EFFICIENT PROCESS ARRIVAL PATTERN AWARE COLLECTIVE COMMUNICATION FOR HPC AND DEEP LEARNING

by

PEDRAM MOHAMMADALIZADEHBAKHTEVARI

A thesis submitted to the
Department of Electrical and Computer Engineering
in conformity with the requirements for
the degree of Master of Applied Science

Queen's University
Kingston, Ontario, Canada
November 2021

Abstract

High-Performance Computing (HPC) is the key to tackle computationally intensive problems such as Deep Learning (DL) and scientific applications. Message Passing Interface (MPI) is the de facto parallel programming standard that fulfills communication in HPC systems. MPI collective communication operations involve all processes within a program-defined group of processes and have been used extensively in parallel applications. Unfortunately, most of the studies attempting to improve the performance of collective operations are based on the premise that all the processes commence the communication simultaneously. However, researchers have shown that imbalanced Process Arrival Pattern (PAP) is ubiquitous in real environments. Therefore, it is important to propose new algorithms that improve the performance of collectives by PAP-awareness.

This thesis presents a complete communication characterization of Horovod as one of the most famous distributed DL frameworks. We provide a thorough study on PAP of the MPI_Allreduce as the most important collective operation used in Horovod and show that the arrival pattern of MPI processes is indeed imbalanced especially for small messages. Furthermore, we present various proposals for improving the MPI collective communication performance in the presence of imbalanced PAPs for different message sizes.

We propose an intra-node PAP-aware shared-memory-aware MPI_Allreduce algorithm for small messages that, based on the arrival time of the processes at each invocation of the collective call, dynamically chooses the leader process. The evaluation results show that our design delivers up to 56% improvement over native algorithms under different imbalanced PAPs.

We also propose a PAP-aware algorithm capable of dynamically constructing the reduction schedule at each invocation of the collective call based on the arrival order of the processes for intra-node MPI_Reduce and MPI_Allreduce collectives with large messages, achieving up to 73%, and 44% improvement over the state-of-the-art algorithms, respectively.

Finally, we evaluate the performance of two state-of-the-art cluster-wide MPI_Allreduce algorithms and introduce a PAP-tolerant cluster-wide allreduce algorithm which imposes less data dependency among processes given its hierarchical nature compared to flat algorithms. This algorithm delivers up to 58% improvement at the microbenchmark level and an average improvement of 10% for Horovod DL application over native algorithms.

Acknowledgments

First and foremost, I would like to thank my supervisor, Prof. Ahmad Afsahi, for his continuous and unconditional support and encouragement throughout my Master's studies. My thanks go to him for his priceless guidance toward high-quality research and admirable patience. Without him, this thesis would have never been feasible.

Many thanks to all my current and former colleagues at Parallel Processing Research Laboratory, Dr. Mahdieh Ghazi, Leila Habibpour, Benjamin Kitor, Amirhossein Sojoodi, and Yiltan Hassan Temucin; I will cherish your friendship forever. I would like to especially thank Amirhossein Sojoodi and Yiltan Hassan Temucin for their great help and support. I would also like to thank the Natural Science and Engineering Research Council of Canada (NSERC) for supporting this research. Thanks to Vector Institute for awarding me the prestigious Vector Scholarship in Artificial Intelligence. I would like to acknowledge the Department of Electrical and Computer Engineering of Queen's University as well as the school of graduate studies for their financial support. I would also like to thank Compute Canada for the access to their large-scale resources, such as Beluga and Helios in Calcul Québec and Cedar in WestGrid.

To all my friends in Tehran and Kingston, I cannot acknowledge your support enough. In alphabetical order, Azar, Alireza, Behnam, Ehson, Hadi, Majid, Maryam, Mehran, Mohammadreza, Mojtaba, Nima, Parisa, Pourya, Reza, Saber, and Sadegh, thank you.

Last but not least, I would like to thank my family for their patience and unconditional support over the years. To my father Nader, my mother Roya, my lovely sister Paniz, my grandparents Samad and Fakhri, you are the pure love that I carry with myself.

Statement of Collaboration

The work in Chapter 3 was done collaboratively with Yiltan Hassan Temucin. We worked together on the development of the profiler for extracting the communication characterization of Horovod beneficial to both of our works. All other codes, including the process arrival pattern analysis code, data collection, pre- and post-analysis of the results in this chapter were accomplished by myself. The work in Chapter 4 was done collaboratively with Amirhossein Sojoodi, who helped me in verifying the validity of the code that I developed for constructing shared-memory between MPI processes in distributed systems.

Contents

List of Algorithms

List of Tables

List of Figures

Listings

Glossary

CA	Channel Adapter
CFD	Computational Fluid Dynamic
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DL	Deep Learning
DMA	Direct Memory Access
DNN	Deep Neural Network
FLOPS	FLoating-point Operations Per Second
GPU	Graphics Processing Unit
HCA	Host Channel Adapter
HPC	High-Performance Computing
IB	InfiniBand
IF	Imbalance Factor
MIF	Maximum Imbalance Factor
MPI	Message Passing Interface
NIC	Network Interface Card
OS	Operating System

PAP	Process Arrival Pattern
PAT	Process Arrival Time
PCIe	Peripheral Component Interconnect Express
PGAS	Partitioned Global Address Space
PMPI	MPI Profiling Interface
PPN	Process Per Node
PRR	Pre-Reduced Ring
PS	Parameter Server
RD	Recursive Doubling algorithm
RDMA	Remote Direct Memory Access
RMA	Remote Memory Access
RSA	Reduce Scatter Allgather
RSG	Reduce Scatter Gather
SLT	Sorted Linear Tree
TCA	Target Channel Adapter

Chapter 1

Introduction

Artificial Intelligence (AI) has been evolving from a hot new trend to a seamless enabler of business transformation. Machine Learning (ML) and Deep Learning (DL) enable AI systems to learn without being explicitly programmed. As a result, DL applications have been growing in prominence as a way to automatically characterize objects, trends, and anomalies merely by observing large amounts of data. However, as datasets increase in size and the DL models in complexity, the computational intensity increases proportionally, resulting in significantly long if not unfeasible training times on a single sequential processor for DL applications. This ever-growing demand in computational power is common in other scientific and engineering areas such as Genomics, Molecular Sciences, Physics, and Mechanics, to name a few.

High-Performance Computing (HPC) is the answer to execute such applications in a sensible amount of time. Parallel computing, as the primary approach exploited in HPC systems, enhances the performance of computationally intensive applications by dividing the main problem into smaller sub-problems that can be solved on different processing units of a parallel computer simultaneously. Cluster computers are the most popular type of parallel computers, making up 93% of the top 500 supercomputers in the world [7].

A cluster computer comprises multiple compute nodes, each having several processing elements. The compute nodes are connected to each other by a high-performance interconnection network. The processing elements compute and then communicate with each other to exchange the intermediate results and synchronize.

Message Passing Interface (MPI) [3] is the most used parallel programming paradigm in HPC applications. In this standard, the communication between the processing units is performed using explicit message transfers. The MPI standard provides different communication semantics such as one-sided, point-to-point, partitioned point-to-point, and collective operations. Collective communication operations involve more than two processes. Utilizing collective operations, processes can perform one-to-all, all-to-one, and all-to-all communications in an optimized yet convenient way. Collective communications have been used in many parallel applications including DL applications because they offer performance portability and scalability. In some HPC applications, more than eighty percent of the overall communication time is spent in collective operations [56]. Therefore, the performance of collective communication operations is critical to the performance of HPC applications [31].

1.1 Motivation

Optimizing collective operations has been an active area of research for a long time [10, 48, 58, 62, 70, 32, 26]. Studies have shown that optimizing collective operations, specifically MPI_Allreduce and MPI_Bcast, two most exploited collectives in DL applications [13], can significantly improve the performance of DL applications such as Horovod [60] and CNTK [59] distributed DL frameworks [17, 63]. Reduction operations are among the most commonly used collective communication operations in HPC applications [56]. Consequently, the researchers have been ever optimizing the performance of these collectives from different aspects [67, 30, 57, 34, 49, 24, 63].

Unfortunately, however, similar to other collective operations, reduction algorithms have been optimized only under the premise that all processes start the operation at the same time. Research conducted on the arrival time of processes at the collective calls has shown that this is rarely the case and that imbalanced *Process Arrival Patterns (PAPs)* are ubiquitous in HPC applications [21, 55, 44]. Thus, the benchmarking methodologies should consider imbalanced arrival times to depict the accurate picture of real-world algorithm performance. It has been shown that the well-performing algorithms for the balanced microbenchmarks, which are usually the algorithms of interest in MPI implementations, perform poorly in imbalanced process arrival patterns [21]. Therefore, it is necessary to propose new algorithms capable of exploiting process imbalance to deliver high performance when there is an asynchrony among the processes at the start of the collective operation.

1.2 Problem Statement

In this section, we present the research questions we sought to address in this thesis:

1. What are the communication characteristics of Horovod application [60], as one of the most famous distributed deep learning frameworks? Do Horovod collective communications suffer from imbalanced process arrival patterns similar to traditional HPC applications? What should be the focus of researchers to enhance the performance of Horovod?

2. How can we design an intra-node MPI_Allreduce algorithm for small message sizes

capable of achieving high performance under different process arrival patterns? How can we dynamically change the leader of the operation based on the arrival order of the processes at each invocation of the collective call?

3. How can we design intra-node MPI_Allreduce and reduce algorithms for large message sizes that could exploit the imbalance in the arrival time of the processes to improve the performance of the collective operation? How can we dynamically construct the reduction schedule at each invocation of the collective call without having any pre-knowledge of process arrival times?

4. Are hierarchical algorithms for inter-node MPI_Allreduce collectives capable of outperforming their flat counterparts under different process arrival patterns? What algorithms should be utilized when facing applications with balanced or imbalanced workloads? Based on the communication characterization and process arrival pattern of distributed DL frameworks, what is the best performing algorithm delivering the highest throughput?

1.3 Contributions

This thesis contributes by studying the PAP behavior of the Horovod application, as one of the most famous distributed deep learning (DL) frameworks, and then proposes different PAP-aware designs capable of delivering high-performance under imbalanced PAPs, as follows:

1.3.1 Communication Characterization of Horovod

In Chapter 3, we investigate different collective communication characteristics of the Horovod application in terms of their frequency and contribution to the communication runtime. We show that the MPI_Allreduce plays the most critical role in the performance of Horovod. We further characterize the communication behavior of the MPI_Allreduce operation used in Horovod by presenting the contribution of different message sizes to the number of calls and the overall communication runtime of this collective operation. Finally, we examine the process arrival pattern of the MPI_Allreduce operations. We provide evidence that the MPI processes in Horovod arrive asynchronously at the MPI_Allreduce collective calls. Especially, the MPI_Allreduce calls with small message sizes exhibit significantly large imbalance factors (IFs). The findings highlighted in this chapter will provide the HPC researchers with an understanding of the potential bottlenecks of Horovod and opportunities to improve its performance.

1.3.2 Intra-node PAP-aware MPI_Allreduce for Small Messages

Chapter 4 proposes an intra-node PAP-aware MPI_Allreduce algorithm for small messages. The state-of-the-art algorithms used for small message MPI_Allreduce collectives in modern HPC systems exploit the fast shared memory available on each node of the cluster to reduce communication latency [53, 8, 28, 61, 65, 33, 38]. The majority of these algorithms have been designed for synchronous *Process Arrival Times (PATs)* and are not optimized for imbalanced PAPs ubiquitous in many parallel systems. We propose a PAP-aware shared-memory aware MPI_Allreduce algorithm that dynamically chooses the leader of the operation based on the arrival time of the processes at each invocation of the collective call.

We implemented our PAP-aware design on top of the shared-memory MPI_Allreduce algorithm of MVAPICH. Then, we present the evaluation of our design against the native algorithms utilized in MVAPICH at the microbenchmark level for MPI_Allreduce operations with messages up to 64KBs on two platforms. The experimental evaluations show up to 36% and 56% improvement over native algorithms under different imbalanced process arrival patterns, with 32 Processes Per Node (PPN), on Beluga and Cedar clusters at Compute Canada, respectively.

1.3.3 Intra-node PAP-aware MPI_Reduce/Allreduce for Large Messages

In Chapter 5, we propose a PAP-aware algorithm capable of exploiting the imbalance in the process arrivals to improve the performance of intra-node MPI_Reduce and MPI_Allreduce collectives with large message sizes. Our proposed algorithm dynamically constructs the reduction schedule at each invocation of the collective call based on the arrival order of the processes. It is noteworthy that our design does not need any prior knowledge about the process arrival times when scheduling the reduction operation. The PAP-aware algorithm minimizes the time each process spends in the collective call by letting the arriving processes contribute their data and leave the collective call as soon as possible. We provide the performance evaluation of our algorithm using balanced and imbalanced microbenchmarks with different process arrival patterns on two platforms. For the MPI_Reduce operation, the experimental evaluations show up to 30%, and 73% improvement over the best performing native algorithms of MVAPICH under different imbalanced process arrival patterns, with 4 and 16 PPN, on Cedar and Helios clusters at Compute Canada, respectively. In addition, for the MPI_Allreduce operation, the experiments show the maximum improvement of 44% over the best performing native algorithm of MVAPICH under various tested PAPs with 4 PPN on Cedar cluster computer.

1.3.4 Cluster-wide PAP-tolerant MPI_Allreduce for Large Messages

Chapter 6 investigates a cluster-wide MPI_Allreduce algorithm for large messages capable of delivering high performance under imbalanced workloads. We evaluate the performance of two famous cluster-wide MPI_Allreduce algorithms, flat reduce-scatter followed by allgather (RSA) and hierarchical RSA algorithms with balanced and imbalanced microbenchmarks as well as the Horovod application. The microbenchmark experiments show that the flat algorithm delivers the best performance in balanced PATs. However, when there are deviations in the arrival pattern of the processes, the flat algorithm suffers from performance degradation. On the other hand, hierarchical algorithms perform better under imbalanced PAPs than their flat counterparts since they impose less data dependency on the participating processes. The imbalanced microbenchmark results show that for 64KB to 64MB messages, the hierarchical algorithm, compared to its flat counterpart, improves the latency of the MPI_Allreduce operation by up to 57%. The application-level evaluation with Horovod also presents that using the hierarchical algorithm instead of the flat algorithm improves the throughput of Horovod for all the studies we conducted with different GPU counts by an average of 10%.

1.4 Organization of the Thesis

The rest of this thesis is divided into six chapters. Chapter 2 provides some background information and lays the groundwork for the following chapters. It starts by discussing cluster computers as the most common types of parallel computers. It then explains different message-passing semantics and features specified within the MPI standard as the de-facto parallel programming paradigm. Next, it explains the PAP and the metrics used to describe it since our proposals in this thesis are primarily based on this research front in MPI. Finally, it briefly discusses the Horovod distributed deep learning framework used as our understudy application. In Chapter 3, we study the communication characterization of Horovod and present an in-depth process arrival pattern analysis of this parallel application. Chapter 4 proposes a novel approach for improving intra-node MPI_Allreduce collective operation under imbalanced process arrival patterns for small messages using shared memory, followed by its performance evaluation. Chapter 5 discusses our PAP-aware algorithm for improving the performance of intra-node MPI_Reduce and MPI_Allreduce collectives for large message sizes and provides the microbenchmark results against some famous algorithms implemented in MVAPICH. Chapter 6 introduces an inter-node PAP-tolerant solution for the MPI_Allreduce collective call for large message sizes, achieving high performance under imbalanced PAPs, followed by microbenchmark and Horovod application results. Finally, Chapter 7 concludes the thesis and provides some potential future research directions.

Chapter 2

Background

In this chapter, we lay the foundation for the next chapters. We provide some background information about HPC and its application that are relevant to this thesis. We provide an overview of some interconnection networks used in HPC systems. Then, we present MPI and its communication semantics, including collective communication operations. We discuss some well-known algorithms used in MPI libraries for the implementation of some collective communications. Furthermore, we discuss the Process Arrival Pattern issue of collectives and the metrics used to measure it. In the end, we discuss distributed Deep Learning frameworks as one of the most important applications in the HPC domain, and for that we specifically introduce Horovod as the application studied in this thesis.

2.1 High-Performance Computing (HPC)

As the complexity of algorithms and the volume of datasets used in scientific and engineering applications increases, the computational intensity grows proportionally, resulting in significantly long, even impractical, execution time on an ordinary computer. HPC is the key to execute such applications in a reasonable amount of time. Computational Fluid Dynamic (CFD) [43], genomics [36], financial risk modeling [71], weather prediction [35], as well as emerging applications, such as machine learning [9] and Deep Learning [59, 60] are just a few examples of computationally intensive HPC application domains. The HPC community investigates cutting-edge hardware, software, as well as novel techniques to deliver state-of-the-art computational power required by HPC applications.

Parallel computing is the principal approach to satisfy the ever-growing demands for computational power. In parallel computing, an application is broken down into a number of sub-tasks that can be processed independently. Each sub-task is then executed by one of the many processing units of the system. This way, parallel computers can utilize the computation power of multiple processors to enhance the performance of applications.

Cluster computers are one of the most common types of parallel computers. At the time of writing this thesis, cluster computers account for 93% of the top 500 supercomputers in the world [7]. Figure 2.1 depicts the architecture of a cluster computer, which consists of multiple compute nodes connected to each other via a high-performance interconnection network. The computations are executed on the processing elements within the compute nodes. Whenever a synchronization needs to be performed among the processing elements or exchange of intermediate results is required, the interconnection network is utilized. The multi-core processors are the main component of each compute node and may be supplemented by accelerators such as GPUs, which are becoming inevitable parts of modern cluster computers. GPUs have a huge number of low-power cores compared to multi-core processors, which is ideal for massively data-parallel applications such as mathematical libraries or kernels working with large datasets that utilize simple instructions but for a large number of invocations.

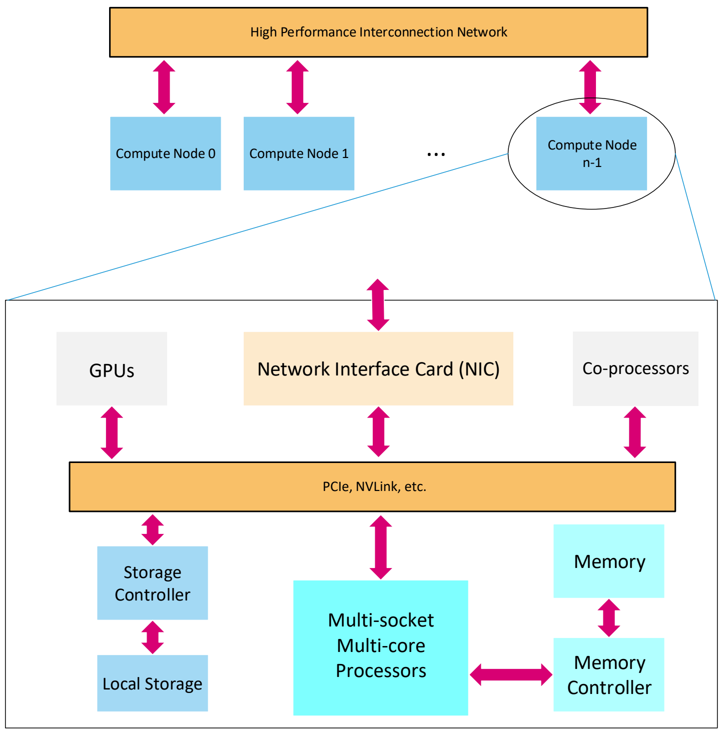


Figure 2.1: An example of a typical HPC cluster

Floating-point operations per second (FLOPS) is the metric used by Top500 [7] to measure the compute power and rank the fastest supercomputers in the world. Currently, Fugaku, Summit, and Sierra are the top three publicly known supercomputers, each having millions of processors and capable of executing operations at the petascale computing (10^{15} FLOPS) level. The ever-increasing size and the complexity of current and emerging problems lead to the demand for even higher computational power. Consequently, the HPC community aims to achieve exascale computing (10^{18}FLOPS) level in the near future.

HPC systems could only deliver as much computing power to the applications as they have the potential to, only if software layers would utilize the hardware resources optimally. One of the most important bottlenecks in achieving maximum performance in HPC systems is the communication between the processes. As mentioned earlier, in an HPC system, the processes are distributed among the compute nodes. When the computation on each process is finished, a communication mechanism is required to help the processes exchange the intermediate data or synchronize with each other. However, as the number of processing elements grows in parallel computers, the communication between them becomes critical in achieving high performance. Therefore, the HPC community has been ever enhancing the communication performance of parallel applications. Parallel programming paradigms such as *Shared Memory* [19], *Partitioned Global Address Space (PGAS)* [6], and *Message Passing* have been introduced as a software layer to support the communication between the processes. MPI is the most frequently used parallel programming model in HPC clusters. In the following sub-section, we further introduce MPI as the main focus of this thesis.

2.1.1 High-Performance Interconnects

Interconnection networks connect the compute nodes on a cluster computer so that they can communicate with each other. In order for HPC applications to scale appropriately on multiple machines, low latency and high bandwidth communication between the compute nodes is necessary. Different interconnection networks are used in different cluster computers. InfiniBand [2], Omni-Path [15], Ethernet and proprietary interconnects are the most commonly used interconnection networks in HPC. InfiniBand (IB) has become the de-facto standard architecture used to interconnect servers. This standard uses switched fabric network topology and delivers decent scalability by providing very low latency and high throughput. In InfiniBand, all the transmissions start or end at a Channel Adapter (CA). Each processor is connected to the network by a Host Channel Adapter (HCA), and each peripheral device is connected to the network via a Target Channel Adapter (TCA). As opposed to older networks, InfiniBand can bypass the Operating System (OS) for transferring the messages. HCAs are able to directly access application memory buffers without calling into the OS kernel using Direct Memory Access (DMA). This feature of IB ensures low latency and high application performance. At the time of writing this thesis, InfiniBand, Ethernet, and Omni-Path are used in 31%, 50.8%, and 9.4% of the top 500 supercomputers in the world, respectively. The performance share of the aforementioned interconnects are, however, 40%, 19.6%, and 7.6% in the same order [7]. The remaining share belongs to proprietary interconnects.

2.2 Message Passing Interface (MPI)

As it can be seen in Figure 2.2, MPI functions as an interface between the high-level application layer and the lower-level system layers. It provides the user application with the abstraction of the underlying network hardware. This standard introduces different communication semantics such as one-sided, point-to-point, partitioned point-to-point, and collective operations. We briefly introduce point-to-point, partitioned point-to-point, and RMA for completeness, as they are not the focus of this thesis. Then, we discuss collective communications in detail.

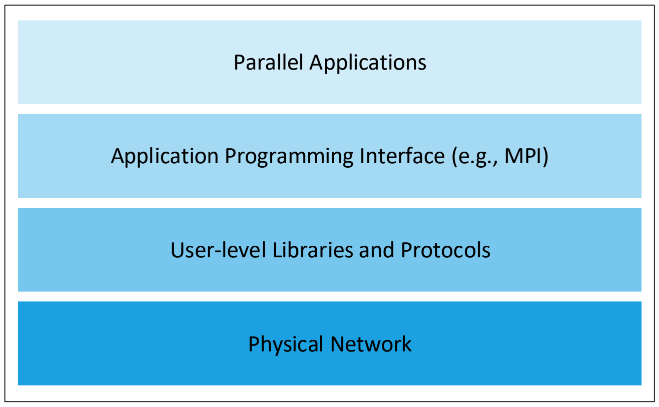

Parallel Applications
Application Programming Interface (e.g., MPI)
User-level Libraries and Protocols
Physical Network

Figure 2.2: Layers of abstraction in a parallel system

2.2.1 Point-to-point Communications

Point-to-point or two-sided communication involves only two processes, namely the source and the destination processes. Using MPI_Send family of calls, the source process initiates sending a message to the destination process. On the other hand, the destination process initiates a receive request by calling the MPI_Recv family of calls. The message traveling from the sender process contains the actual data that is to be sent, the *datatype* of each element of the data, the number of elements the data is comprised of, the identification number (*tag*) of the message, *communicator ID* which is a handle specifying a group of processes, and the *rank*s of the source and destination processes within the defined *communicator ID*. The *tag*, *communicator ID*, and *rank*s parameters will be used by the receiving side to match the sent message. The destination process then stores the received data in its receive buffer. The point-to-point communications in MPI are available in blocking and non-blocking fashions. In a blocking send call (MPI_Send), the sender process is blocked until its local send buffer can be used again. The blocking receive operation (MPI_Recv) blocks the caller process until the expected message arrives. The non-blocking send operation (MPI_Isend) initiates the send operation and returns before the message is copied out of the send buffer. The non-blocking receive operation (MPI_Irecv) initiates the receive operation and returns once the receive request is posted. Therefore, these two calls do not guarantee the completion of the corresponding calls. A send/receive request is completed by MPI_Wait family of calls. Using non-blocking communication usually leads to better performance by providing the opportunity to overlap communication and computation. There are numerous studies in the literature on improving the performance of point-to-point communications [63, 68, 47, 69, 27].

