication Channels for Efficient GPU Selection on Multi-GPU Nodes. *Parallel Computing Journal* - in press - 0167-8191, http://dx.doi.org/10.1016/j.parco.2017.07.001, 2017.
- **Iman Faraji** and Ahmad Afsahi. Design Considerations for GPU-Aware Collective Communications in MPI. *Concurrency and Computation: Practice and Experience Journal - Accepted for publication.*

Conference and Workshop Publications

- **Iman Faraji**, Seyed H. Mirsadeghi, and Ahmad Afsahi. Topology-aware GPU selection on multi-GPU nodes. *In Proceedings of the International Parallel and Distributed Processing Symposium, Accelerators and Hybrid Exascale Systems Workshop (AsHES), 712–720, 2016* - ***Best Paper Award***.
- Seyed H Mirsadeghi, **Iman Faraji**, and Ahmad Afsahi. MAGC: A Mapping Approach for GPU Clusters. *In Computer Architecture and High Performance Computing (SBAC-PAD), 28th International Symposium on, 50–58, IEEE, 2016.*
- **Iman Faraji** and Ahmad Afsahi. Hyper-A Aware Intranode MPI Collectives on the GPU. *In Proceedings of the SuperComputing Conference, First International Workshop on Extreme Scale Programming Models and Middleware (ESPM2), pages 47–50, 2015.*

- **Iman Faraji** and Ahmad Afsahi. GPU-Aware Intranode MPI_Allreduce. *In Proceedings of the 21st European MPI Users' Group Meeting (EuroMPI), page 45–50. ACM, 2014.*