2.2.2 Partitioned Point-to-point Communications

Prior to the introduction of the MPI-4.0 standard [3], MPI provided all multi-threaded functionality through an interface that was designed primarily for single-threaded operations. To address the requirements of multi-threaded communication, a unique interface, partitioned point-to-point, has been introduced in the MPI-4.0 standard. Partitioned point-to-point communication is ”partitioned” because it provides a threaded interface for message passing that allows multiple parallel contributions of data to be made from potentially multiple threads or tasks to a single communication operation. With partitioned point-to-point primitives, while the HPC application provides partial contributions using multiple threads or tasks, the MPI library can transfer parts of the data buffer to the destination buffer. This way, utilizing partitioned communications in multi-threaded applications potentially improves the performance with overlapping buffer completion with its transfer.

2.2.3 One-sided Communications

One-sided communication, also called Remote Memory Access (RMA), allows a process to access the address space of another process without any explicit participation by the remote process. In RMA communication, each process exposes a part of its memory to other processes so that they can directly read from or write to the exposed memory on the remote process without the need to synchronize with it. The remote process is called the *target* process, and the exposed buffer on it is called a *window*. The process that performs the one-sided operation on the *target* process's *window* is called the *origin* process. Using MPI_Get and MPI_Put calls, the *origin* process reads/writes data from/to the *window* allocated on the remote process. The RMA communication is especially advantageous in applications with dynamic, unstructured computations, where the communication patterns change throughout the computation.

2.2.4 Collective Communications

Collective communication operations involve multiple processes on the system. With these operations, processes are able to perform one-to-many, many-to-one, and many-to-many communications in an optimized, yet convenient way. Examples of one-to-many collective communications within the MPI standard are broadcast and scatter. In an MPI broadcast operation (MPI_Bcast), a process sends the same data to all other processes in the communicator, while in MPI scatter operation (MPI_Scatter), one process sends different chunks of data to different processes. Gather and reduce are two examples of many-to-one collective operations defined within the MPI standard. Using MPI gather operation (MPI_Gather), one process gathers data from different processes in the communicator. MPI reduce operation (MPI_Reduce) is similar to MPI gather, except that a reduction operation is performed on the data supplied by the contributing processes. Alltoall, allgather, reduce-scatter, and allreduce collective operations are some examples of many-to-many collective communications. In an MPI alltoall operation (MPI_Alltoall), also called complete exchange operation, all processes scatter and gather data to and from every other process in the communicator. MPI_Allgather operation is a variation of MPI gather where each process gathers data from all the processes within the communicator. MPI reduce-scatter is a variant of MPI reduce where the result of the reduce operation is scattered to all the members in the group of participating processes. MPI_Allreduce operation is another variation of MPI reduce where all the processes within the communicator receive the result.

MPI_Allreduce and MPI_Reduce are among the most commonly used collective communication operations in HPC applications [56, 18, 14]. Therefore, various algorithms and techniques have been proposed by researchers to improve the performance of these operations for different specifications such as topologies, network interconnects, hardware technologies, number of processes, and message sizes [67, 30, 57, 34, 49, 25].

Flat algorithms aim to improve the performance of collectives by increasing the bandwidth utilization. The design of these algorithms is based on the flat communication model. This model assumes that the communication between any pair of processes occurs at the same cost. Ring, Recursive Doubling, Binomial-Tree, and Rabenseifner's algorithms, are some examples of flat algorithms.

The *Ring* algorithm for allreduce, shown in Figure 2.3, uses the nearest-neighbor communication pattern and is composed of computation and distribution phases. In this algorithm, the data on each process is divided into p chunks. At each round of the computation phase each process sends a chunk of its data to its right neighbor (the chunk index is determined based on the round number and the process's rank). Then, it receives a chunk of data from its left neighbor, and performs a reduction operation on the received data before forwarding it to the right neighbor in the next round. After $p - 1$ rounds of computation phase, each process has exactly one chunk fully reduced. Then, the distribution phase begins. In $p - 1$ rounds of this phase, each reduced chunk is gathered by all the processes with the same fashion (each process receives a reduced chunk from its left neighbor and forwards it to its right neighbor), so the final result will be available on all the participating processes [64].

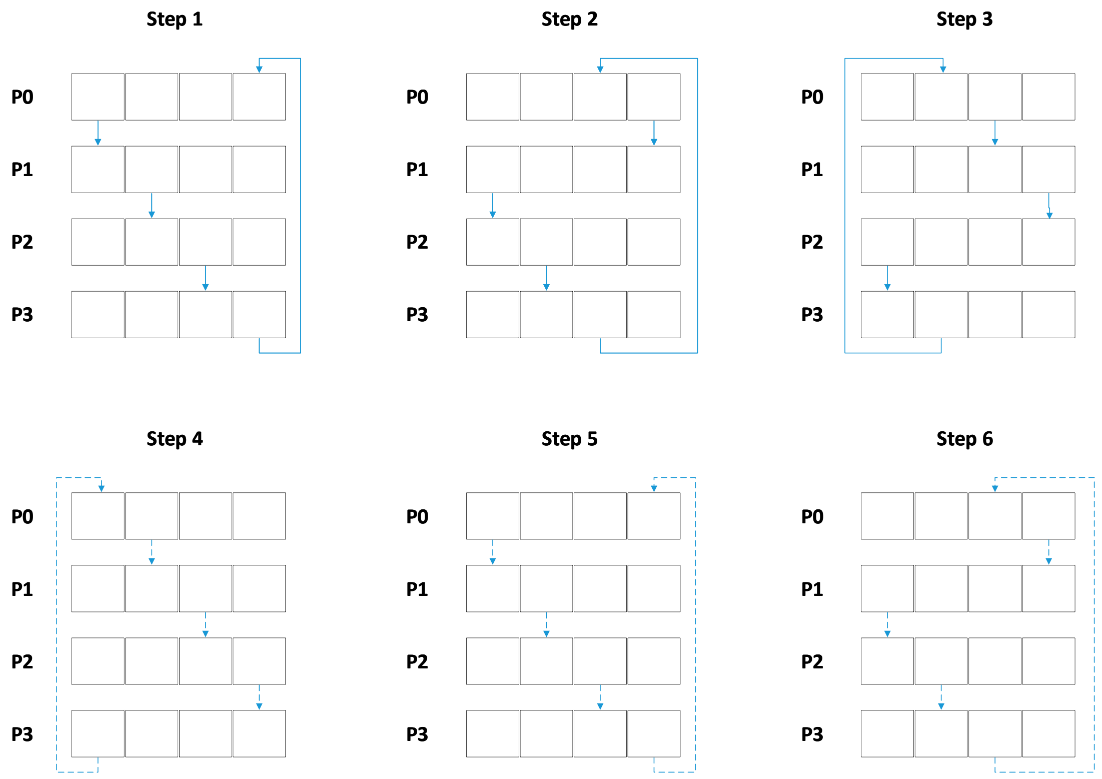


Figure 2.3: Ring algorithm for Allreduce with four processes (Solid arrows represent the computation phase, while the dotted arrows illustrate the distribution phase. The reduction operation is not shown.)

In the first step of the *Recursive Doubling (RD)* algorithm for allreduce, the processes with a distance one apart exchange their data. The distance between the communicating processes is doubled in each algorithm's step. Each step also involves a local reduction by the corresponding processes. For a power-of-two number of processes, the algorithm continues for $\log_2 p$ steps, where p is the number of processes [64]. Figure 2.4 illustrates how the recursive doubling algorithm works.

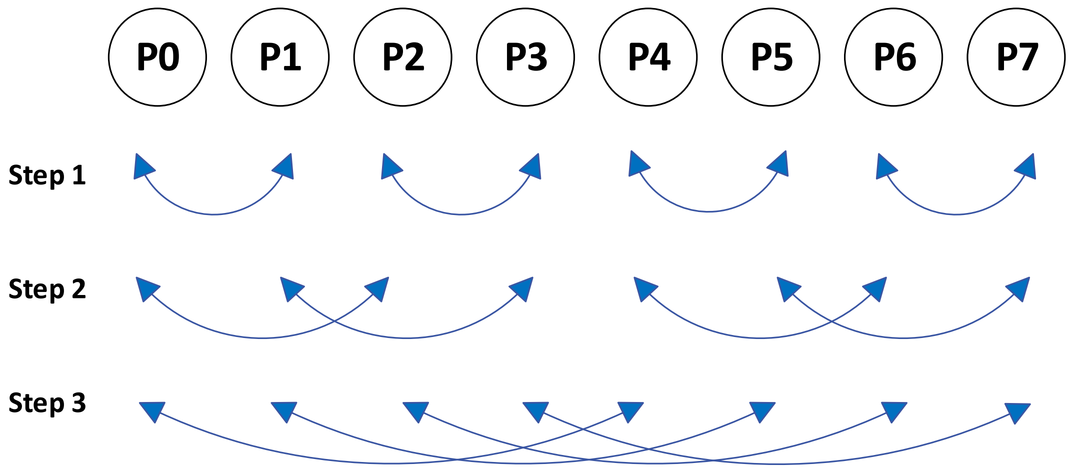


Figure 2.4: Recursive Doubling algorithm for Allreduce with eight processes (The reduction operation is not shown.)

Figure 2.5 demonstrates the *Binomial-Tree* algorithm for reduce. This algorithm is similar to the recursive doubling algorithm, except that in binomial tree algorithm, after each step, half of the processes finish their contribution to the operation and become inactive. This algorithm consists of $\lceil \log_2 p \rceil$ steps. At the end of the last step, the root process holds the result of the reduce operation [4]. The binomial tree allreduce operation, can be implemented by the aforementioned reduce operation followed by a broadcast operation. The *Rabenseifner's* algorithm for allreduce, also called *Reduce-Scatter Allgather (RSA)* algorithm, is a combination of a reduce-scatter followed by an allgather operation. The reduce-scatter operation in RSA is implemented with the recursive doubling algorithm (Figure 2.4). The allgather operation, however, uses the *Recursive Halving* algorithm presented in Figure 2.6. The Rabenseifner's algorithm for reduce, performs the same Reduce-Scatter algorithm as RSA. However, instead of an allgather operation it uses a gather to the root operation implemented with the binomial tree algorithm [57].

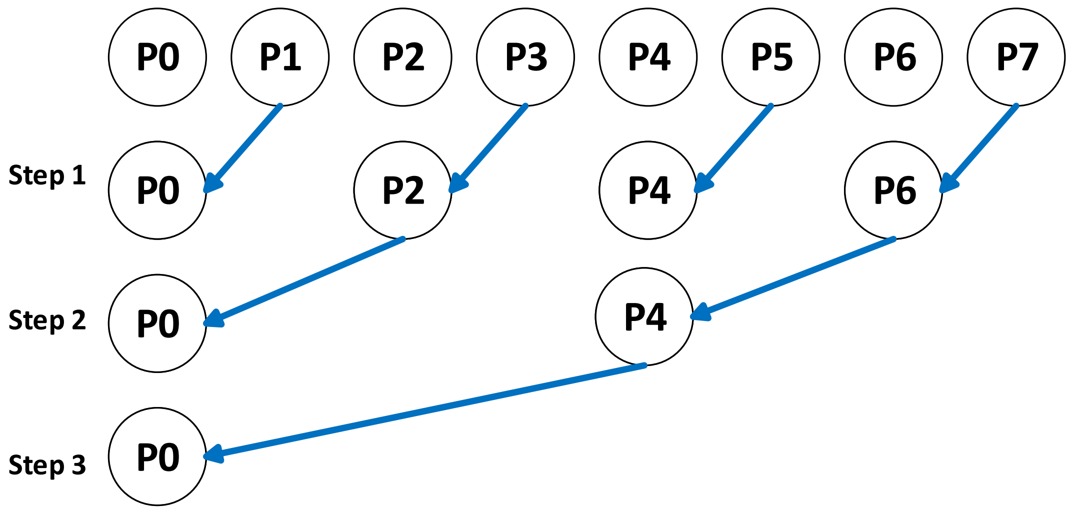


Figure 2.5: Binomial-Tree algorithm for Allreduce with eight processes (The reduction operation is not shown.)

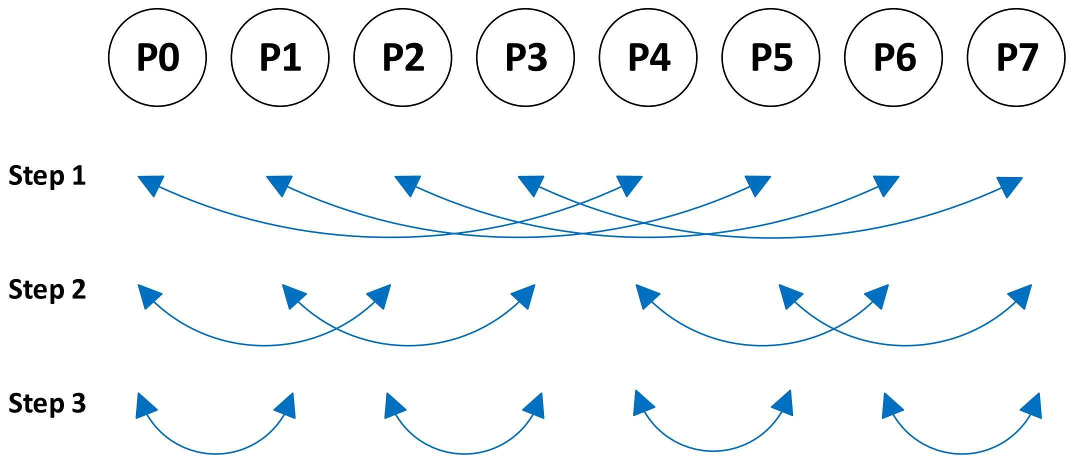


Figure 2.6: Recursive Halving algorithm for Allreduce with eight processes (The reduction operation is not shown.)

Hierarchical algorithms, on the other hand, aim to scale and perform better by minimizing the data transfers through the slower communication channels and instead utilizing faster communication channels available on a multicore cluster. The hierarchical algorithms usually perform in several stages. In these algorithms, the process communicator splits into sub-communicators, each having a local root (leader) process. Processes within a sub-communicator communicate through the faster communication channels, such as shared memory, and inter sub-communicator communications are executed through the slower channels, such as network interconnects [70]. The performance of existing hierarchical algorithms for several MPI implementations has been evaluated and optimized for various routines on platforms with different architectures and hardware supports in [72, 29, 70, 30].

Collective communications have been extensively used by a large number of MPI applications because of their ease-of-use, and potential performance and scalability. The performance of collective operations has a pivotal effect on the performance of MPI applications. Hence, researchers have conducted numerous investigations in order to improve the performance of MPI collective communications over the years [16, 64, 42, 32, 22, 66, 37]. Most of these works along with the existing algorithms for MPI collective operations have been designed and evaluated under an impractical premise that all the processes arrive at the collective call simultaneously and participate in the execution of the operation at the same time. While this assumption is correct for microbenchmark analysis, as shown in [21], even in programs with perfectly balanced workloads the processes arrive at the collective site at notably different times. The imbalance in the arrival time of processes is believed to be the consequence of various system features, including but not limited to operating system noise, cache misses, hardware interrupts, branch mispredictions, and stall CPU cycles. The arrival pattern of processes can significantly affect the performance of collective communication operations in that it determines the time each process can start its contribution to the operation. Therefore, it is remarkably important to evaluate the performance of existing collective communication operations under different arrival patterns of MPI processes and propose new algorithms to deliver high throughput for a broad range of process arrival patterns. There has been a limited number of works in the literature studying the impact of arrival pattern of processes on the performance of different collective operations and only a few of them have proposed new algorithms which perform efficiently when there is an asynchrony among the processes at the start of the collective operation execution [21, 44, 45, 46, 55, 52, 39, 40].

2.3 Process Arrival Pattern

In this section, we introduce the metrics used in the literature to describe the arrival pattern of processes in the applications. Considering the fact that each process involved in a collective operation arrives at the operation at a different time, the process arrival pattern of the collective call can be presented by the set of unique times each process arrives at the collective site. With a given world size of *n* processes $P_0, P_1, ..., P_{n-1}$, the PAP can be presented by the tuple $(a_0, a_1, ..., a_{n-1})$ as illustrated in Figure 2.7. The PAP is considered to be balanced if all the processes arrive at the collective site simultaneously, and is imbalanced otherwise. The imbalance in the PAP can be described by *average-case imbalance time* and *worst-case imbalance time* metrics that were originally defined in [21].

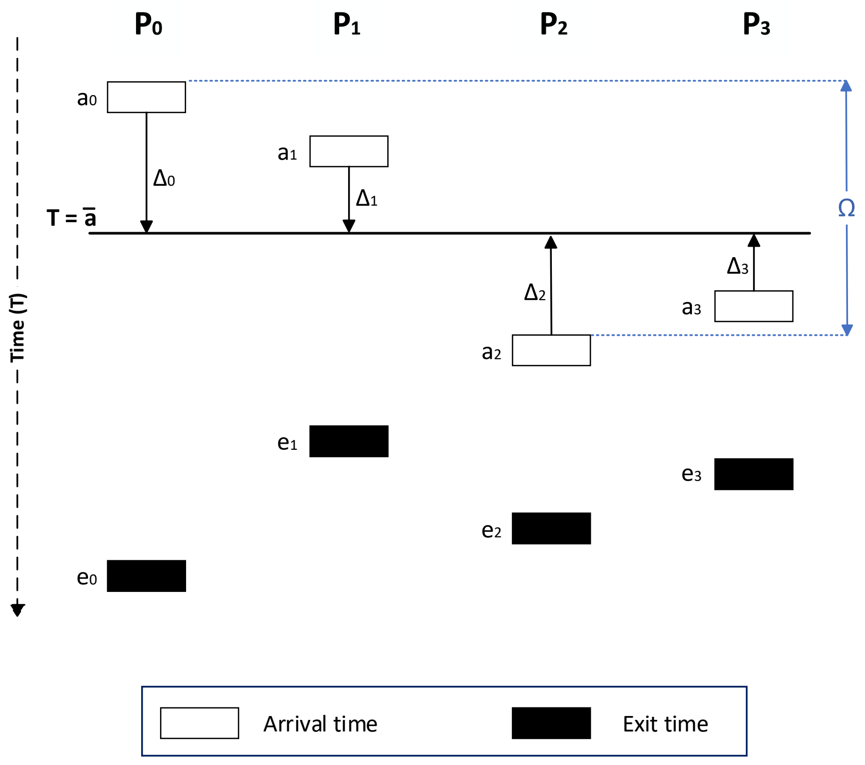


Figure 2.7: Process arrival pattern parameters

The *worst-case imbalance time* (Ω), defined in Equation (2.1), denotes the difference in time between the earliest arriving process and the last process entering the collective. *Average-case imbalance time* (Δ), on the other hand, is the average time difference between the arrival time of each process and the average of arrival times. The average of arrival times ($\bar{a}$), and the *average-case imbalance time* have been illustrated in Equation (2.2 (a)) and Equation (2.3), respectively.

$$\Omega = max_i\{a_i\} - min_i\{a_i\} \tag{2.1}$$

$$\bar{a} = \frac{a_0 + a_1 + ... + a_{n-1}}{n} \tag{2.2 (a)}$$

$$\Delta_i = |a_i - \bar{a}| \tag{2.2 (b)}$$

$$\bar{\Delta} = \frac{\Delta_0 + \Delta_1 + ... + \Delta_{n-1}}{n} \tag{2.3}$$

Average-case imbalance factor and *worst-case imbalance factor* are two other metrics that are very useful for characterizing the imbalanced PAPs. Let α be the time it takes to communicate a message (the size of this message is equal to the size of the message in the collective operation). The *average-case imbalance factor* $\frac{\bar{\Delta}}{\alpha}$, and *worst-case imbalance factor* $\frac{\Omega}{\alpha}$ are defined as the *average-case imbalance time* and *worst-case imbalance time* normalized by the time α.

2.4 Deep Learning Frameworks

Deep Learning frameworks are designed to support the execution and design of different types of Deep Neural Networks (DNN). The Distributed Deep Learning Frameworks make the distributed training of the DL models possible. These frameworks exploit the inherent parallelism of the DNNs to make the training task distributed. With the utilization of these frameworks, an existing single-node training can be scaled up to run on hundreds of GPUs on multiple cluster nodes. There are three techniques used by the Deep Learning frameworks for distributed training, model parallelism, data parallelism, and hybrid parallelism.

In the model parallelism technique, different parts of a single DL model are implemented on different computational units within a distributed cluster. Hence, the computations regarding each part of the model are assigned to different machines. This distributed training method is advantageous for very large DL models where the whole model cannot be implemented on a single machine within the cluster. The data parallel technique, on the other hand, implements a complete copy of the model on each machine and distributes the training dataset between the worker machines. This way, each worker gets a different subset of the training dataset referred to as a batch. When the computation of each batch has finished, the results (calculated gradients) from each worker need to be exchanged somehow. Hybrid parallelism combines the model and data parallel techniques. In this method both the model and data are partitioned and distributed to different machines within a distributed cluster.

Distributed Deep Learning frameworks usually implement the gradient exchange step with a Parameter Server (PS) or the All-reduce collective method. In the PS method, a single machine, the *parameter server*, is responsible for periodically gathering the gradients computed by the other machines, updating the parameters needed for the next iterations of the training, and broadcasting the most up-to-date parameters to the workers. One downside of this method is that the other workers do not contribute much toward updating the parameters globally. Hence, the *parameter server* may suffer from overutilization. The All-reduce collective method, on the other hand, involves all the workers exchanging gradients and calculating the updated parameters distributedly.

2.4.1 Horovod

Horovod [60] is an open-source and free distributed Deep Learning training framework for TensorFlow, PyTorch, Keras, and MXNet used by most of the researchers. This framework developed by Uber makes the distributed Deep Learning fast and easy to use and delivers high performance and scalability for different DL models. Using the Horovod framework, a single-GPU training can be scaled to train across many GPUs in parallel with minimum modifications in the training script. Uber's Horovod exploits the MPI model and its concepts such as size, rank, local rank, allreduce, allgather and, broadcast to make the training distributed. The developers in Uber utilized the MPI model in that it delivers high performance and requires fewer code changes than other solutions, such as the parameter server model. Horovod also offers a unique feature called *Tensor Fusion*. *Tensors* are data buffers that the reduction operations are performed on them. The *Tensor Fusion* feature enables communication and computation overlap. Using this feature, as the tensors are computed on each machine, the data will be gathered into the *Fusion Buffer*. Then, when the *Fusion Buffer* is full, or the desired time-out threshold, set by environment variables, is met, the MPI_Allreduce collective is performed. This method, the batch allreduce operation, is especially beneficial when the number of small message allreduce operations is significant. Horovod can also be used with other collective communications libraries such as NCCL [5, 11], Gloo, MLSL, and DDL. In addition, it can be executed both on CPU and GPU platforms. In this study, we concentrate on using Horovod on GPU platforms with MPI collective communication library. Figure 2.8 presents the complete software stack for distributed DL training using Horovod.

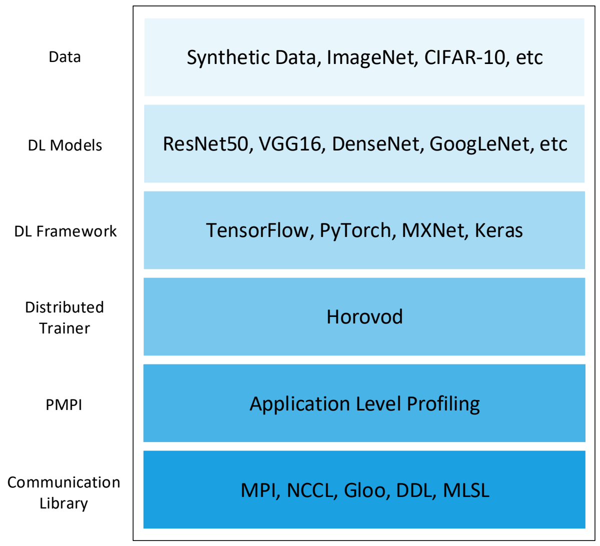

Data
Synthetic Data, ImageNet, CIFAR-10, etc
DL Models
ResNet50, VGG16, DenseNet, GoogLeNet, etc
DL Framework
TensorFlow, PyTorch, MXNet, Keras
Distributed Trainer
Horovod
PMPI
Application Level Profiling
Communication Library
MPI, NCCL, Gloo, DDL, MLSL

Figure 2.8: Software stack for Horovod DL training

Chapter 3

Communication Characterization of a Distributed Deep Learning Framework (HOROVOD)

Parallel applications involve several processes exchanging messages. The communication characterization of these MPI processes is essential in that it can be crucial to the performance of the parallel applications running on clusters. This chapter investigates characteristics of collective communications in Horovod as one of the most famous distributed Deep Learning frameworks. We first present some related works in the literature. Then, we introduce the experimental setup, including hardware/software platforms used in our experimentations. We study the frequency and run-time contribution of different collective operations in Horovod. Then, we introduce the most critical collective and further investigate the frequency of message ranges/sizes, their contribution to the collective's overall run-time, and its Process Arrival Pattern.

It is noteworthy that the collective communication behavior of the Horovod exhibited similar characteristics independent of the platforms we used in our experiments. Therefore, the information presented in this chapter will provide the HPC users, programmers, and Deep Learning scientists with a better understanding of Horovod and opportunities to improve its performance.

3.1 Related Work

Awan et al. [12] provide a performance analysis of of a relatively new HPC application area, distributed Deep Learning, on different CPU (Intel Skylake, AMD EPYC, and IBM POWER9) and GPU (Volta V100) platforms with variuos underlying high-performance interconnects (InfiniBand, Omni-Path, PCIe Gen3, and NVLink). The authors use the Horovod distributed DL framework with TensorFlow for their study and use different DL models such as ResNet-50/101 and Inception-v3/4. They also present some communication characterization results for MPI/NCCL allreduce collective operation such as message size, number of calls for each message size, and the latency for each call. Their study shows that the allreduce suffers from severe performance inconsistency for non-power-of-two message sizes for both CPU and GPU trainings on all interconnects when *Tensor Fusion* is enabled. Ben-Nun and Hoefler [13] provide a comprehensive survey on DNNs and techniques for accelerating their training. They first describe the theoretical aspect of DL models and then introduce approaches for their parallelization. Next, the authors model and analyze the different types of concurrency in DNNs such as Data Parallelism, Model Parallelism, Pipelining, and Hybrid Parallelism. They also discuss different techniques for distributed training of the Deep Learning models such as Parameter Server (PS) and Decentralized methods. The study concludes with some potential directions for optimized parallelism in Deep Learning.

It should be noted that the current literature misses a systematic communication characterization study of DL frameworks. Furthermore, to the best of our knowledge, our work in this chapter is the first study investigating the Process Arrival Pattern of the MPI collectives in such applications.

3.2 Experimental Results and Analysis

3.2.1 Experimental Setup

Our studies were conducted on three clusters. The configuration of these clusters along with the software platform are described below.

Beluga GPU Cluster

Beluga is a general-purpose cluster at Compute Canada, hosted at the Ecole De Technologie Superieure University in Montreal. This cluster consists of 172 nodes, each node having two Intel Gold 6148 Skylake for a total of 40 cores running at 2.4 GHz and 186 GB of memory. Each node also contains four NVIDIA Tesla V100 SXM2 16 GB GPUs connected via NVLink. The Network Interconnect is Mellanox Infiniband EDR with the maximum bandwidth of 100 Gb/s.

Cedar CPU and GPU Clusters

Cedar is a heterogeneous cluster at Compute Canada, located at the Simon Fraser University in Vancouver. Cedar has 192 nodes, each having two Intel Silver 4216 Cascade Lake for a total of 186 GB of memory and 32 CPU cores, running at 2.1 GHz. Each node also has four NVIDIA Tesla V100 32G HBM2 GPUs connected via NVLink. An Intel Omni-Path interconnect with the maximum bandwidth of 100 Gb/s connects together all the nodes of this cluster.

Helios GPU Cluster

Helios is a general-purpose supercomputer at Compute Canada, hosted at the Laval University. This cluster is equipped with eight nodes, each consisting of eight K80 GPUs and two Intel Xeon Ivy Bridge E5-2697 v2 for a total of 256 GB of memory and 24 CPU cores running at 2.7 GHz.

Software Platform and Data Collection

For all the platforms we used the CUDA-aware implementation of MVAPICH2-2.3 with CUDA 10.0.130. We compiled Horovod-0.18.0 to use TensorFlow-1.13.0. For all the tests we utilized the synthetic benchmarks within Horovod which provide the throughput of the image classification task by the number of images processed per second (Images/Sec). These benchmarks remove the overhead of I/O by using synthetic data, leading to more accurate measurements of MPI communication time in our study. The benchmark runs for 10 iterations and uses 10 batches of size 32 as default. As the default configuration suggests for all the tests we assigned one process per GPU and distributed the available CPU cores evenly among processes. We developed a custom MPI profiler to study the characteristics of MPI collective communications, especially their process arrival patterns. Our profiler uses the PMPI interface [3] to obtain the data from the application layer such as the collective operation, the message size, count, type, process arrival times and the time spent in each MPI collective. The times were measured using the *MPI_Wtime*. Also, for accurately measuring the times on different processes on different machines, we called the *MPI_Barrier* after the *MPI_Init* routine and normalized all the processes' reported times by their exit time of the MPI_Barrier. Doing so, we could synchronize the clocks globally between the processes of the cluster, with an inaccuracy equal to the latency of sending a few small messages, which is based on the premise that all the processes exit the barrier operation at the same time. The profiler stores all the gathered data in the memory during the execution of the application, and gets the processes to print them into log files for post-processing purposes, just after the *MPI_Finalize* is called.

3.2.2 Distribution of Collective Communication Calls

In this section, the distribution of MPI collective communication calls per process in Horovod is presented on three different platforms. In Figure 3.1 it can be seen that, on the Beluga cluster, the allreduce operation accounts for the majority of the collective calls in the Horovod by 84%, 93%, and 79% for ResNet50, VGG16, and DenseNet DL models, respectively. The second important collective operation in terms of frequency is the broadcast operation which makes up 11%, 4%, and 17% for ResNet50, VGG16, and DenseNet DL models, respectively. Allgather, Gather, and Gatherv are another collective operations used by Horovod which only accounts for up to 5% of the collective calls for the aforementioned DL models. Since the frequency of collectives in Horovod is similar for different DL models, we provide the results only for the ResNet50 model for the remaining platforms.

Figure 3.2 shows that for the ResNet50 model, the allreduce operation accounts for 75% and 96% of the number of collective calls on Cedar and Helios, respectively. Similar to the results from Beluga, broadcast is the second important operation which makes up 17% and 3% of the collectives, and allgather, gather, and gatherv operations account for the remaining 8% and 1% of the collective calls on Cedar and Helios clusters, respectively.

We investigated the source code of Horovod and observed that the allgather operations have been used to collect values of sparse tensors. Broadcast operations have been utilized to enforce consistent initialization of the model along with the distribution of training/test dataset on all workers. Allreduce operations, on the other hand, are responsible for averaging the gradients among all workers and distributing the results back to them at the end of each iteration. Since the number of iterations for training a DL model with a dataset is usually a large number in the charcterization results we observe that the allreduce operations are the dominant collective used in Horovod. It should be noted that Horovod is strong in terms of scaling delivering up to 76% scalability for the ResNet50 model on 256 GPUs on Cedar.

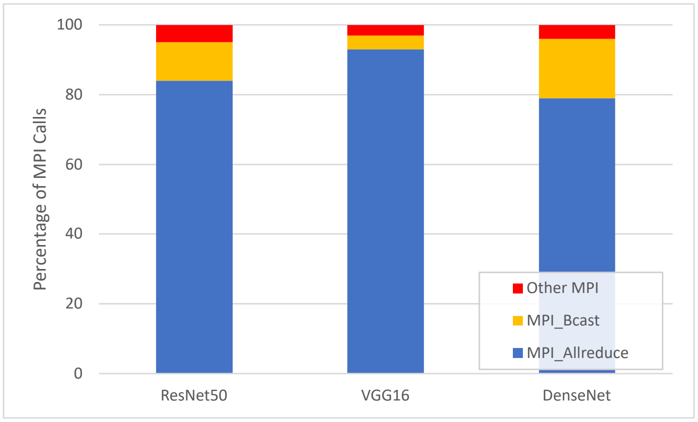


Figure 3.1: Distribution of MPI collective calls for Horovod + TensorFlow with different DL models using 64 GPUs evenly distributed among 16 nodes on Beluga

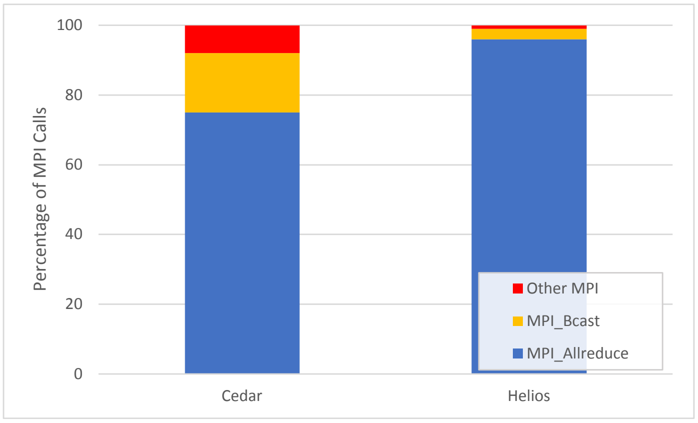


Figure 3.2: Distribution of MPI collective calls for Horovod + TensorFlow for ResNet50 on Cedar (256 GPUs, 64 nodes) and Helios (16 GPUs, single node) clusters

3.2.3 Run-time Contribution

In this sub-section, the contribution of the MPI collectives to the communication run-time on three different platforms for Horovod is investigated . Figure 3.3 shows that the allreduce operation dominates the communication run-time of Horovod by 95%, 96%, and 97% for ResNet50, VGG16, and DenseNet DL models, respectively, on the Beluga cluster. The broadcast operation, although important in terms of number of calls, does not play a key role in the run-time and only accounts for up to 5% of the communication run-time. Allgather, gather, and gatherv collective operations have the least effect on the run-time and only make up less than 1% of the communication run-time. Like the previous sub-section, since the contribution of different collective operations to the communication run-time for different DL models is similar, we only provide the results for the ResNet50 model for the remaining platforms.

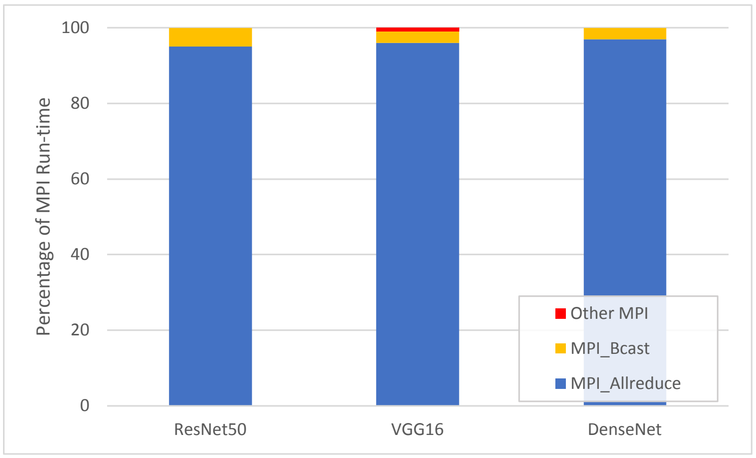


Figure 3.3: Contribution of MPI collective calls to the communication run-time for Horovod + TensorFlow with different DL models using 64 GPUs evenly distributed among 16 nodes on Beluga

In Figure 3.4 it can be seen that for the ResNet50 model, the allreduce operation accounts for 96% and 98% of the communication run-time on Cedar and Helios platforms, respectively. Similar to the results from Beluga, broadcast is the second important collective with contributing up to 4% to the communication run-time. Similarly, allgather, gather, and gatherv operations make up less than 1% of the communication run-time on both Cedar and Helios clusters.

We have observed that the allreduce operation is the most significant collective operation used by Horovod both in terms of its frequency and communication run-time contributions. Hence, in the following sub-sections we further study the characterization of this collective.

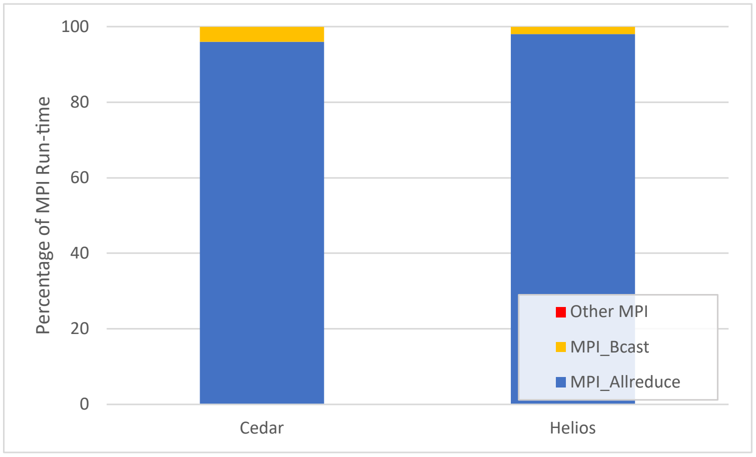


Figure 3.4: Contribution of MPI collective calls to the communication run-time for Horovod + TensorFlow for ResNet50 on Cedar (256 GPUs, 64 nodes) and Helios (16 GPUs, single node) clusters

3.2.4 MPI_Allreduce Characterization

We start by studying the distribution of allreduce calls among different message sizes. Then, we present their contribution to the MPI_Allreduce communication run-time. In the end, we investigate the Process Arrival Pattern of the allreduce calls for different message sizes.

Distribution of Allreduce Calls among Different Message Sizes

Our profiling results show that Horovod uses allreduce with messages in the 4B-64MB range for all of the DL models and on all the clusters we used for this study. We split this message range into three sub-ranges of small, medium, and large message sizes for better understanding the impact of different message sizes on the allreduce communication time and calls. The small, medium, and large message sub-ranges would be 0-1KB, 1KB-1MB, and 1MB-64MB, respectively.

In Figure 3.5 it can be seen that, on the Beluga cluster, the allreduce calls with small message sizes account for the majority of all the allreduce calls by 93%, 88%, and 96% for ResNet50, VGG16, and DenseNet DL models, respectively. Medium message sizes are the least important allreduce calls in that they only account for up to 4% of the allreduce calls for VGG16, and less than 1% for the ReNet50, and DenseNet models. The second important message size sub-range in terms of number of calls is the large message allreduce which makes up 7%, 8%, and 4% of all the allreduce calls in Horovod for ResNet50, VGG16, and DenseNet DL models, respectively.

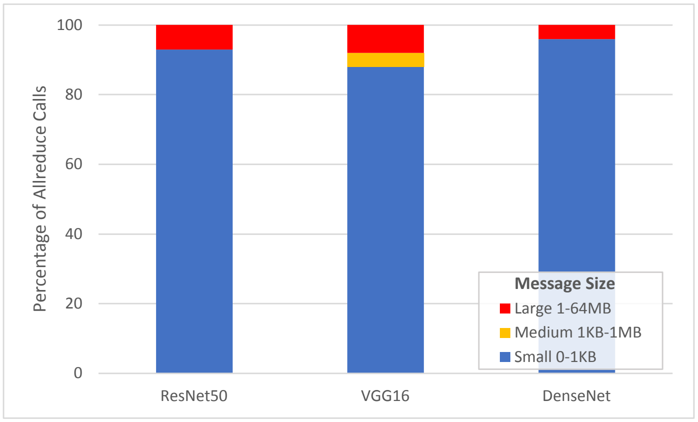


Figure 3.5: Distribution of Allreduce collective calls among different message size sub-ranges for Horovod + TensorFlow with different DL models using 64 GPUs evenly distributed among 16 nodes on Beluga

Since the distribution of allreduce calls among different message sizes for different DL models is similar, for the remaining platforms, we provide the results only for the ResNet50 model.

Figure 3.6 shows that for the ResNet50 model, the small message allreduce operation accounts for 89% and 95% of all the Horovod allreduce calls on Cedar and Helios clusters, respectively. Similar to the results from Beluga, medium and large message allreduce calls are of a less importance in terms of their frequency on Cedar and Helios. On Cedar they only account for 1% and 10% of all allreduce calls, respectively. On Helios, 3% and 2% of all allreduce calls are dedicated to medium and large messages, respectively.

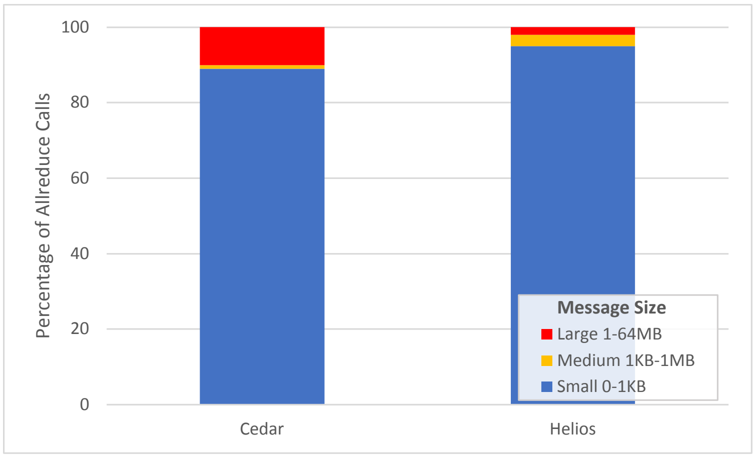


Figure 3.6: Distribution of Allreduce collective calls among different message size sub-ranges for Horovod + TensorFlow for ResNet50 on Cedar (256 GPUs, 64 nodes) and Helios (16 GPUs, single node) clusters

The investigation of the source code of Horovod along with analyzing the characterization results suggest that Horovod uses allreduce operations for two purposes. Eight-byte messages have been used as part of the implementation of the *Tensor Fusion* feature. These allreduce calls are used for synchronization between the workers in order to determine which tensors are ready to be reduced and which workers are involved in the reduction operation. The large message allreduce operations, however, are used to average dense tensors and disperse them to all workers. This explains the massive amount of small message allreduce calls and the notable contribution of the large message allreduce operations to the application run-time.

Contribution of Different Message Sizes to the Allreduce Communication Time

Here, we present the contribution of different message sizes to the allreduce communication time. Figure 3.7 shows that, on Beluga, the allreduce calls with small message sizes dominate the allreduce communication time by 78% and 92% for ResNet50 and DenseNet DL models, respectively. Furthermore, for these two models, the second important message size sub-range is the large message allreduce, making up 22% and 8% of the allreduce communication time in the previously given order. Unlike ResNet50 and DenseNet DL models, for the VGG16 model the large message sizes dominate the allreduce communication time by 63% while the small message sizes account for 36% of application's allreduce time. On the other hand, allreduce with medium sized messages are the least important in that they contribute less than 1% to the allreduce run-time for all the DL models we studied.

Similarly, Figure 3.8 suggests that for ResNet50, the small message allreduce accounts for 65% and 80% of the allreduce time on Cedar and Helios platforms, respectively. The allreduce with large messages have the next significant impact on the allreduce time by making up 35% and 19% on Cedar and Helios clusters, respectively. Again, the medium message allreduce calls do not have any notable contribution to the communication time.

Our investigation into the allreduce operation revealed that small and large messages are responsible for all the allreduce communication time in Horovod. In addition, it was showed that for ResNet50 and DenseNet DL models, small message allreduce outweighs its large massage counterpart, while for the VGG16 large message allreduce has a more significant impact on the Horovod run-time.

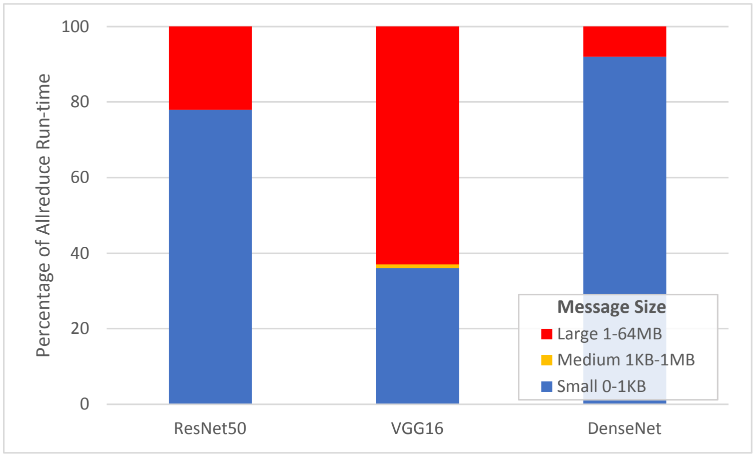


Figure 3.7: Contribution of different message sizes to the application's Allreduce communication time for Horovod + TensorFlow with different DL models using 64 GPUs evenly distributed among 16 nodes on Beluga

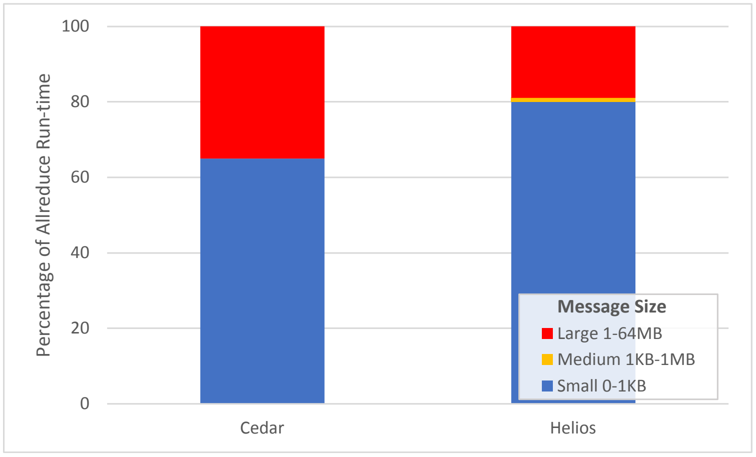


Figure 3.8: Contribution of different message sizes to the application's Allreduce communication time for Horovod + TensorFlow for ResNet50 on Cedar (256 GPUs, 64 nodes) and Helios (16 GPUs, single node) clusters

Process Arrival Pattern

In this section, we investigate the PAP of allreduce collectives. We first study the cluster-wide and then we present the node-wide PAP metrics.

1. Cluster-wide Study

In this part, we measure the PAP parameters by considering all the processes on all nodes separately. Table 3.1 presents the worst case and average case imbalance factors, averaged among the invocations, for allreduce operations used by Horovod for the training of ResNet50, VGG16, and DenseNet Deep Learning models with the synthetic data on Beluga. The first row of results shows the value of aforementioned parameters for all the allreduce operations, while the next rows present the same information for small, medium, and large messages. As it can be seen, both the average worst-case and average-case imbalance factors are relatively very large numbers which suggests that the process arrival pattern is noticeably imbalanced for all the DL models studied. In addition, it can be noted that this imbalanced PAP is significantly larger for the allreduce operations with small message sizes. In other words, the PAP for the allreduce operations with medium/large messages are generally less imbalanced than those with small messages. This is due to the fact that for larger messages the one-way point-to-point communication latency (α) is higher; on the other hand, the maximum/average imbalance time is quite similar for different message sizes which leads to a smaller imbalance factors for collective operations with large message sizes. Similar conclusion can be made by the data presented in Table 3.2 which demonstrates the average of worst/average imbalance factors for the ResNet50 on Cedar and Helios clusters.

Figure 3.9 to Figure 3.11 depict the maximum and average imbalance factors for the

Table 3.1: The average *worst-case* (Ω/α) and *average-case* ($\bar{\Delta}/\alpha$) imbalance factors for Allreduce collective operation of Horovod + TensorFlow with different DL models using 64 GPUs evenly distributed among 16 nodes on Beluga

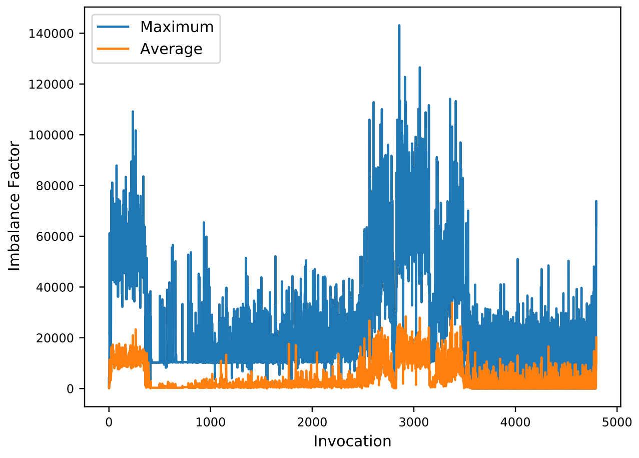
Message Range	Imbalance Factor					
	ResNet50		VGG16		DenseNet	
	Worst	Average	Worst	Average	Worst	Average
All Message Sizes	25328.46	3949.91	14459.75	2303.45	43159.49	8058.54
Small Messages	27175.09	4237.89	15994.99	2523.75	44962.60	8395.21
Medium Messages	153.19	18.04	8614.97	1871.28	266.36	47.12
Large Messages	1.91	0.29	0.88	0.15	5.22	1.11

Table 3.2: The average *worst-case* (Ω/α) and *average-case* ($\bar{\Delta}/\alpha$) imbalance factors for Allreduce collective operation of Horovod + TensorFlow with ResNet50 DL model on Cedar (256 GPUs evenly distributed among 64 nodes) and Helios (16 GPUs on a single node)

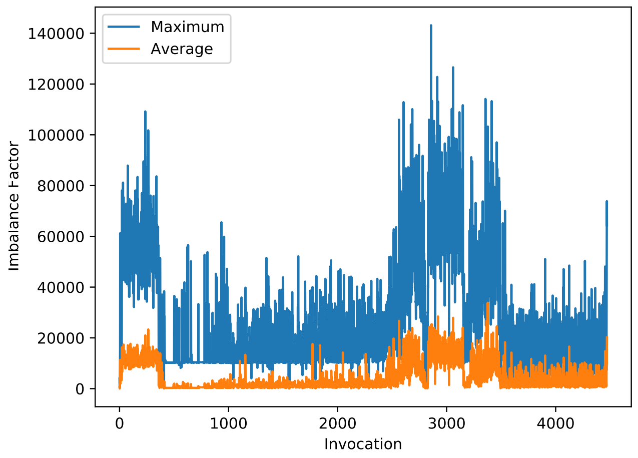
Message Range	Imbalance Factor			
ResNet50	Cedar		Helios	
	Worst	Average	Worst	Average
All Message Sizes	33631.77	5516.57	14003.95	2611.91
Small Messages	37767.87	6194.69	14716.65	2744.46
Medium Messages	327.21	94.41	394.02	85.83
Large Messages	5.85	1.07	6.49	1.51

invocations of allreduce in Horovod for ResNet50, VGG16, and DenseNet DL models, respectively, on Beluga. Figure 3.12 and Figure 3.13 present the same information for the ResNet50 model on Cedar and Helios platforms. Each of the above figures shows the imbalance factors for all the message sizes along with the small, medium, large messages. For the ResNet50 model on Beluga, since there were not any allreduce calls in the medium message sub-range, we did not include it in the corresponding figure.

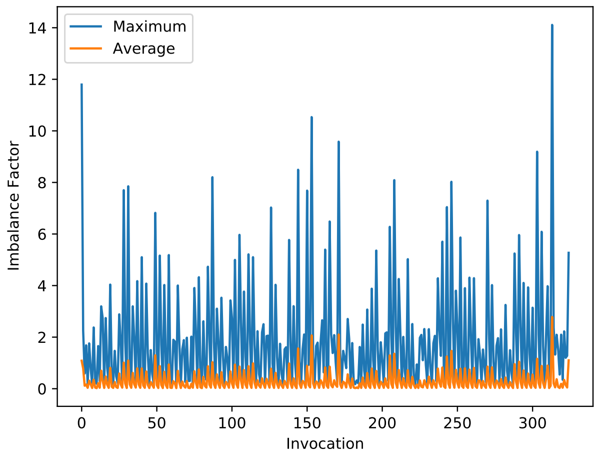
The figures mentioned above reveal that the PAP characteristics of Horovod remain similar on different platforms. Specifically, we notice that for allreduce operations with



(a) All message sizes



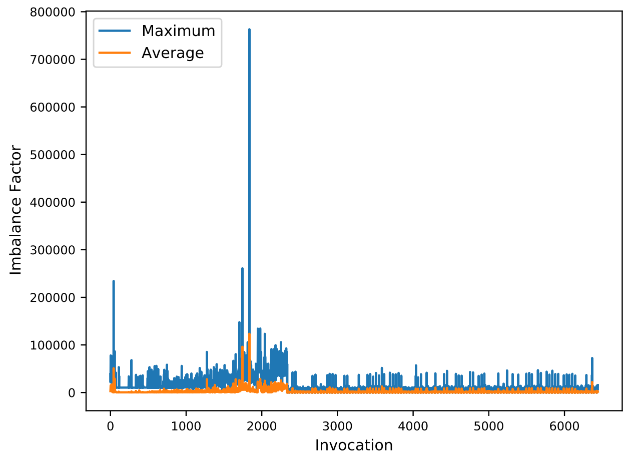
(b) Small message sizes



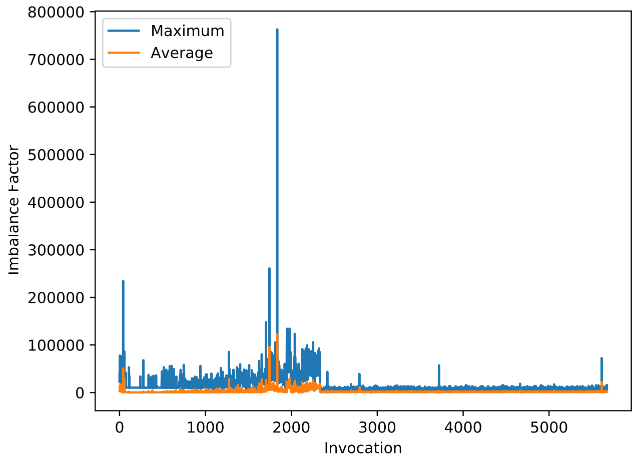
(c) Large message sizes

Figure 3.9: The maximum/average imbalance factors for invocations of Allreduce made by Horovod for ResNet50 using 64 GPUs evenly distributed among 16 nodes on Beluga

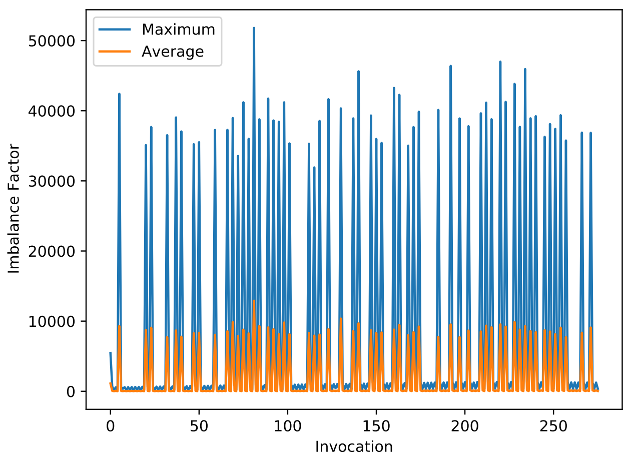
small-sized messages, almost all of the calls have a very large worst/average imbalance factors. However, for medium/large allreduce calls, most of the invocations have very small worst/average imbalance factors, and only a few have slightly larger imbalance factors. Also, we observe that the allreduce PAPs on Cedar are generally more imbalanced than the



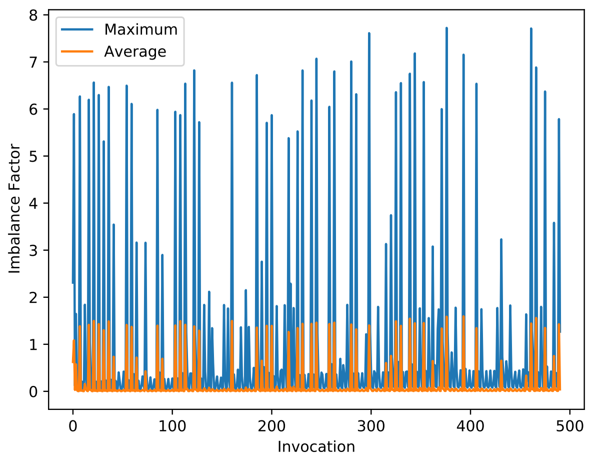
(a) All message sizes



(b) Small message sizes



(c) Medium message sizes

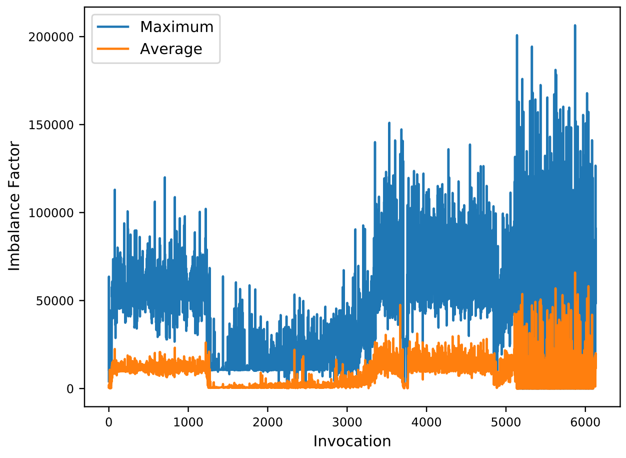


(d) Large message sizes

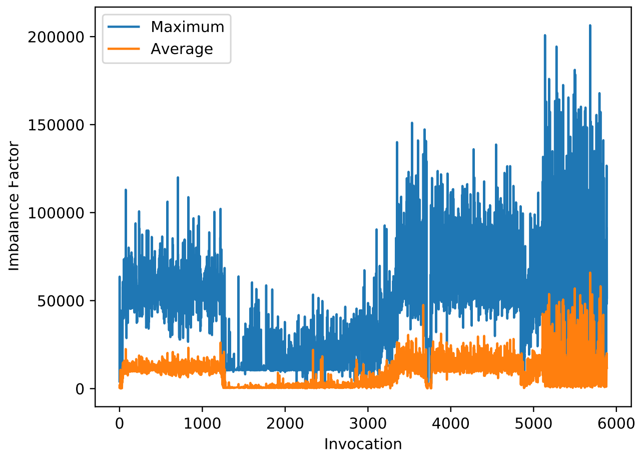
Figure 3.10: The maximum/average imbalance factors for invocations of Allreduce made by Horovod for VGG16 using 64 GPUs evenly distributed among 16 nodes on Beluga

two other platforms. This is most likely due to the larger number of nodes/processes used in our tests on Cedar that translates to larger delays between the arrival of the processes.

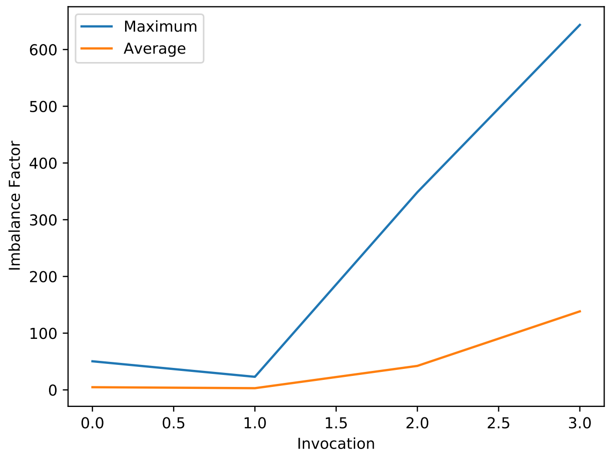
It should be noted that for each setup, the majority of the invocations have similar maximum/average imbalance factors, while there are only a few periods of spikes that occur



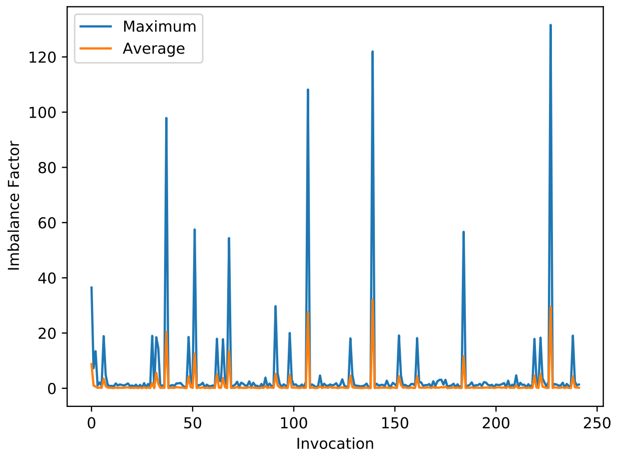
(a) All message sizes



(b) Small message sizes



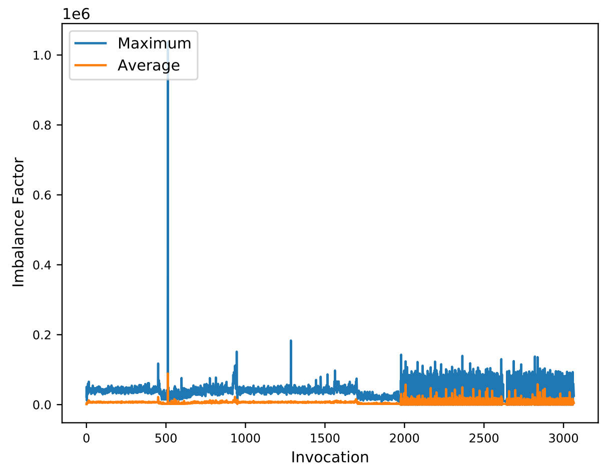
(c) Medium message sizes



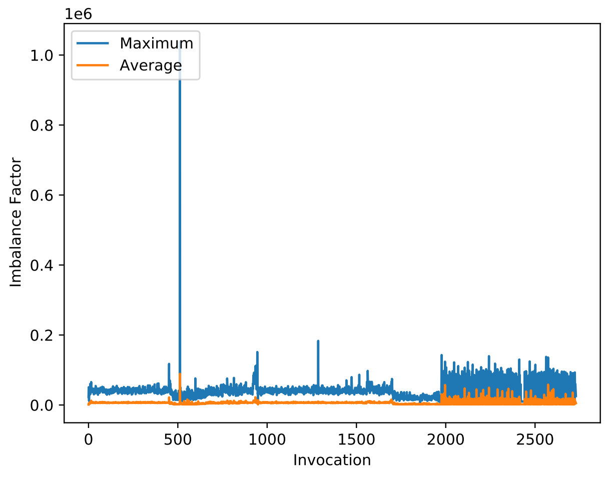
(d) Large message sizes

Figure 3.11: The maximum/average imbalance factors for invocations of Allreduce made by Horovod for DenseNet using 64 GPUs evenly distributed among 16 nodes on Beluga

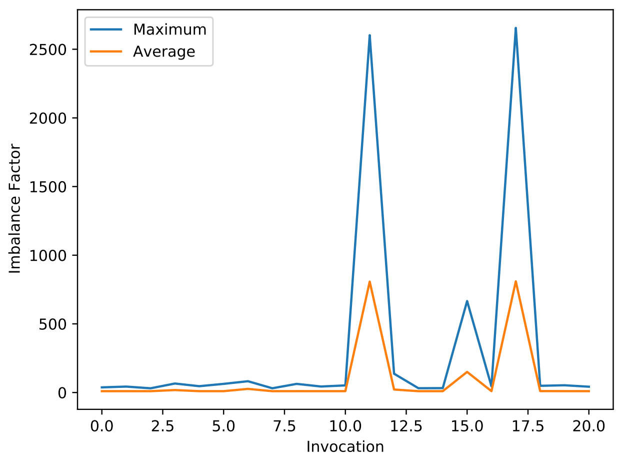
from time to time. In addition, although the imbalance pattern for each of the invocations seems random, a phased behavior can be noted in the process arrival patterns. In other words, the process arrival patterns tend to remain in a roughly steady range for a long span



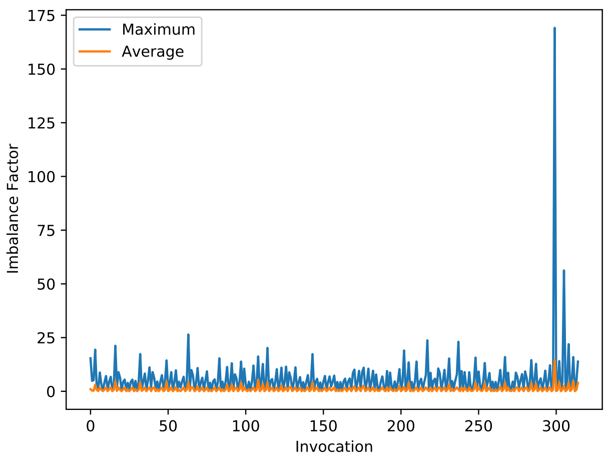
(a) All message sizes



(b) Small message sizes



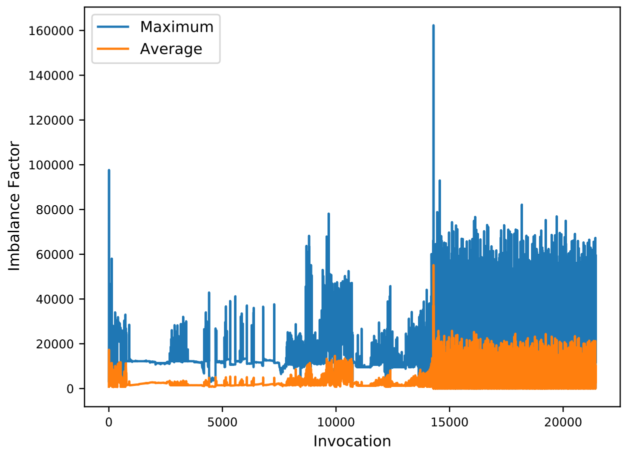
(c) Medium message sizes



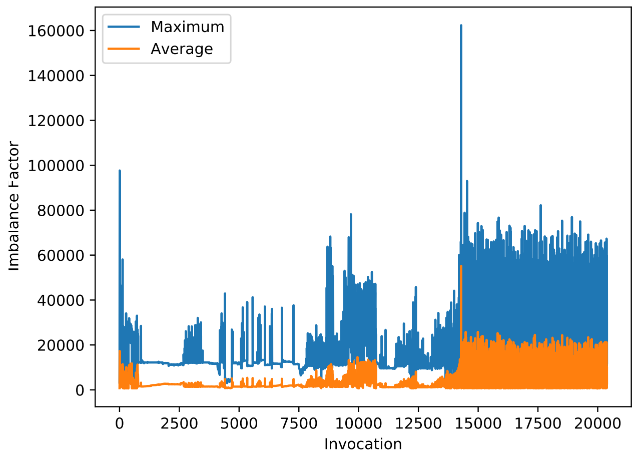
(d) Large message sizes

Figure 3.12: The maximum/average imbalance factors for invocations of Allreduce made by Horovod for ResNet50 using 256 GPUs evenly distributed among 64 nodes on Cedar

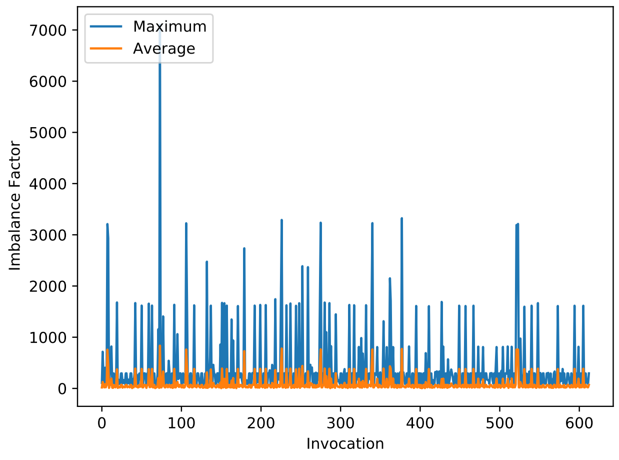
of time before they fluctuate drastically. The aforementioned insights suggest that PAP-ware algorithms could be designed that practically predict the PAP at each invocation of the operation by only considering the history of the arrival pattern of the processes.



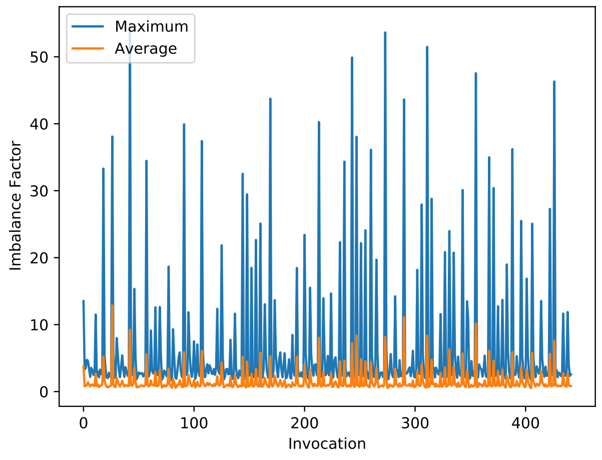
(a) All message sizes



(b) Small message sizes



(c) Medium message sizes



(d) Large message sizes

Figure 3.13: The maximum/average imbalance factors for invocations of Allreduce made by Horovod for ResNet50 using 16 GPUs on a single node on Helios

2. Node-wide Study

In this part, we extract the PAP parameters by calculating the worst/average imbalance factors for the processes residing on each node, without taking into account the processes on the other nodes, and then computing the average of the results gathered from each of the nodes. Table 3.3 and Table 3.4 show that the average of worst/average imbalance factors on each node are significantly lower than the cluster-wide worst and average imbalance factors. In other words, processes on the same node arrive at the collective calls with considerably less delay with respect to each other, compared to all the processes on the cluster, most likely because processes on the same node suffer from similar architecture/software-related features resulting in the imbalance in the arrival time of processes.

Table 3.3: The average node-wide *worst-case* and *average-case* imbalance factors for Allreduce collective operations of Horovod + TensorFlow with different DL models using 64 GPUs evenly distributed among 16 nodes on Beluga

Message Range	Imbalance Factor					
	ResNet50		VGG16		DenseNet	
	Worst	Average	Worst	Average	Worst	Average
All Message Sizes	4360.88	1654.11	2506.73	912.72	9533.577	3594.40
Small Messages	4678.81	1774.70	2773.58	1010.36	9931.80	3744.55
Medium Messages	23.40	8.35	1479.01	528.84	83.99	30.75
Large Messages	0.45	0.15	0.13	0.04	2.29	0.83

Table 3.4: The average node-wide *worst-case* and *average-case* imbalance factors for Allreduce collective operations of Horovod + TensorFlow with ResNet50 DL model using 256 GPUs evenly distributed among 64 nodes on Cedar

Message Range	Imbalance Factor	
	Worst	Average
All Message Sizes	7747.23	2848.16
Small Messages	8698.83	3197.90
Medium Messages	220.12	92.99
Large Messages	1.84	0.77

3.2.5 Summary

In this chapter, we studied the characterization of the collective communications in Horovod. We observed that MPI_Allreduce is the dominant collective in terms of frequency and contribution to the application run-time. Investigations on allreduce operations showed that small and large messages are responsible for the majority of the communication time and hence play a pivotal role in the application's run-time. The process arrival study on the allreduce revealed that small message allreduce collectives suffer from significant imbalanced PAPs. In addition, the PAP for large message allreduce is sufficiently imbalanced to harm the performance of Horovod. In the following chapters, we propose node-wide PAP-aware allreduce algorithms capable of delivering high-performance in imbalanced workloads for small and large messages. Then, in Chapter 6 we investigate a cluster-wide allreduce algorithm for large messages that undermines the negative impacts of imbalanced PAPs on the performance.

Chapter 4

Efficient Intra-node PAP-aware MPI_Allreduce for Small Messages

In this chapter we investigate a process arrival pattern aware MPI_Allreduce algorithm for small messages capable of delivering high performance under imbalanced PAPs. We first discuss the motivation for our proposal. Next, we present the related works in the literature. Then, we propose our idea and explain our design along with some details regarding the implementation of our algorithm. At the end, we evaluate the performance of our process arrival pattern aware algorithm against two high-performance algorithms used in MVAPICH.

4.1 Motivation

As the communication characterization of Horovod in Chapter 3 showed, the process arrival time of the MPI processes can be significantly imbalanced in deep learning workloads, especially for small message MPI_Allreduce collectives. This observation directed us to the following research question:

- How can we design a process arrival pattern aware algorithm for MPI_Allreduce with small message sizes capable of delivering high-performance under different imbalanced PAPs.

We first introduce the native MVAPICH algorithm for small message MPI_Allreduce and then take up the aforementioned challenge and propose a PAP-aware adaptive design for MPI_Allreduce with small message sizes. In MVAPICH, the algorithm of interest for intra-node MPI_Allreduce operation with small message sizes (from 1B-1KB) is a two-step shared-memory algorithm, as follows: In the first phase of this algorithm, each leader process (which is the process with the intra-node rank of zero) waits for all the processes on its node to arrive at the collective call. Once all the processes arrive and copy their data into the shared memory, the leader process reduces the data within the shared memory. In the second phase, the leader process sends the reduced data to all processes on the node via a shared memory broadcast operation.

- **Phase 1**: Intra-node shared-memory reduce by the leader process of the node
- **Phase 2**: Intra-node shared-memory broadcast

One major problem with the aforementioned algorithm is that in the first phase it does not take into account the PAP of the processes. In fact, it performs poorly when some processes enter the MPI_Allreduce operation well after the other processes. This is because, the leader process will not start the intra-node reduce operation until all the processes have already entered the collective operation. So, in this case, any progress in the reduction operation is hindered by the late processes. We addressed this issue by having the leader process poll on the arrival of processes. This way, the leader can reduce the data of the processes as soon as they arrive. In other words, the last process will not block the reduce operation anymore. However, the proposed approach does not provide an opportunity for enhancement if the leader process arrives last. This led us to consider PATs in our design, and choose the leader process dynamically at runtime. This way, we let the early arriving processes contribute to the progression of the reduce operation without the need to wait for all the processes to arrive.

4.2 Related Work

Faraj et al. [21] studied the process arrival pattern characteristics of MPI applications such as FT, LAMMPS, NBODY, and NTUBE on two HPC platforms. The authors introduced some metrics to measure the imbalance in the PATs, such as average and worst-case imbalance factors described in Section 2.3. They showed that the differences between the PATs at a collective operation are usually significant, even for applications with perfectly balanced workloads. Therefore, the imbalance in the process arrival pattern cannot be controlled by making the workload balanced at the application level. They also presented that the process arrival pattern has a notable effect on the performance of different collective communication algorithms and hence HPC applications. Finally, they proposed and evaluated a potential solution for the MPI_Alltoall routine, which achieves high performance with different PAPs. This proposal involves using an automatically tuned collective communication framework that was first presented by Fagg et al. in [20]. The authors developed a dynamic adaptive framework, STAR-MPI [23], which contains a set of eight alltoall algorithms. During the application execution for each invocation of the alltoall, one of these eight algorithms is used and then its performance is measured. This procedure will be executed for all the available algorithms for a number of times. Finally, based on the measured performance an algorithm is selected as the best performing algorithm and will be used thereafter. This way, the authors could achieve better performance than the native MPI implementations for different applications on different environments. This method, however, is only beneficial for applications that run for a large number of iterations. Otherwise, the overhead of measuring the performance of different algorithms for the alltoall routine at the runtime results in a poorer overall performance. In addition, the authors assume that for a certain application on a specific environment the imbalance in the arrival time of the processes for every invocation of the collective routine exhibits the same behavior whereas the random nature of the PAPs contradicts this assumption. Furthermore, the authors do not really propose any PAP-aware algorithm that in any way uses the delays between the arrival times of the processes toward the progression of the desired collective communication.

Patarasuk and Yaun [46] investigated the performance of different MPI_Bcast algorithms such as the flat tree, binomial tree, and linear tree algorithms under different process arrival patterns and showed that they could not achieve high performance for most of the PAPs. Therefore, they proposed two PAP-aware algorithms for the broadcast operation with large message sizes, one for each of the blocking and non-blocking models. In the aforementioned PAP-aware algorithms, the root sends the data to the processes as they enter the collective routine. However, if multiple processes arrive at the collective call simultaneously, the root initiates a sub-group broadcast among the newly-arrived processes. The root then assigns a process within the sub-group to forward the data to the rest of the sub-group processes. This way, serialization at the root could be avoided whenever some processes enter the broadcast operation at the same time. To inform the root of the operation of the early-arriving processes, they used additional control messages. Since the proposed algorithms are meant for broadcasting large messages, the overhead of sending/receiving small control messages is assumed to be negligible. The experiments were performed on two different platforms, one with InfiniBand and the other with an Ethernet interconnection network, to evaluate the performance of the proposed algorithms. It was shown that the proposed PAP-aware broadcast algorithms could achieve high performance for large message sizes across different process arrival patterns. The authors in this study only aim for the broadcast operation and do not provide any PAP-aware algorithms for more communicationally-intensive collectives such as reduce and MPI_Allreduce operations. In addition, they do not propose any PAP-aware broadcast algorithm for small message sizes. Furthermore, the proposed algorithm is not built on hierarchical algorithms. It, therefore, does not take advantage of the fast intra-node shared memory available on modern systems with hierarchical architectures to reduce the communication latency. Instead, they use the MPI point-to-point primitives as the means to communicate control messages as well as data.

Qian and Afsahi [54, 55] proposed RDMA-based PAP-aware algorithms for MPI_Allgather and MPI_Alltoall routines for different message sizes on InfiniBand clusters. The PAP-aware alltoall and allgather designs in this work are based on the direct algorithm proposed in [53] where each process directly communicates with all the other processes within the communicator. Instead of sending/receiving additional control messages, the authors used the RDMA control registers for notifying each process of the arrival of other processes. By using this notification mechanism, each process could send its data to the already-arrived processes without incurring any extra communication overhead. They also extended their work for having a better performance for small messages by making their design also shared-memory aware. The authors then compared the performance of their proposed designs against the native MVAPICH algorithm and observed an average speedup of 1.44 for the FT, RADIX, and N-BODY applications. Although the algorithms proposed in this work achieve good performance in the presence of imbalance, they are aimed for the systems supporting Infiniband RDMA. Otherwise, they would require alternative synchronization/-control mechanisms.

Parsons and Pai [45] used a microbenchmark to study the process imbalance in perfectly balanced workloads. They monitored the performance counters, such as hardware interrupts, cache misses, and stall cycles for each process while executing the microbenchmark and showed that there is not any direct correlation between an event and the imbalance. Instead, the imbalance is caused by multiple hardware and software components. The authors then propose imbalance-tolerant large-message reduce, broadcast, and alltoall algorithms. The reduce and broadcast algorithms use a hierarchical fashion, where the intra-node communication is performed through shared memory while the inter-node step exploits the binomial tree algorithm. In the imbalance-tolerant algorithms for reduce and broadcast, the leader process on each node is selected based on the arrival pattern at each invocation. This way, the early arriving processes can make progress before the later ones arrive. For the alltoall algorithm, a method called *opportunistic message fragmentation* was used to pre-send the early arriving processes' data. The authors also utilize multiple sender processes on each node to diminish the delays incurred by late-arriving leader processes. The performance evaluation of their algorithm on a Cray XE6 cluster exhibited notable speedups over native MPICH's algorithms. Shared memory is usually used for intra-node communications with small messages because the shared memory space is limited on each node, and it provides faster communication compared to the other means only up to medium messages. Therefore, using the shared memory for large message reduce and broadcast operations may lead to severe performance degradation of the proposed algorithms compared to other non-shared-memory algorithms such as Binomial-Tree and the Reduce-Scatter followed by a Gather (RSG) algorithms, especially for messages larger than 2KB, which has not been presented in their study. Furthermore, in the proposed algorithm for the reduce operation, each process is required to reduce its data into a buffer protected by lock(s) in the shared memory. This method might lead to long congestions in the shared memory because the time it takes for each process to execute the reduce operation before releasing the lock variable is quite long, especially for the aimed message range (large messages) in this work.

4.3 The Proposed PAP-aware Design for Small Message MPI_Allreduce

Our proposed intra-node PAP-aware MPI_Allreduce algorithm for small message sizes consists of two steps (reduce + broadcast), similar to MVAPICH's algorithm explained in 4.1. However, unlike MVAPICH, in our design for the reduce step, at each invocation of the MPI_Allreduce collective operation, we dynamically assign the earliest arriving process of each node as the leader process of that node. This leader process is responsible for the execution of the reduction operation on the data of the processes on the node. Other processes, on the other hand, only need to copy their data into the shared memory upon their arrival and set a flag to make the leader process aware that their data in the shared memory is ready to be reduced. Therefore, the leader process polls on the flags and executes the reduction operation whenever data is ready to be reduced. With our proposed algorithm, there is no need to wait for all the processes to arrive to commence the reduce operation anymore. Furthermore, unlike the MVAPICH's algorithm, which chooses the leader process on each node statically (always the process with the intra-node rank zero), we dynamically select the leader process based on the PAT of the processes. This way, the reduction operation will be started as soon as the first process arrives, and there is no need to wait for the statically chosen leader to arrive and execute the reduction operation anymore. This feature of our algorithm is essential, especially when the static leader is the last arriving process. For the second step (broadcast), we use a shared memory broadcast by the leader process to all the processes on the node. In this step, the leader process simply writes its data into the shared memory, and other processes copy that data from the shared memory into their own address spaces.

Algorithm 4.1 presents the pseudo-code of our proposed design. In order to implement the synchronizations between processes on each node for assigning the earliest process as the leader process of that node, we use two variables called *Leader_Defined_Flag* and *Is_Leader*. *Leader_Defined_Flag* is a shared variable among the processes residing on the same node. This flag is protected by lock/unlock so that the first arriving process can be safely determined to avoid race conditions. *Is_Leader* on the other hand, is a local variable to each process which demonstrates whether the process has been assigned as the leader process responsible for the execution of the reduction operation.

In addition to the variables stated above, we use two other shared buffers, namely, *Data_Buffer* and *Data_Ready_Flags*. This way, the processes could share their data and their availability with the leader process through the shared memory. *Data_Buffer* is the shared buffer where the processes copy their data into the preallocated segments of it so that the leader process can execute the reduction operation on them. *Data_Ready_Flags* is the shared buffer filled with flags, each dedicated to one process. These flags demonstrate whether the data has been successfully copied into the *Data_Buffer* by the corresponding processes.

Algorithm 4.1: PAP-aware MPI_Allreduce for Small Messages

```
Input   : The data residing in send buffers (sendbuf)
Output: The data residing in receive buffers (recvbuf)
Variables:
Leader_Defined_Flag: A shared flag protected by lock/unlock to determine the first
  arrived (the leader) process.
Data_Buffer: A shared buffer populated by processes with their data.
Data_Ready_Flags: A shared buffer consisting of flags, each dedicated to one process,
  demonstrating whether the process's data in the Data_Buffer is ready.
Is_Leader: A local variable determining the leader process.

begin
    // Check if the leader has not been defined yet.
    if Leader_Defined_Flag == 0 then
        if trylock(&mutex) then
            if Leader_Defined_Flag == 0 then
                Leader_Defined_Flag = 1; // Leader is defined now.
                Is_Leader = 1; // The process is assigned as the leader.
            end
            unlock(&mutex); // Release the lock.
        end
    end
    if Is_Leader == 1 then
        // Leader polls on Data_Ready_Flags and does the reductions.
        while Operation_Progression < $(local\_size - 1)$ do
            for $(i = 0; i < local\_size; i++)$ do
                if Data_Ready_Flags[i] == 1 then
                    Operation_Progression++;
                    Data_Ready_Flags[i] = 0 ; // Reset the flag.
                    recvbuf += Data_Buffer[i]; // Reduce the data.
                    if Operation_Progression $== (local\_size - 1)$ then
                        // There is no more data to be reduced.
                        Copy(recvbuf, ..., Result,...); // Write result into shared
                         memory.
                        for $(j = 0; j < local\_size; j++)$ do
                            Completion_flags[j] = 1; // Set the
                             Completion_flags.
                        end
                        // Reset the Is_Leader, Operation_Progression,
                         and Leader_Defined_Flag for the next invocation.
                    end
                end
            end
        end
    else
        // Non-leader processes copy their data into Data_Buffer.
        Copy(sendbuf, ..., Data_Buffer[local_rank],...);
        Data_Ready_Flags[local_rank] = 1; // Set the data copy flag.
        while Completion_flags[local_rank] $== 0$ do
            // Wait for own Completion flag to be set
            .
        end
        Copy(Result, ..., recvbuf,...); // Copy the result into own recvbuf.
        Completion_flags[local_rank] = 0 ; // Reset Completion_flags.
    end
end
```

Once a process enters the collective operation, it first reads the *Leader_Defined_Flag* to check whether the leader process has been defined. If the leader process has not been defined yet, it means that the arrived process is among the earliest processes entering the collective and hence could be assigned as the leader. Therefore, the process tries to lock the lock variable (mutex). Upon acquiring the lock, the process re-validates the *Leader_Defined_Flag* to ensure that no process has been assigned as the leader so far. Next, the process assigns itself as the leader process and informs other processes on the node by setting the *Leader_Defined_Flag* and then releases the mutex, as shown in Line 1 to 6 of Algorithm 4.1.

The processes can now perform the actions assigned to them based on whether they are the leader process or not. The role of the non-leader processes is to copy their data (residing in their *sendbuf*) into the appropriate segment of *Data_Buffer* in the shared memory and inform the leader of its availability by setting its dedicated flag in *Data_Ready_Flags* (Lines 18 to 20). The leader process, on the other hand, is responsible for performing the reduction operation on the data. This process polls on the *Data_Ready_Flags*. Whenever a data is ready in the *Data_Buffer*, the leader reduces it into its own *recvbuf*. This operation continues as long as there is no more data to be reduced. At this point, the result of the intra-node reduce step of the hierarchical MPI_Allreduce operation is available in the *recvbuf* of the leader process of the node (Lines 7 to 13).

The second phase (shared memory broadcast) begins by the leader process copying the result of the reduce operation from its own *recvbuf* into the dedicated area in *Data_Buffer* defined as *Result* and sets the *Completion_flags* for each of the processes to inform them that the collective result is ready in the shared memory (Lines 14 to 17). Then, the leader process resets the appropriate flags, such as *Is_Leader* and *Leader_Defined_Flag* for the next invocation of the collective. The non-leader processes, on the other hand, poll on their dedicated *Completion_flags* and once it is set, they copy the MPI_Allreduce result from shared memory into their own *recvbuf* (Lines 21 to 22). Then, they reset their own *Completion_flags* for the next MPI_Allreduce collective call (Line 23). Figure 4.1 presents an example execution of the proposed PAP-ware MPI_Allreduce algorithm for four processes. It should be mentioned that our design is thread-safe, and hence it can be used in multi-threaded environments. In addition, our algorithm supports single-communicator allreduce collectives; however, it can be modified to support multi-communicator allreduce collectives too.

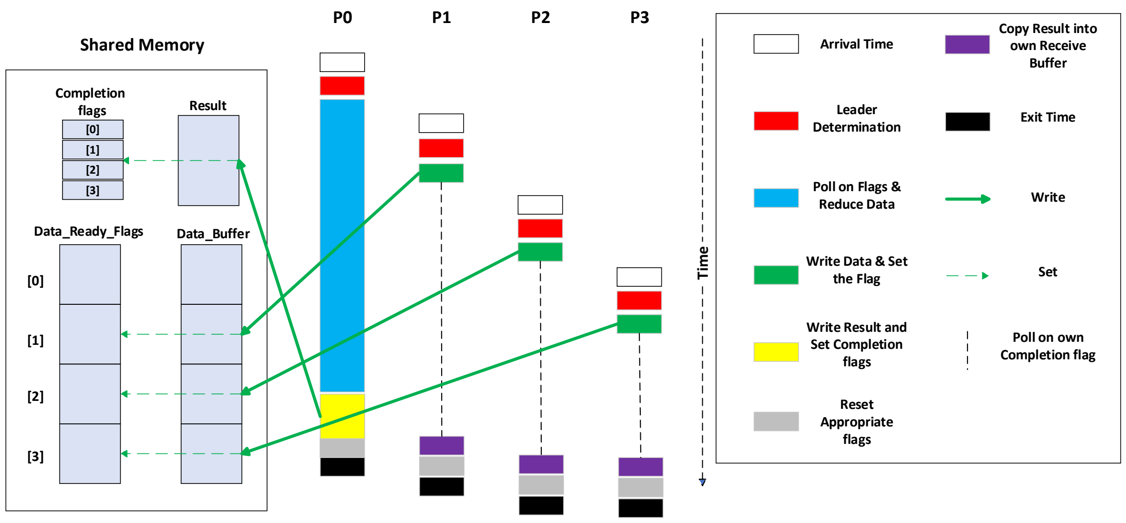


Figure 4.1: Example run of the proposed small message PAP-aware MPI_Allreduce algorithm for four processes

4.4 Evaluation Results and Analysis

In this section, we evaluate the performance of our proposed design for small message MPI_Allreduce collective against two state-of-the-art algorithms under balanced and imbalanced PAPs. For the imbalanced PAPs, we deliver the results under different MIFs. The experimental platform for our tests in this chapter consists of Beluga and Cedar clusters as defined in Section 3.2.1.

4.4.1 Microbenchmark Studies

Our intra-node shared-memory MPI_Allreduce operation is evaluated for message sizes up to 64KB. For larger message sizes, the shared-memory-aware algorithms in MVAPICH as well as our algorithm lose to non-shared-memory-aware algorithms by a large margin. Therefore, we do not present the results past 64KBs. As it was mentioned earlier, the default intra-node MPI_Allreduce algorithm in MVAPICH for messages up to 1KB is the two-step shared-memory MPI_Allreduce algorithm. However, for message sizes from 1KB to 64KB, MVAPICH switches to a flat recursive doubling MPI_Allreduce algorithm, we call the set of the two aforementioned algorithms (shared-memory algorithm up to 1KB and the RD algorithm from 1KB to 64KB) Def-MVAPICH in the rest of this chapter. In order to evaluate the performance of our PAP-aware shared-memory algorithm more fairly, we also compare its performance against an algorithm that uses the same two-step shared-memory algorithm of MVAPICH, but for all messages up to 64KB. We call this algorithm, shmem-MVAPICH algorithm.

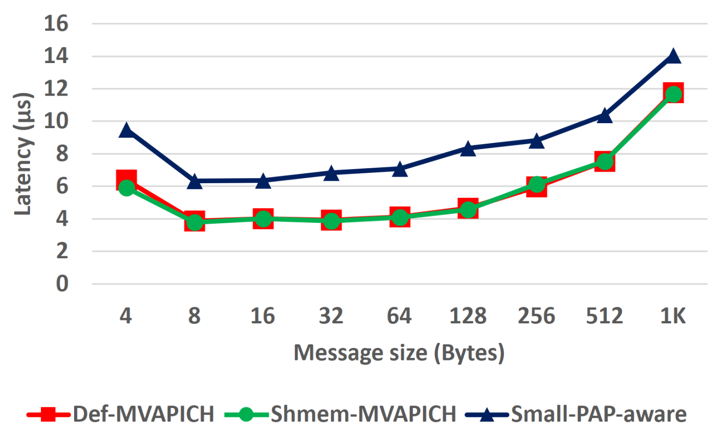
Microbenchmark Results for MPI_Allreduce with Balanced PATs

Here, we present the performance comparison of our PAP-aware algorithm referred to as Small-PAP-aware algorithm against the Def-MVAPICH and shmem-MVAPICH under balanced PATs. We use the OMB suite for this experiment. Listing 4.1 provides the pseudo-code for this balanced OMB microbenchmark which assures that all the processes start the MPI_Allreduce collective call at the same time by calling the MPI_Barrier routine before the MPI_Allreduce call. The OMB microbenchmark runs for 200 iterations for cache warm-up reasons and then runs for 1000 iterations and then reports the results averaged over the number of iterations without considering the warm-up iterations.

```
for (i=0; i<ITERATION; i++) {
  MPI_Barrier(MPI_COMM_WORLD);
  t_start_balanced = MPI_Wtime();
  MPI_Allreduce(...);
  elapsed_time_balanced += MPI_Wtime() - t_start;
}
```

Listing 4.1: OMB pseudo-code for the balanced microbenchmark for the MPI_Allreduce operation

Figure 4.2 and Figure 4.3 present the latency of the three aforementioned algorithms for the balanced microbenchmark for a single node with 32 processes on Beluga and Cedar, respectively. First, it can be seen that the Def-MVAPICH and the shmem-MVAPICH algorithms deliver the same performance up to 1KB messages, while from 1KB to 64KB, the Def-MVAPICH outperforms the shmem-MVAPICH. This is due to the fact that the Def-MVAPICH, like the shmem-MVAPICH algorithm uses the two-step shared-memory MPI_Allreduce algorithm up to 1KB.



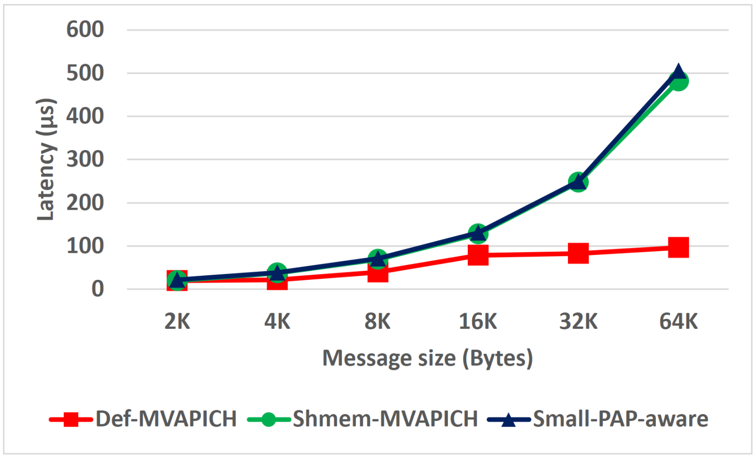
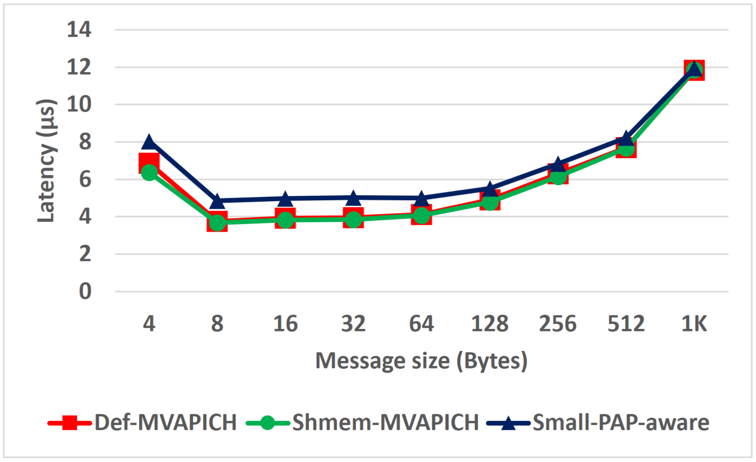


Figure 4.2: Latency comparison of MPI_Allreduce under balanced PATs for the proposed PAP-aware, Def-MVAPICH, and shmem-MVAPICH algorithms with 32 processes on a single node on Beluga

In addition, it can be observed that from 1KB to 64KB, where the Def-MVAPICH uses the flat Recursive Doubling algorithm, it outperforms the two-step shared-memory MPI_Allreduce used by the shmem-MVAPICH algorithm. This suggests that the performance of shared-memory drops significantly as the size of the messages grows after 1KB. Secondly, as shown in the figures, when the PAP is perfectly balanced, the Def-MVAPICH algorithm delivers slightly better performance than the PAP-aware algorithm up to 1KB.



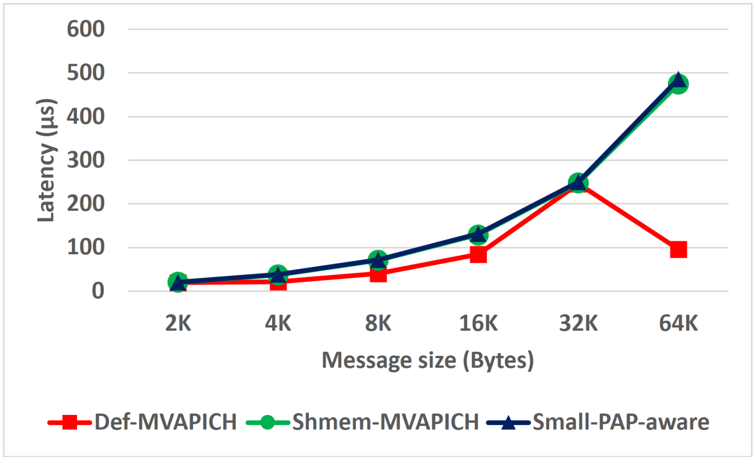


Figure 4.3: Latency comparison of MPI_Allreduce under balanced PATs for the proposed PAP-aware, Def-MVAPICH, and shmem-MVAPICH algorithms with 32 processes on a single node on Cedar

The lower latency of the Def-MVAPICH compared to the PAP-aware algorithm in this message range is because our algorithm uses control messages to detect the first arriving process at each invocation of the collective call. When all the processes arrive at the collective at the same time, this method adds a slight extra overhead to the algorithm's execution time. However, for messages larger than 1KB the performance difference between the two algorithms becomes significant due to the poor performance of shared memory for medium messages. We observe an average of 65% and 25% performance degradation comparing the PAP-aware algorithm and the Def-MVAPICH algorithm for messages smaller than 64KB under balanced PAT, on Beluga and Cedar clusters, respectively. The performance difference between the PAP-aware and the shmem-MVAPICH algorithms, however, is negligible. Also, in Figure 4.3, it can be noted that moving from 32KB to 64KB messages, the latency drops. It is because MVAPICH uses the two-step shared-memory MPI_Allreduce algorithm for 32KB messages, and it switches back to the RD algorithm for larger messages on Cedar, and as it can be observed, the RD algorithm performs much better for 64KB messages.

Microbenchmark Results for MPI_Allreduce with Imbalanced PATs

We compare the performance of our PAP-aware algorithm with the Def-MVAPICH and shmem-MVAPICH under imbalanced PATs. For this purpose, we designed an imbalanced microbenchmark, provided in listing 4.2, which induces a random computation before the collective communication for each process. The upper bound for the random computation in this microbenchmark is determined by the variable Maximum Imbalance Factor.

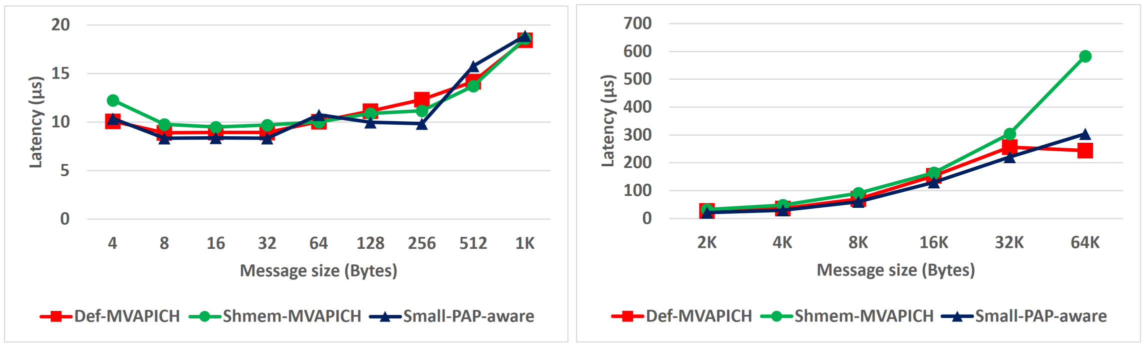
Figure 4.4 and Figure 4.5 present the latency comparison of our proposed PAP-aware algorithm against the Def-MVAPICH and the shmem-MVAPICH algorithms for three imbalanced workloads with MIFs equal to 10, 20, and 50 on Beluga and Cedar, respectively. We have opted for these values for the MIFs in that at MIF of 10 our algorithm starts to outperform the other algorithms, at MIF of 20 it delivers the highest improvement, and at MIF of 50 it shows saturation in the performance improvement. Therefore, these MIFs represent a proper performance evaluation of our proposal under different imbalanced arrival patterns. As it can be seen in both figures, for a small MIF of 10, our PAP-aware algorithm outperforms both the other algorithms almost for all message sizes. At this MIF, the maximum improvements over the best performing algorithm are 20% and 22%, on Beluga and Cedar, respectively.

```
r = MIF * (rand() / RAND_MAX); //MIF would be the upper bound for r.
for (i=0; i<ITERATION; i++) {
  MPI_Barrier(MPI_COMM_WORLD);
  for (k=0; k<r; k++) {
    /* computation equal to the latency of a point-to-point
   communication time for desired message sizes */
  }
  t_start_imbalanced = MPI_Wtime();
  MPI_Allreduce(...);
  elapsed_time_imbalanced += MPI_Wtime() - t_start;
}
```

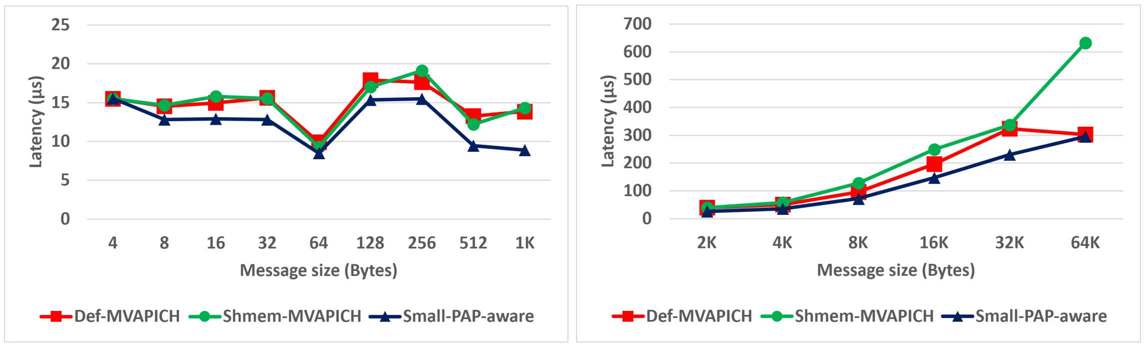
Listing 4.2: Pseudo-code for the imbalanced microbenchmark with controlled random PAPs for the MPI_Allreduce operation

As we increase the MIF to 20, we observe the maximum performance improvement over the Def-MVAPICH, and the shmem-MVAPICH algorithms by up to 36% and 56% on Beluga and Cedar, respectively. We further increase the MIF to 50 and observe that our PAP-aware algorithm outperforms the two other algorithms similar to the MIF of 20. However, the observed improvement in the latency translates to a lower improvement in terms of percentage at MIF of 50 because the delay in the arrival time of the processes is so large that it dominates the latency of the whole collective call. At MIF of 50, we observed the maximum improvements of 21% and 37% over the best performing algorithm on Beluga and Cedar, respectively. We observed the same pattern for the MIFs larger than

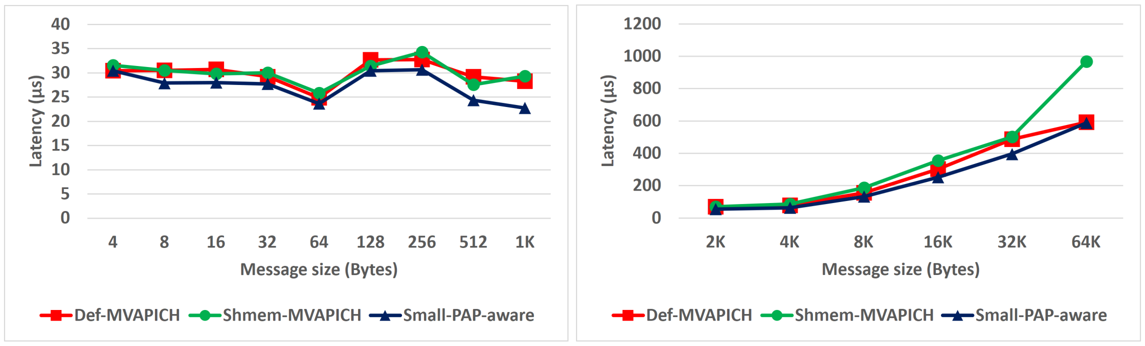
50 where the PAP-aware algorithm was able to outperform the two other algorithms for all the message sizes.



(a) MIF = 10

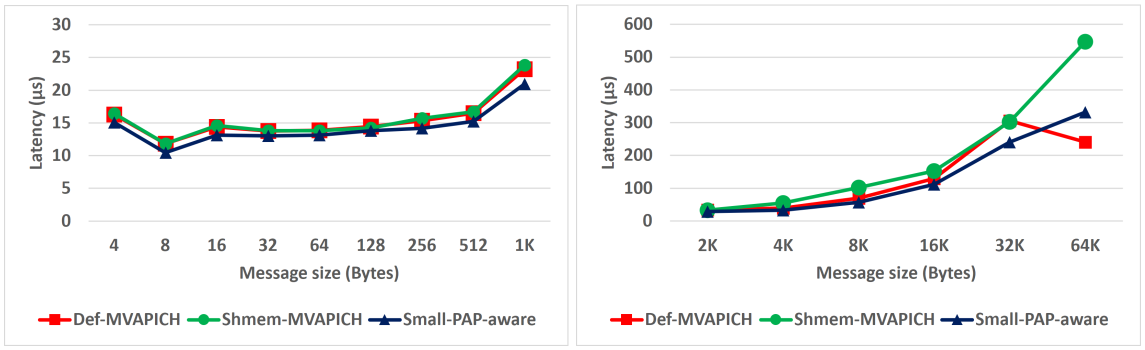


(b) MIF = 20

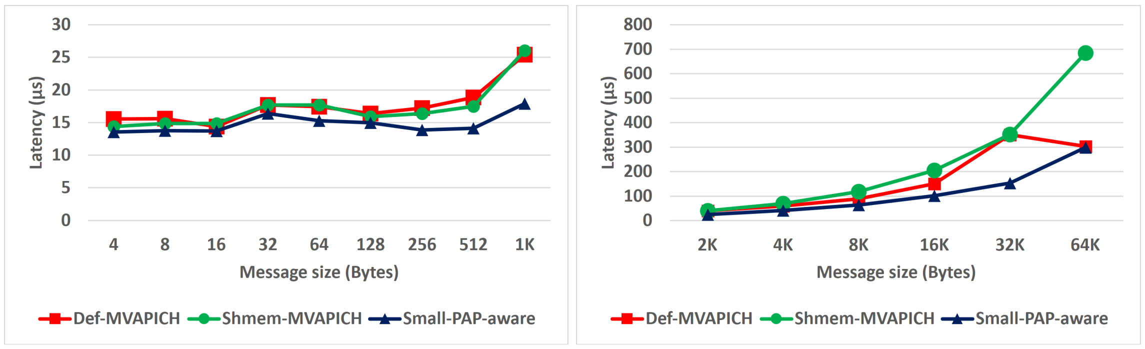


(c) MIF = 50

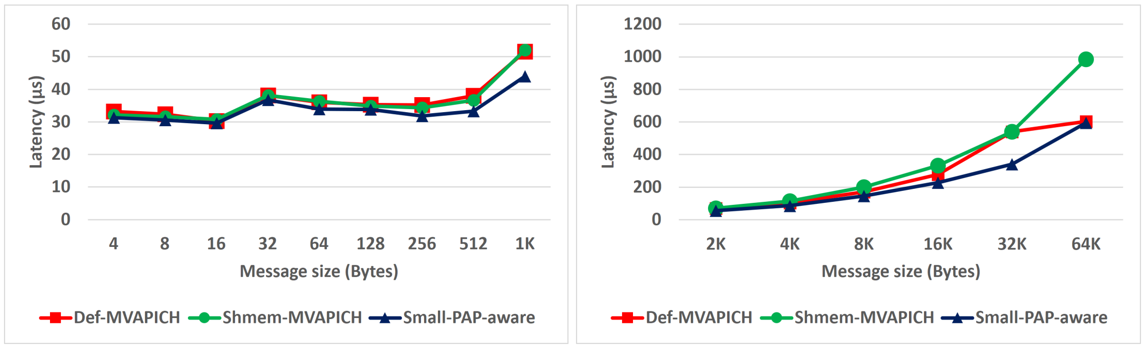
Figure 4.4: Latency comparison of MPI_Allreduce under imbalanced PATs with different MIFs for the proposed PAP-aware, Def-MVAPICH, and shmem-MVAPICH algorithms with 32 processes on a single node on Beluga



(a) MIF = 10



(b) MIF = 20



(c) MIF = 50

Figure 4.5: Latency comparison of MPI_Allreduce under imbalanced PATs with different MIFs for the proposed PAP-aware, Def-MVAPICH, and shmem-MVAPICH algorithms with 32 processes on a single node on Cedar

4.5 Summary

In this chapter, we proposed an intra-node shared-memory PAP-aware algorithm for small message MPI_Allreduce collective. Our design dynamically chooses the leader process at each invocation of the collective routine based on the PAT to help the early arriving processes communicate as much as possible before the later processes arrive to improve the performance of the MPI_Allreduce operation in imbalanced PATs. We evaluated our algorithm against the default algorithm of MVAPICH as well as a shared memory based algorithm native to MVAPICH with balanced and imbalanced microbenchmarks on two different platforms. We observed up to 36% and 56% performance improvement on Beluga and Cedar clusters, respectively. In the next chapter, we investigate a PAP-aware algorithm for MPI_Allreduce operation with large messages.

Chapter 5

Efficient Intra-node PAP-aware MPI_Reduce/Allreduce for Large Messages

5.1 Motivation

In Chapter 4 we proposed a process arrival pattern aware algorithm for the MPI_Allreduce operation with small message sizes. In this chapter, we take up the challenge of designing a PAP-aware algorithm for the MPI_Allreduce collective operation for large messages. The communication characterization of the Horovod application showed that the PAT of the MPI processes for MPI_Allreduce calls with large message sizes could be sufficiently imbalanced to affect the performance negatively in deep learning workloads. In addition, it has been shown in the literature that the typical MPI reduction algorithms, although performing well for balanced workloads, face significant performance issues under imbalanced PAPs [50, 39, 40, 51, 52]. We propose a PAP-aware reduce operation for large messages, which takes the PATs into account in each invocation, to allow the early arriving processes to start the communication. This way, the number of processes waiting for the late-arriving ones will be minimized. Then, using our proposed PAP-aware reduce algorithm followed by an MPI_Bcast operation, we are able to extend our idea to the MPI_Allreduce collective operation. In the following sections, we first review the related studies in the literature. Then, we propose our PAP-aware reduce and allreduce collective operations and explain the details regarding their design. Finally, we present the performance evaluation of our designs against a number of well-known MPI_Reduce/Allreduce algorithms implemented in MVAPICH using balanced and imbalanced microbenchmarks.

5.2 Related Work

Proficz [50] proposed two PAP-aware algorithms for the MPI_Allreduce operation, namely Sorted Linear Tree and Pre-Reduced ring algorithms. The author introduced a background thread for each process responsible for monitoring the progress of the computation phase. The background thread estimates the remaining computation time for its process and communicates this information with other processes on the other nodes. Using the gathered data, each background thread is able to approximate the PAP for itself and other processes. The estimated PAP is then used by the proposed algorithms. One problem with this method is that introducing an extra thread (background thread) would lead to the oversubscription of the processing cores and, consequently, performance penalties. In addition, having the background threads to monitor the progress of the computation phase requires manipulating the source code of applications which might incur performance overheads, especially when the computation phase time is comparable to the collective communication time, this overhead would be significantly large.

The Sorted Linear Tree (SLT) algorithm is based on the well-known linear tree algorithm. Using the estimated PAP, the sorted linear tree algorithm sorts the processes based on their arrival times. Then, the algorithm allows the faster processes to start the communication before the later ones arrive. It is shown in the paper that the SLT algorithm is capable of delivering speedup of up to 1.16 over the standard linear tree algorithm in specific cases. The Pre-Reduced Ring (PRR) algorithm is an extension of the ring algorithm. Similar to the SLT algorithm, this algorithm uses the information regarding the estimated PAP to sort the processes based on their arrival time and assigns new IDs to them. In the PRR algorithm based on the predicted process arrival times, the number of reducing pre-steps to be performed by the faster processes is calculated. Using the method mentioned above, for specific cases of PAP, the speedup of up to 1.14 over the regular ring algorithm could be achieved. The proposed SLT and PRR algorithms delivered 4.2% and 4.0% improvement compared to a typical ring algorithm for training a convolutional neural network (CNN) with the CIFAR-10 [1] dataset.

Marendic et al. [39] studied the performance of reduction algorithms under imbalanced PAPs. Then, they propose two load balancing reduction algorithms, static and dynamic. The static load balancing reduction algorithm depends on a priori knowledge of PATs of all the participating processes and is shown to achieve near-optimal latency. This algorithm is based on the unrealistic assumption that the PATs can be predicted before the call to the collective operation. The dynamic load balancing reduction algorithm, on the other hand, does not require a priori information about the PATs. This algorithm is an extension of the binomial tree algorithm. It assumes atomicity of the reduced data meaning that the data on each process cannot be split and reduced segment by segment. The authors used small control messages to signal the PATs between the involved processes so the early arriving processes can reduce their data and redirect their sub-results to the later arriving ones. One major problem with this work is that the authors do not consider all the possible PAPs in their design. In the proposed algorithm the imbalance in the PAP can only be absorbed if the neighbor processes exhibit similar arrival pattern; otherwise, the communication progression will be hindered or at least will not be optimal. Finally, the performance of the dynamic load balancing algorithm was compared to other algorithms such as binomial tree and all-to-all reduce algorithms. It was shown that for specific PAPs, specifically when there is only a single slow process, the proposed algorithm outperforms the under-study algorithms.

Marendic et al. [40] continued the study of reduction algorithms under imbalanced PAPs and proposed a new PAP-aware reduce algorithm, called Clairvoyant. The proposed algorithm, unlike the algorithms presented in [39], can be applied for both atomic and non-atomic input data. The Clairvoyant algorithm requires pre-knowledge of the entire PATs to construct an optimized reduction schedule and it does not itself include any solution for the PAT estimation. The authors assume that the PAPs exhibit a recurring pattern and therefore predict the PATs statistically using a simple moving average (SMA) approximation. However, it is impossible to predict the PAP accurately for each invocation of the collective operation due to the random nature of it. Furthermore, additional communications are required to exchange the PAT values predicted by the SMA model between the processes that introduces an extra overhead to the algorithm. Finally, this study proposes a pre-computed reduction schedules which remains the same for a certain number of collective invocations, whereas a reduce algorithm changing the communication pattern dynamically based on the PAP for each invocation could be developed as a better solution. The authors compared the performance of their proposed algorithm with some typical reduction algorithms such as the binomial tree, parallel ring, and butterfly algorithms. The results showed that the Clairvoyant algorithm could take advantage of imbalanced PAPs and outperform the experimented algorithms. However, the results provide the performance evaluation of the proposed design only under specific imbalanced PAPs where all the processes except one arrive without any delay. Experiments with random imbalanced PAPs could have evaluated the performance of studied algorithms more accurately.

5.3 The Proposed PAP-aware Designs

5.3.1 The Proposed MPI_Reduce for Large Messages

In this section, we introduce our proposed PAP-aware algorithm for large message reduce operation. In our algorithm, we allow the early arriving processes to start the reduce operation before the later processes arrive and leave the collective call as soon as they make their contribution to the collective communication operation. This way, in the presence of imbalanced PAPs, we minimize the time each process spends in the collective site and therefore, the overall collective communication latency will be decreased.

One principal challenge with all the PAP-aware algorithms is to develop a way to inform every other process of the processes that have already arrived at the collective call. One method for doing this is to exchange point-to-point control messages between the processes, as exploited by authors in [46] for MPI_Bcast collective operation. Using control messages introduces an extra overhead and affects the performance of the algorithm negatively. In order to make the overhead of control messages less significant, we constructed a shared-memory structure between the processes on each node. This way, we could communicate the necessary synchronization signals between the processes faster than the point-to-point message exchange method.

Algorithm 5.1 presents the pseudo-code of our proposed PAP-aware large message reduce algorithm.

Algorithm 5.1: PAP-aware MPI_Reduce for Large Messages

Input : The data residing in send buffers (*sendbuf*)

Output: The data residing in the root's receive buffer (*recvbuf*)

Variables:

***Process_Counter*:** A shared counter protected by lock/unlock to keep the track of the number of arrived processes.

***Sorted_Ranks_Buffer*:** A shared buffer filled with ranks of the processes based on their arrival order.

***Arrival_Rank*:** A local variable for storing the value of *Process_Counter*.

```
begin
1     lock(&mutex); // Acquire the lock.
2         Arrival_Rank = Process_Counter; // Read Process_Counter and
          determine your arrival position.
3         Process_Counter + = 1; // Increment Process_Counter.
4     unlock(&mutex); // Release the shared resource.
5     Sorted_Ranks_Buffer[Arrival_Rank] = MPI_Rank; // Write your MPI_Rank in
      the appropriate segment of Sorted_Ranks_Buffer based on the
      arrival order.
6     if First Process to Arrive then
7         while (Sorted_Ranks_Buffer[Arrival_Rank + 1] = = NULL); // Wait for the
          next process to arrive.
8         Send(... , Sorted_Ranks_Buffer[Arrival_Rank + 1] , ...); // Send your data
          to the newly arrived process.
9     else if Last Process to Arrive then
10        Receive(... , Sorted_Ranks_Buffer[Arrival_Rank - 1] , ...); // Receive the
          Reduced data from the previously arrived process.
11        Reduce the received data with your own data.
12        Send(... , root , ...); // Send the result to the root process.
13    else
14        Receive(... , Sorted_Ranks_Buffer[Arrival_Rank - 1] , ...); // Receive the
          Reduced data from the previously arrived process.
15        Reduce the received data with your own data.
16        while (Sorted_Ranks_Buffer[Arrival_Rank + 1] = = NULL); // Wait for the
          next process to arrive.
17        Send(... , Sorted_Ranks_Buffer[Arrival_Rank + 1] , ...); // Send your data
          to the newly arrived process.
18    end
end
```

In this algorithm, we minimize the data dependency between the processes. Whenever a process arrives, it receives the reduced data from the last process that has already arrived. After performing its part to the reduce operation, the process passes the updated reduced data to the next arriving process and leaves the collective call. This way, in any given imbalanced PAPs and with any number of processes, there would be only one process waiting for the arrival of one another process at each point in time. In order to implement the synchronizations between the processes necessary for the execution of our algorithm, we use two shared-memory variables called *Process_Counter* and *Sorted_Ranks_Buffer*. *Process_Counter* is a counter shared between the processes on the node. This shared variable is protected by lock/unlock so the processes can safely access its value even at the presence of race conditions. *Sorted_Ranks_Buffer*, on the other hand, is a shared buffer containing as many cells as the number of processes on the node. Each cell of this buffer will be dedicated to a process based on the arrival order of processes. For example, the first cell of the *Sorted_Ranks_Buffer* will be assigned to the earliest arriving process while the latest arriving process occupies the last cell of *Sorted_Ranks_Buffer*. Each process writes its *MPI_Rank* in its designated cell. Since *Sorted_Ranks_Buffer* is shared, each process will have access to *Arrival_Rank* (determined by the position of the cell) and *MPI_Rank* (determined by the value of the cell) of all processes on the node.

Once a process arrives at the collective call, it tries to lock the mutex. Upon acquiring the lock, the process reads the *Process_Counter*'s value, recognizes its rank among the already arrived processes, and stores it into a local variable called *Arrival_Rank*, then increments the counter and releases the mutex, as shown in Lines 1 to 4 in Algorithm 5.1. Next, the process writes its *MPI_Rank* in the appropriate cell of *Sorted_Ranks_Buffer*, which its *Arrival_Rank* suggests (Line 5). Now that the process is aware of its arrival rank, it can perform the actions assigned to it based on its arrival time.

For the earliest process, since there is no predecessor process already entered the call, it only needs to wait for the next process to arrive (through the *Sorted_Ranks_Buffer*) to be able to pass its data to the newly arrived process and leave the collective call (Lines 6 to 8). For the processes who arrive between the first and the last processes, the first step is to receive the reduced data from their predecessors. The predecessor process can be recognized through the *Sorted_Ranks_Buffer*. Then, they reduce their own data with the received data and wait for their successor process to arrive. Upon arrival of the next process, they will send the updated data to it. At this point, they can leave the collective since they have accomplished their contribution to the operation (Lines 13 to 17).

When the last process arrives and receives the reduced data from the previous process, it will reduce its own data with the received one. At this moment, the result of the reduce operation is ready in the receive buffer (*recvbuf*) of this process. Finally, the result will be sent to the root process and the operation will be completed (Lines 9 to 12). Figure 5.1 presents an example execution of the proposed PAP-ware reduction algorithm for four processes.

5.3.2 The Proposed MPI_Allreduce for Large Messages

We design a two-step intra-node PAP-aware allreduce algorithm by using our proposed PAP-aware reduce, followed by a broadcast collective operation. For the broadcast operation, we utilize two broadcast algorithms used frequently in MPI implementations. A shared-memory-based algorithm was used for small message sizes up to 64KBs. This algorithm implements the broadcast operation by having each process to copy the data from shared-memory to its own receive buffer, as explained in Chapter 4. For larger message

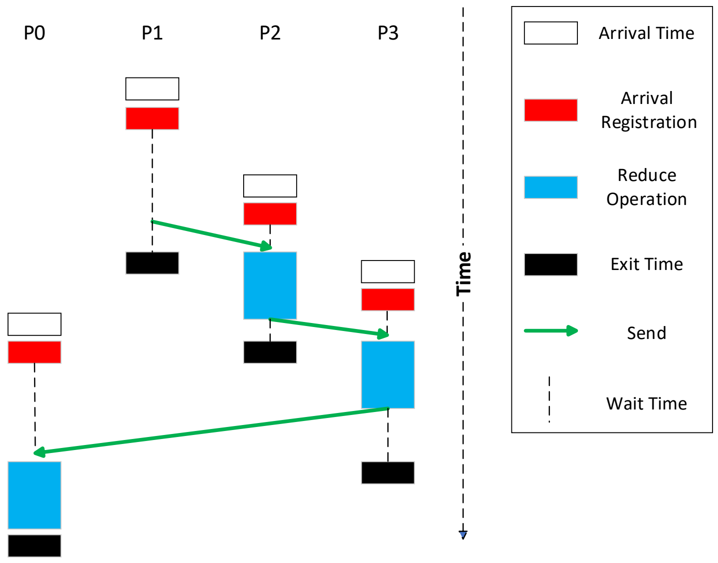


Figure 5.1: Example run of the proposed large message PAP-aware reduce algorithm for four processes

sizes, on the other hand, the commonly used Binomial-Tree broadcast algorithm of MVAPICH was exploited.

5.4 Evaluation Results and Analysis

In this section, we evaluate the performance of our proposed design for large message reduce and MPI_Allreduce collectives against high-performance algorithms used in MVAPICH under balanced and imbalanced PAPs. For the imbalanced PAPs, we provide the results under different MIFs. The experimental platform for our tests in this chapter consists of Cedar and Helios clusters as defined in Section 3.2.1.

5.4.1 Microbenchmark Studies

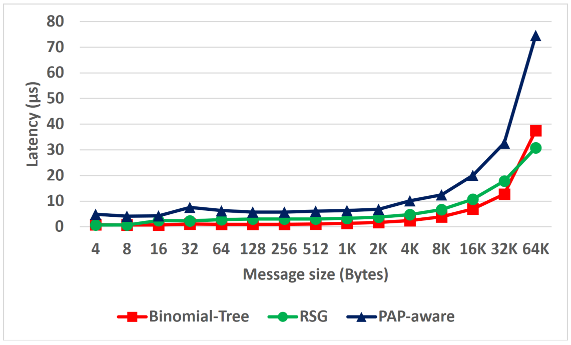
MPI_Reduce

We begin by comparing the performance of our proposed PAP-aware reduce algorithm against Binomial-Tree and Reduce-Scatter followed by a Gather (RSG) algorithms in balanced and imbalanced circumstances. These algorithms are two famous algorithms delivering state-of-the-art performance for the reduce operation in MPI libraries including MVAPICH.

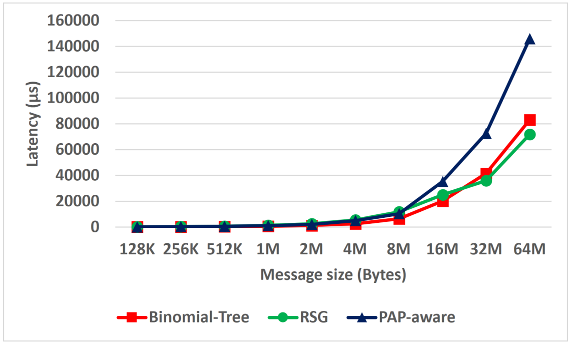
Microbenchmark Results with Balanced PATs

We use a similar microbenchmark as in Listing 4.1 to measure the latency of the algorithms. Figure 5.2 and Figure 5.3 present the performance of the three aforementioned algorithms when the microbenchmark is perfectly balanced on the Cedar cluster with 4 Processes Per Node and the Helios platform with 16 PPN, respectively.

As it can be seen in these figures, the PAP-aware algorithm is not as good as the two other algorithms for most of the message sizes when the microbenchmark is perfectly balanced. When the PAP is balanced, the algorithms with fewer execution steps or with high level of pipelining provide a better performance. Our proposed algorithm takes $n-1$ steps to execute the reduction operation, while the Binomial-Tree algorithm takes only $log_2 n$ steps (n is the number of processes). The fewer number of steps leads to a smaller latency for the Binomial-Tree algorithm. The RSG algorithm, on the other hand, exploits data segmentation and pipelining and therefore, achieves performance by maximizing the bandwidth utilization when the PAP is perfectly balanced.

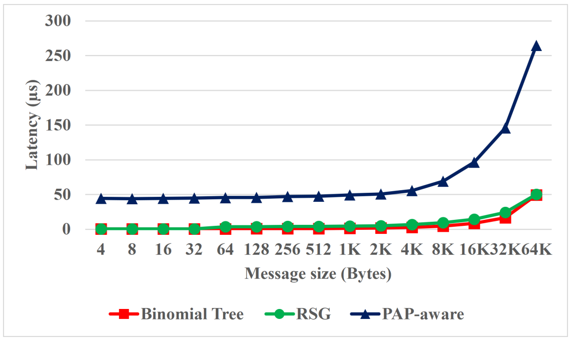


(a) small to medium messages

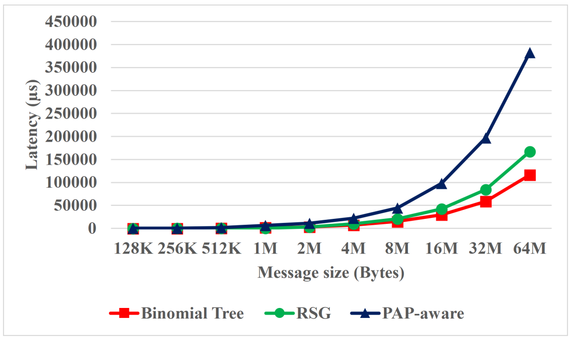


(b) large to very large messages

Figure 5.2: Latency comparison of MPI_Reduce under a balanced microbenchmark for Binomial-Tree, RSG, and PAP-aware algorithms with 4 processes on a single node on Cedar



(a) small to medium messages



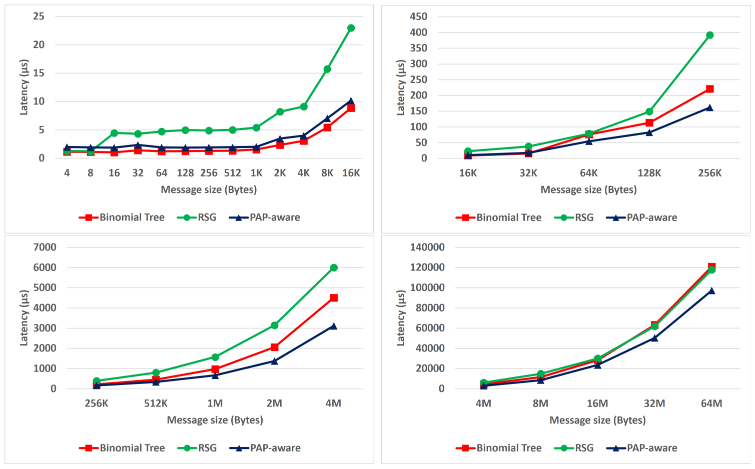
(b) large to very large messages

Figure 5.3: Latency comparison of MPI_Reduce under a balanced microbenchmark for Binomial-Tree, RSG, and PAP-aware algorithms with 16 processes on a single node on Helios

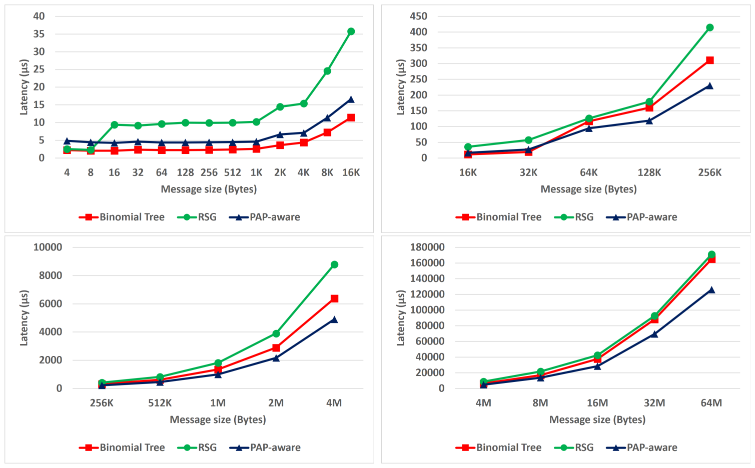
Microbenchmark Results with Imbalanced PATs

Here, we compare the performance of our PAP-aware reduce algorithm with the Binomial-Tree and RSG algorithms under imbalanced workloads. We present the results under three different imbalanced PAPs using a similar imbalanced microbenchmark as in Listing 4.2. For the imbalanced microbenchmarks' figures, we present the studied message range in four plots for each MIF so that the performance of the algorithms can be distinguished more easily.

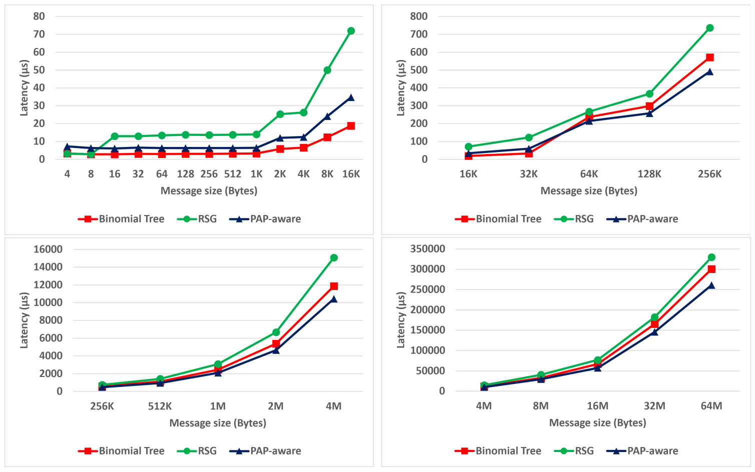
Figure 5.4 and Figure 5.5 present the performance of the PAP-aware, Binomial-Tree, and RSG algorithms for the reduce operation under imbalanced workloads on Cedar with 4 PPN and Helios with 16 PPN, respectively. On Cedar, we have presented the latency of the algorithms for three imbalanced workloads with MIFs equal to 10, 20, and 50. We chose the MIFs in this section for the same reasons explained in Section 4.4.1. As it can be seen in Figure 5.4, for a small MIF of 10, our PAP-aware algorithm outperforms both the other algorithms almost for all larger than 32KB messages. This is the MIF which our algorithm starts to consistently achieve better performance than the two other algorithms on Cedar with 4 processes. At this MIF, we observed up to 30% performance improvement. For smaller MIFs, the PAP-aware algorithm delivers comparable performance with other algorithms. As we increase the MIF to 20, we observe the maximum performance improvement over the Binomial-Tree and RSG algorithms for the messages in the range of 64KB to 64MB. For the messages larger than 64KB, the average improvement over other algorithms at this MIF is 26%, while the maximum improvement of 28% was observed for the message size of 512KB. Increasing the MIF to 50, we observe that for really high delays in the PAP, our PAP-aware algorithm outperforms the two other algorithms for almost all the message sizes between 64KBs and 64MBs by almost the same difference in latency as for MIF of 20. However, the observed improvement in the latency translates to a lower total speedup for this MIF because the delay in the arrival time of the processes is so large that it dominates the latency of the whole collective call. At MIF of 50, for 64KB to 64MB messages, the average improvement over the best algorithm is 14%, and the maximum improvement of 15% was achieved at the message size of 512KB.



(a) MIF = 10

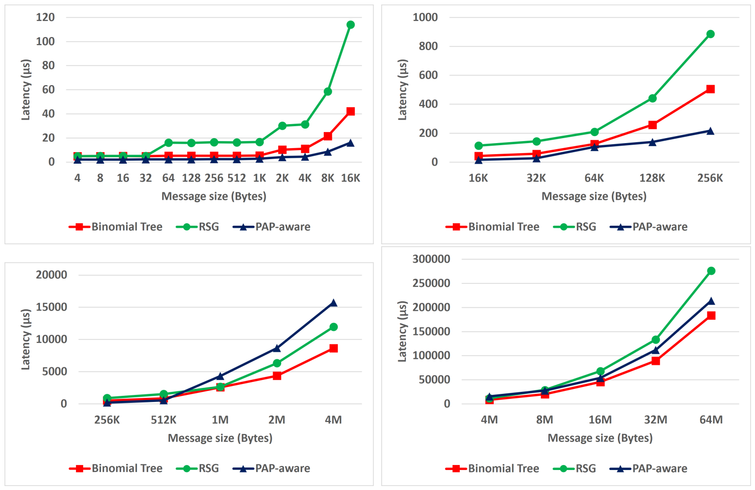


(b) MIF = 20

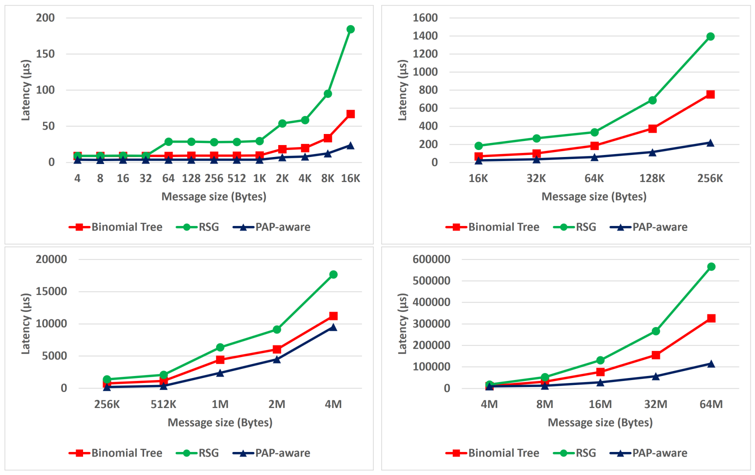


(c) MIF = 50

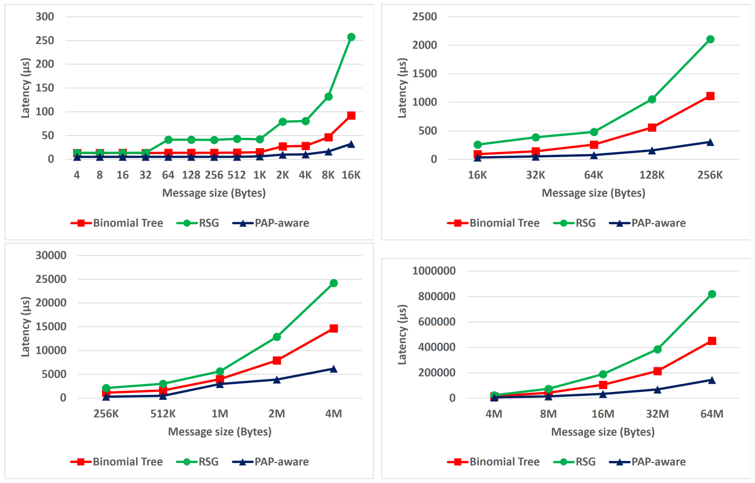
Figure 5.4: Latency comparison of MPI_Reduce under imbalanced microbenchmarks for Binomial-Tree, RSG, and PAP-aware algorithms with 4 processes on a single node on Cedar



(a) MIF = 20



(b) MIF =50



(c) MIF = 75

Figure 5.5: Latency comparison of MPI_Reduce under imbalanced microbenchmarks for Binomial-Tree, RSG, and PAP-aware algorithms with 16 processes on a single node on Helios

On Helios, we have presented the latency of the algorithms for three imbalanced workloads with MIFs equal to 20, 50, and 75. We start with MIF 20 which is larger than the starting MIF tested on Cedar because with 16 processes per node, the number of steps for the reduce operation is larger on Helios, and hence our algorithm starts to outperform the algorithms at higher MIFs.

As it can be seen in Figure 5.5, at MIF of 20, our PAP-aware algorithm outperforms the other algorithms for message sizes up to 512KBs. As we increase the MIF to 50, our PAP-aware algorithm outperforms the other algorithms almost for all message sizes within the 4B to 64MB range. This is the MIF which our algorithm starts to consistently achieve better performance than the two other algorithms on Helios with 16 processes for all the message sizes. At this MIF, the average improvement over the best performing algorithm across all the message sizes is 59%, and the maximum improvement of 70% was observed at the message size of 256KB. We further increase the MIF to 75 and observed the average improvement of 63% across all the messages, while the maximum improvement of 73% was observed for 256KB messages.

When the workload is imbalanced, our PAP-aware algorithm, unlike the two other algorithms, lets the early arriving processes leave the collective as soon as they contribute to the operation. In fact, in our algorithm, the number of processes waiting for the last process to arrive is always one, while for the Binomial-Tree and the RSG algorithms, $log_2 n$ and $n - 1$ processes are involved, respectively. Due to this, the time processes spend in the collective in our design is optimum, leading to the superior performance of our proposed PAP-aware algorithm over Binomial-Tree and RSG algorithms in the presence of imbalanced PAPs. Comparing the results on the two platforms shows that on Helios with 16 PPN the performance improvement is higher compared to the results on Cedar with 4 PPN.

We believe this is also due to the fact that was mentioned earlier. The increase in the number of processes, unlike our PAP-aware algorithm translates to the increase in the number of processes waiting for each other in the Binomial-Tree and RSG algorithms. Therefore, the performance difference between our proposal and the two other algorithms increases as the number of processes grow. In addition, on both platforms, for very large MIFs, we observed the same pattern where our PAP-aware algorithm still outperforms the two other algorithms for these message sizes. However, the total speedup decreases as we increase the MIF.

MPI_Allreduce

In the following, we present the performance evaluation of our proposed PAP-aware MPI_Allreduce algorithm against three other algorithms in balanced and imbalanced circumstances.

Microbenchmark Results with Balanced PATs

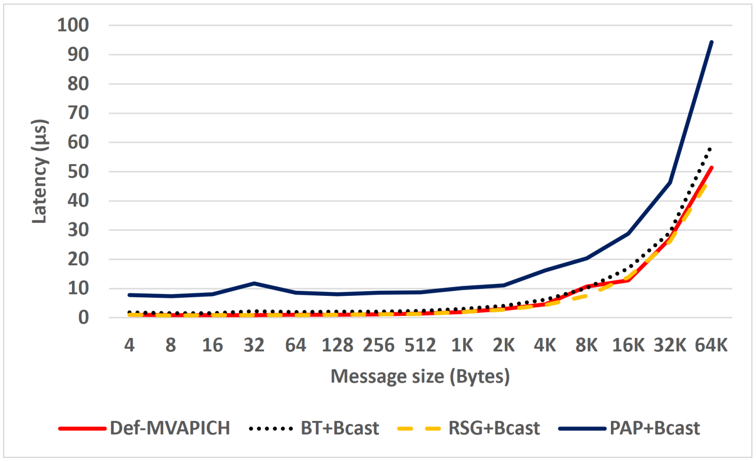
First, we compare the performance of our design with the state-of-the-art algorithms native to MVAPICH using a perfectly balanced microbenchmark provided in Listing 4.1. The default intra-node MPI_Allreduce algorithm for messages larger than 64KBs is the RSA algorithm in MVAPICH. This algorithm delivers the best performance among the other algorithms implemented in MVAPICH according to its tuning table. We call this algorithm Def-MVAPICH in the rest of this chapter. There are another intra-node MPI_Allreduce algorithms native to MVAPICH for large messages that similar to our PAP-aware design for MPI_Allreduce use a two-step (reduce + broadcast) algorithm. The ”BT+Bcast” algorithm implements the MPI_Allreduce operation by exploiting the Binomial-Tree reduce algorithm followed by a broadcast operation. The ”RSG+Bcast” algorithm, on the other hand, utilizes the RSG algorithm for the reduce operation followed by a broadcast operation. All the aforementioned two-step algorithms along with our PAP-aware intra-node MPI_Allreduce design, referred to as as ”PAP+Bcast” algorithm, use the same broadcast operation for the second phase as explained in Section 5.3.2. The phases regarding these algorithms is as follows:

- **Phase 1**: Intra-node Reduce into the leader process of the node
- **Phase 2**: Intra-node broadcast by the leader process

Figure 5.6 presents the performance of the aforementioned algorithms under a balanced workload with four processes on a single node on Cedar. It can be seen that the native algorithm of MVAPICH (Def-MVAPICH) outperforms all the other tested algorithms. In addition, it can be observed that the PAP-aware algorithm is not as good as the other algorithms for most of the message sizes when the microbenchmark is perfectly balanced.

Microbenchmark Results with Imbalanced PATs

Here, we compare the performance of our PAP-aware MPI_Allreduce algorithm with the three previously introduced algorithms under three different imbalanced PAPs using the imbalanced microbenchmark provided in Listing 4.2. In our two-step PAP-aware MPI_Allreduce operation, unlike the PAP-aware reduce operation, we could not release the processes as soon as they contribute to the reduction operation in that all the processes have to wait until they are provided with the result of the collective via a broadcast operation. Therefore, no performance improvement could be gained from the optimal release of processes in the first step of the MPI_Allreduce operation. However, in our algorithm, unlike other algorithms, when the last process arrives, there is only one reduction left to be done to



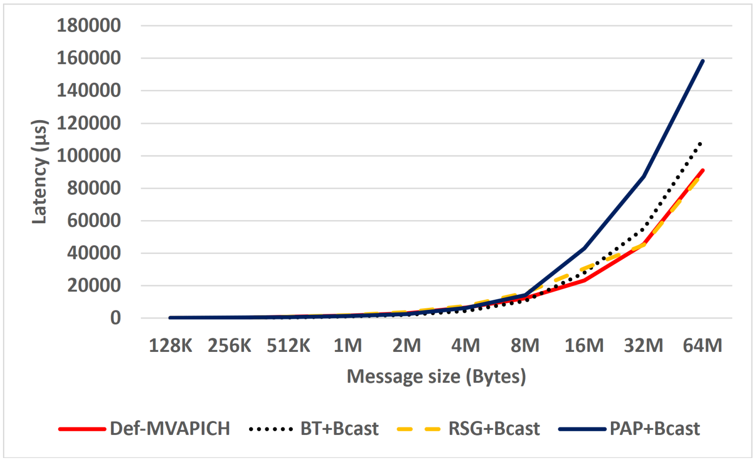
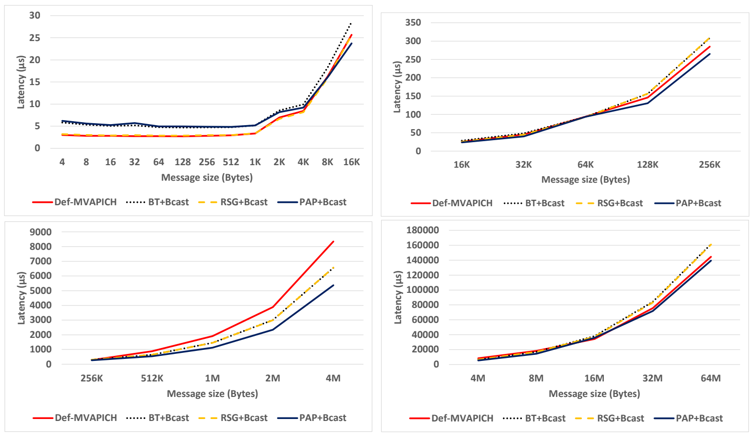
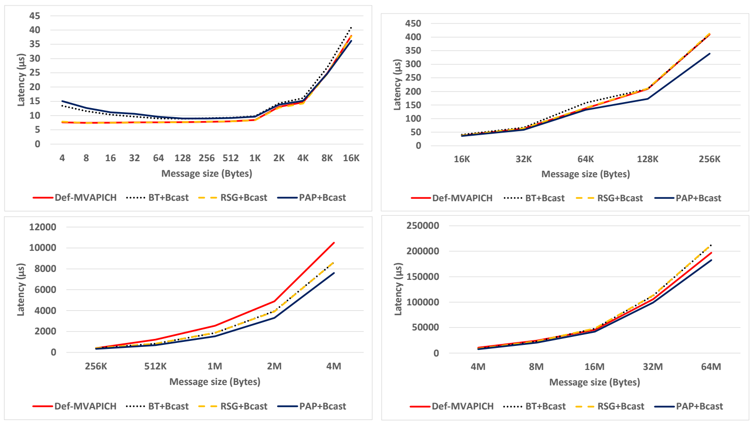


Figure 5.6: Latency comparison of MPI_Allreduce under a balanced microbenchmark for Def-MVAPICH (RSA), "BT+Bcast", "RSG+Bcast", and "PAP+Bcast" algorithms with 4 processes on a single node on Cedar

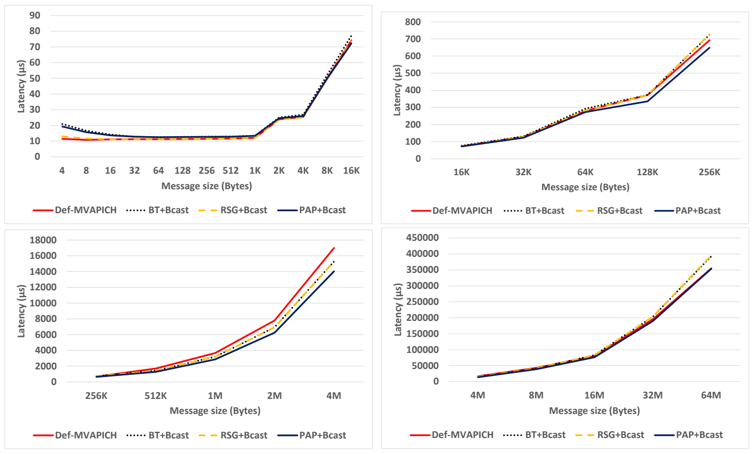
complete the reduce operation. In contrast, the Def-MVAPICH (RSA), "BT+Bcast", and "RSG+Bcast" algorithms have n, $log_2 n$, and n reduction operations left, respectively. Figure 5.7 presents the performance of the PAP-aware algorithm against the studied algorithms for the MPI_Allreduce operation under imbalanced workloads with MIFs equal to 10, 20, and 50 on Cedar with 4 PPN.



(a) MIF = 10



(b) MIF = 20



(c) MIF = 50

Figure 5.7: Latency comparison of MPI_Allreduce under imbalanced microbenchmarks for Def-MVAPICH (RSA), ”BT+Bcast”, ”RSG+Bcast”, and ”PAP+Bcast” algorithms with 4 processes on a single node on Cedar

For a small MIF of 10, our PAP-aware algorithm outperforms all the other algorithms for all message sizes larger than 4KBs. At this MIF, our algorithm starts to achieve better performance than the other algorithms consistently. We observe the average improvement of 18% over the best performing algorithm among the tested algorithms for the message range of 64KB to 64MB. In addition, the maximum improvement of 41% was observed for the message size of 1MB.

As we increase the MIF to 20, we observe the average improvement over the best performing experimented algorithm of 20%, while the maximum improvement of 44% was observed for the message size of 512KB. We further increase the MIF to 50 and observe the average improvement of 11% and the maximum improvement of 25% for the message size of 512KB over the best performing tested algorithm. For significantly larger MIFs, we observe that although our PAP-aware algorithm outperforms all the other experimented algorithms, the total speedup decreases as we increase the MIF.

On Helios with 16 processes, our proposed intra-node PAP-aware MPI_Allreduce operation delivered only comparable performance with respect to the other three algorithms in imbalanced microbenchmarks. Due to this, we do not present the corresponding results here. We believe our proposal could not beat the other algorithms by a considerable margin because, as mentioned earlier, no process can be released only until the end of the operation. Therefore, in this case, with 16 processes at large MIFs (such as 50), the time processes spend in the collective due to the induced imbalance in the arrival time dominates the whole collective operation latency. This suggests that for the MPI_Allreduce collective with large number of PPNs our PAP-aware algorithm should be modified in a way to take less number of steps (for example, $log_2 n$ instead of $n-1$) to considerably outperform the studied algorithms.

5.5 Summary

In this section, we proposed an intra-node PAP-aware algorithm for large message reduce and MPI_Allreduce collectives. Our design for the PAP-aware reduce algorithm dynamically changes the communication pattern between the processes at each invocation of the collective based on the PAP. This way, we could improve the latency by minimizing the time each process spends in the collective waiting for other processes arrival which leads to the decrease in the average time each process spends in the collective routine. Then, we proposed a PAP-aware MPI_Allreduce by adding a broadcast operation after the proposed reduce operation. We evaluated our algorithms against state-of-the-art algorithms in MVAPICH with imbalanced microbenchmarks on two platforms. The proposed reduce operation performed 30% and 73% better than the best experimented algorithm, on Cedar with 4 PPN and Helios with 16 PPN, respectively. The PAP-aware MPI_Allreduce operation also achieved up to 44% performance improvement among the tested algorithms on Cedar with 4 PPN, while on Helios with 16 PPN our MPI_Allreduce algorithm could only deliver comparable performance with respect to the best performing algorithm. In the next chapter, we investigate a cluster-wide MPI_Allreduce algorithm capable of delivering high-performance under imbalanced workloads for large messages.

Chapter 6

Efficient Cluster-wide PAP-tolerant MPI_Allreduce for Large Messages

6.1 Motivation

In Chapter 4 and Chapter 5 we proposed algorithms capable of exploiting process imbalance to improve the performance of intra-node MPI_Reduce and MPI_Allreduce collectives for small and large message sizes, respectively. In addition, in Chapter 3 we observed that Horovod as one of the most commonly used distributed DL frameworks suffers from imbalanced process arrival patterns especially when we measure the skew metrics cluster-wide. Therefore, in this chapter we investigate a cluster-wide MPI_Allreduce algorithm for large messages capable of delivering high performance under imbalanced workloads.

In the following, we first present the related work and then we introduce a PAP-tolerant algorithm (hierarchical RSA) and evaluate its performance against a state-of-the-art cluster-wide MPI_Allreduce algorithm (flat RSA) with balanced and imbalanced microbenchmarks, as well as the Horovod application.

6.2 Related Work

There are only a few works that are very close to the study conducted in this chapter, such as the work by Parsons and Pai [45] covered in Chapter 4 which proposed PAP-aware hierarchical algorithms for modern hierarchical systems. Proficz et al. [52] evaluated the performance of MPI_Allreduce for a geographically-distributed environment consisting of two compute clusters, 900 km apart, interconnected by a 100 Gbps Ethernet-based optical fiber network. The authors studied the performance of a set of six MPI_Allreduce algorithms under balanced and imbalanced process arrival patterns on this testbed. The set of MPI_Allreduce algorithms includes two ring-based, two Binomial-Tree based and two hierarchical algorithms. For the ring-based algorithms, a PAP-aware ring algorithm proposed in [50], and an MPICH algorithm [64] were used. A variant of Rabenseifner's algorithm [57] performing Binomial-Tree Reduce-Scatter followed by Binomial-Tree All-gather was chosen as one of the Binomial-Tree based algorithms. A two-step along with a three-step hierarchical algorithms [30] were also chosen for the experiments. The balanced microbenchmark results showed that the hierarchical algorithms deliver significantly superior performance compared to the other studied algorithms. Also, the PAP-aware ring algorithm outperformed its regular counterpart even for the balanced PAP. This is due to the asymmetric connection and hence uneven communication latency between the nodes, leading to inevitable skew between the steps of the MPI_Allreduce algorithm among the involved processes even in perfectly balanced workloads. Therefore, this study does not provide a valid comparison between the performance of the experimented algorithms under an actual balanced workload. For imbalanced microbenchmark, the authors observed almost the same pattern where the hierarchical algorithms had the lowest latency and that the PAP-aware ring algorithm outperformed its regular counterpart. Consequently, they suggest that in scenarios with imbalanced PAPs the algorithms exploiting hierarchical structures deliver superior performance. However, this study does not provide a fair comparison between the performance of hierarchical and flat algorithms under imbalanced workloads in that they have used hierarchical and flat algorithms with completely different designs, whereas a more accurate evaluation could be made by comparing the performance of hierarchical and flat algorithms that both exploit the same design with only different number of stages (levels of hierarchy). The study also suggests that there is a need to develop and test hierarchical algorithms with PAP-aware support to further increase the performance when imbalanced PAPs are inevitable.

6.3 A Proposed PAP-tolerant MPI_Allreduce

In MVAPICH, the algorithm of interest for MPI_Allreduce for medium to large message sizes (larger than 64KB) is a flat Reduce-Scatter followed by an Allgather (RSA) algorithm because it delivers the best performance among the other algorithms in MVAPICH according to the microbenchmarks used to derive its tuning table. These microbenchmarks only measure the performance under perfectly balanced workloads when processes arrive at the collective call with similar PATs. In practical scenarios with real application workloads on different environments, however, the performance of flat algorithms might be susceptible to imbalanced PAPs because of the inherent all-to-all data dependency that they introduce among all the processes which acts like an unnecessary implicit synchronization between all the processes in the cluster. Hierarchical algorithms, on the other hand, introduce less data dependency among the processes. For instance, in hierarchical algorithms, the first phase of the algorithm is usually an intra-node collective operation that only requires synchronization between the processes residing on the same node. Therefore, we anticipate that the hierarchical algorithms might be less prone to performance degradation in the presence of imbalanced PAPs and hence a better algorithm for applications with imbalanced PAPs. To be able to present a fair comparison between the two algorithms, we compare the performance of a flat cluster-wide MPI_Allreduce algorithm (flat RSA) against its hierarchical counterpart (hierarchical RSA) with balanced and imbalanced microbenchmarks as well as an application (Horovod).

6.4 Evaluation Results and Analysis

6.4.1 Microbenchmark Studies

We measured the performance of the flat RSA algorithm, the default algorithm in MVAPICH, against a hierarchical MPI_Allreduce algorithm for medium to large message sizes (larger than 64KB) on 2 to 32 nodes with 4 PPN on Cedar cluster computer. The hierarchical algorithm for the aforementioned message range consists of three phases as follows:

- Phase 1: Intra-node reduce by the leader process (RSG)
- Phase 2: Inter-node MPI_Allreduce among the leader processes (RSA)
- Phase 3: Intra-node broadcast by the leader process

Microbenchmark Results for MPI_Allreduce with Balanced PATs

We first study the performance of the two algorithms under a balanced workload. Figure 6.1 through Figure 6.5 compare the performance of the flat RSA algorithm (Def-Mvapich) and the previously introduced hierarchical algorithm (Hierarchical-RSA) with a perfectly balanced microbenchmark provided in Listing 4.1.

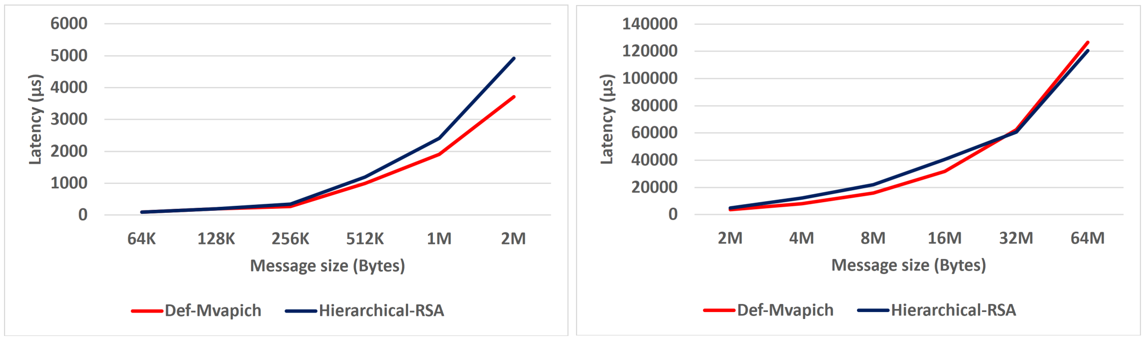


Figure 6.1: Balanced microbenchmark performance comparison between Def-Mvapich and Hierarchical-RSA algorithms for 8 GPUs evenly distributed among 2 nodes on Cedar

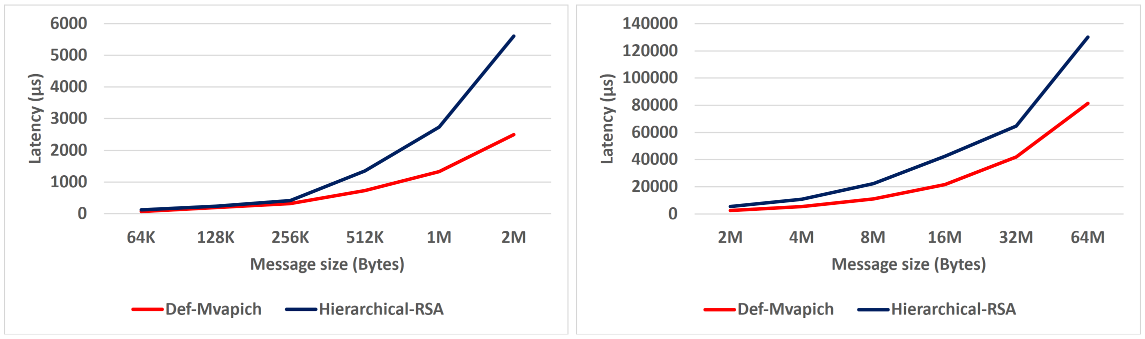


Figure 6.2: Balanced microbenchmark performance comparison between Def-Mvapich and Hierarchical-RSA algorithms for 16 GPUs evenly distributed among 4 nodes on Cedar

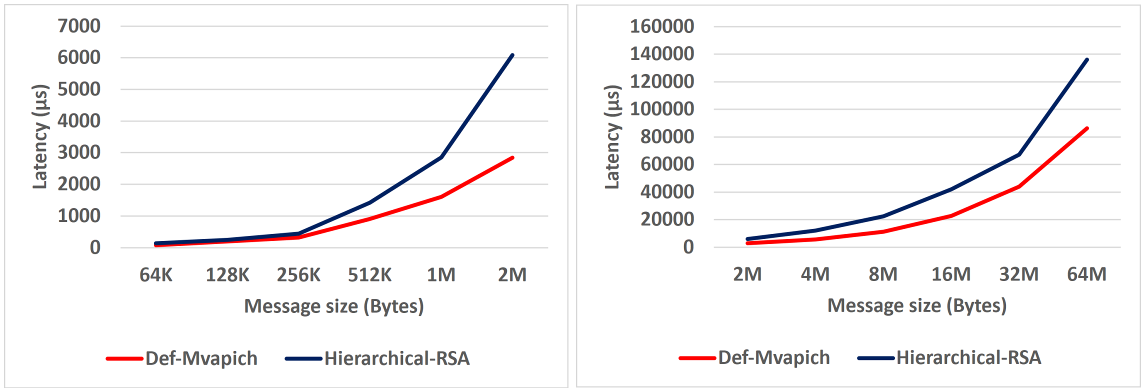


Figure 6.3: Balanced microbenchmark performance comparison between Def-Mvapich and Hierarchical-RSA algorithms for 32 GPUs evenly distributed among 8 nodes on Cedar

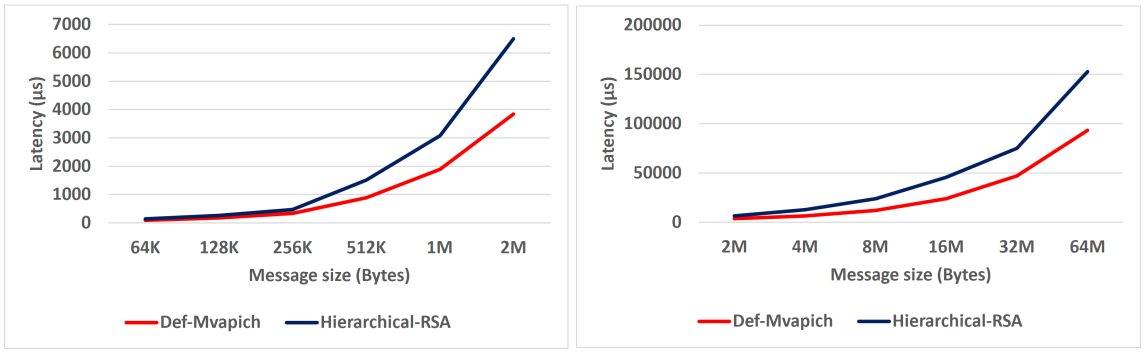


Figure 6.4: Balanced microbenchmark performance comparison between Def-Mvapich and Hierarchical-RSA algorithms for 64 GPUs evenly distributed among 16 nodes on Cedar

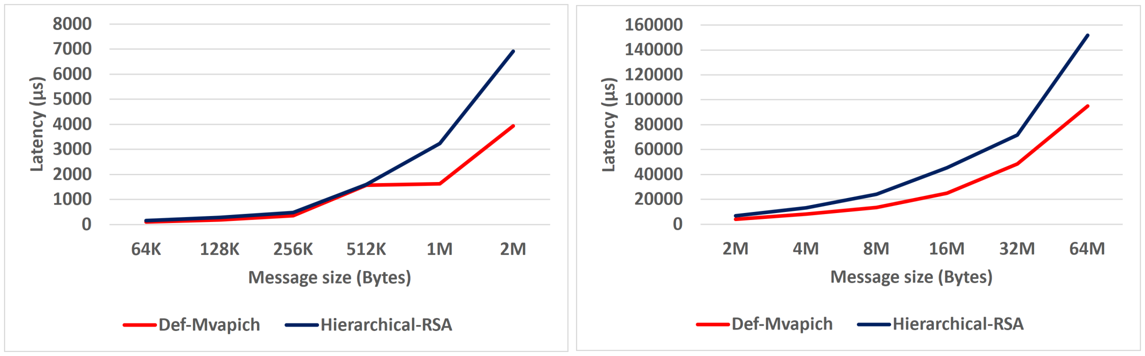


Figure 6.5: Balanced microbenchmark performance comparison between Def-Mvapich and Hierarchical-RSA algorithms for 128 GPUs evenly distributed among 32 nodes on Cedar

The results show that for all the experiments, the flat RSA algorithm outperforms the hierarchical algorithm almost for all the message sizes. This demonstrates the reason why the flat RSA has been chosen as the default algorithm for medium to large message sizes by MVAPICH.

Microbenchmark Results for MPI_Allreduce with Imbalanced PATs

In this sub-section, we measure the performance of the flat and hierarchical algorithms under an imbalanced workload. Listing 4.2 provides the pseudo-code for the imbalanced microbenchmark. The maximum imbalance factor (MIF) in this microbenchmark was chosen in a way to mimic the cluster-wide imbalanced process arrival pattern of Horovod for large messages which was studied in Chapter 3.

Figure 6.6 to Figure 6.10 present the latency of the flat RSA algorithm (Def-Mvapich) and the hierarchical algorithm (Hierarchical-RSA) under the imbalanced microbenchmark. The results show that for all the experiments, the hierarchical algorithm has a smaller latency than the flat RSA algorithm for most of the message sizes between 64KB to 64MB.

In Table 6.1, the average, maximum, and minimum improvement of hierarchical algorithm over the flat algorithm is provided. For the maximum and minimum improvements, the corresponding message size has been provided. As it can be seen, the average improvement among all the message sizes within the studied message range (64KB to 64MB) is greater than 20% for all the scenarios. The provided imbalanced microbenchmark results in this section depict the superior performance of the hierarchical algorithm over the flat algorithm in the presence of imbalanced workloads. We believe this superior performance is due to the fact that in the hierarchical algorithm the early arriving processes will only need to wait for other processes on their own node to arrive to start the communication, whereas in the flat algorithm the communication progression is hampered until all the processes arrive.

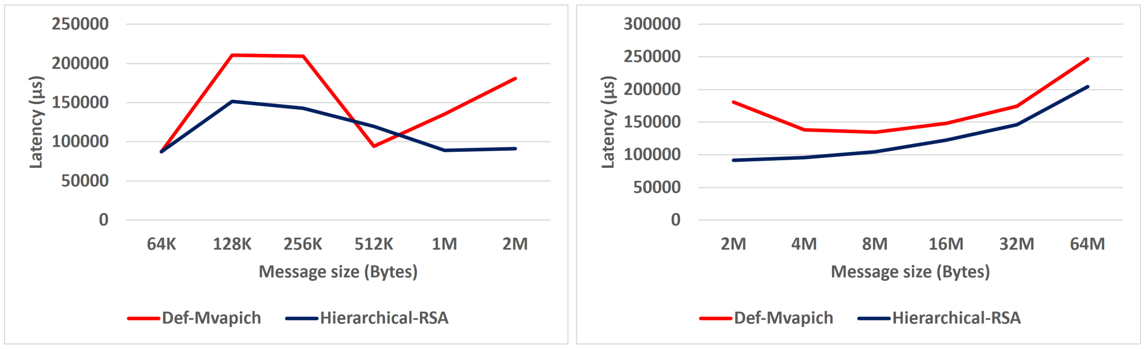


Figure 6.6: Imbalanced microbenchmark performance comparison between Def-Mvapich and Hierarchical-RSA algorithms for 8 GPUs evenly distributed among 2 nodes on Cedar

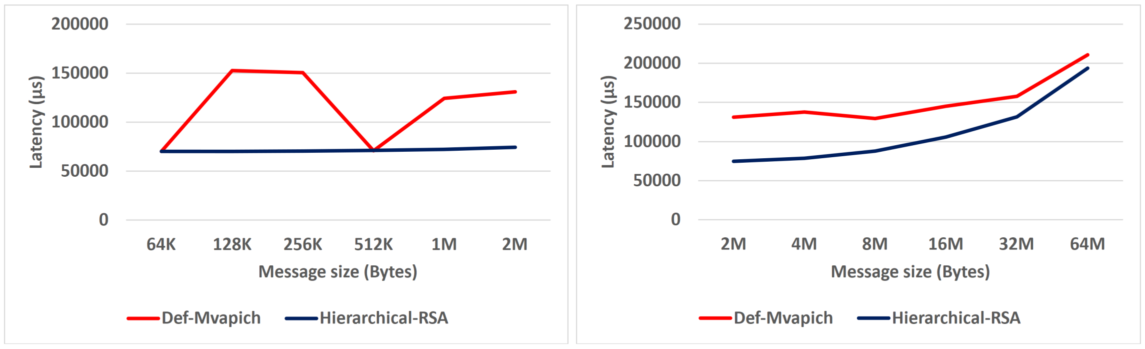


Figure 6.7: Imbalanced microbenchmark performance comparison between Def-Mvapich and Hierarchical-RSA algorithms for 16 GPUs evenly distributed among 4 nodes on Cedar

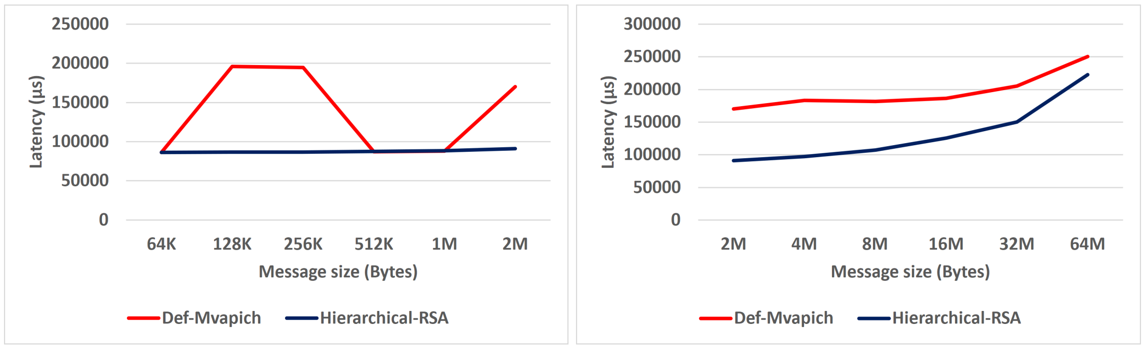


Figure 6.8: Imbalanced microbenchmark performance comparison between Def-Mvapich and Hierarchical-RSA algorithms for 32 GPUs evenly distributed among 8 nodes on Cedar

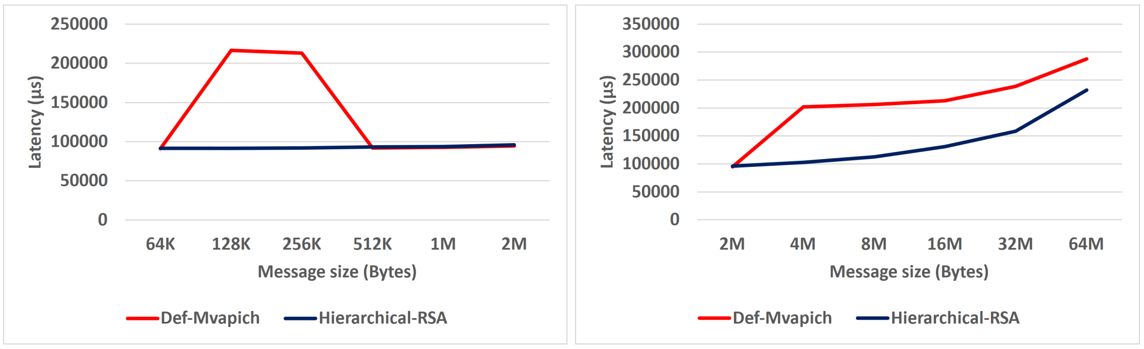


Figure 6.9: Imbalanced microbenchmark performance comparison between Def-Mvapich and Hierarchical-RSA algorithms for 64 GPUs evenly distributed among 16 nodes on Cedar

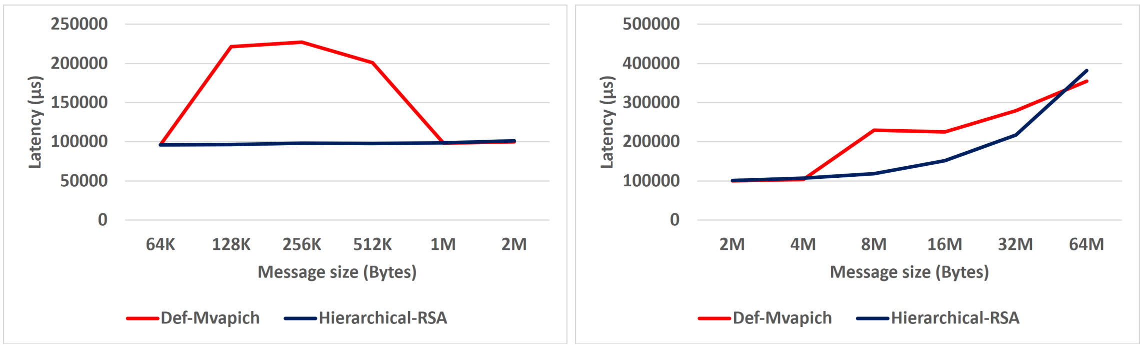


Figure 6.10: Imbalanced microbenchmark performance comparison between Def-Mvapich and Hierarchical-RSA algorithms for 128 GPUs evenly distributed among 32 nodes on Cedar

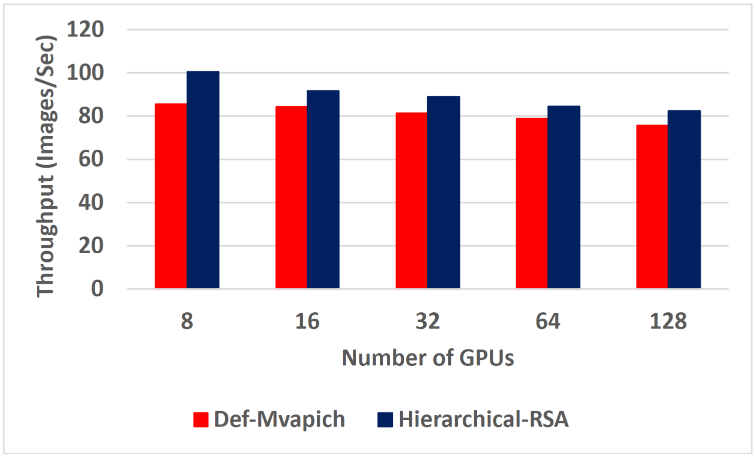
Table 6.1: Performance improvement of Hierarchical-RSA achieved over Def-Mvapich algorithm up to 128 GPUs on Cedar

GPU Count	Improvement				
	Average	Max	Message Size	Min	Message Size
8	20.01%	49.5%	2MB	16.36%	32MB
16	28.97%	54.01%	128KB	8.15%	64MB
32	28.64%	55.81%	256KB	11.15%	64MB
64	26.94%	57.64%	128KB	19.43%	64MB
128	23.17%	56.76%	256KB	22.23%	32MB

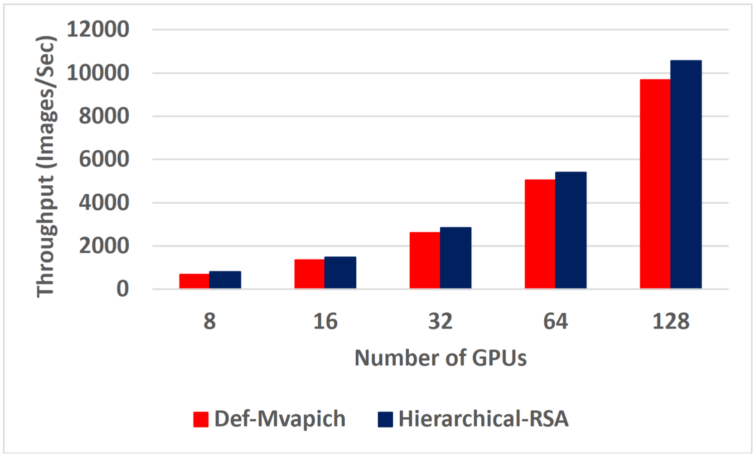
6.4.2 Horovod Application Results

In this sub-section, we measure the performance of the flat RSA algorithm (Def-Mvapich) and the hierarchical algorithm (Hierarchical-RSA) for Horovod application. For this test we utilized the synthetic benchmarks within Horovod which provide the throughput of the image classification task by the number of images processed per second (Images/Sec). Figure 6.11 exhibits the per-GPU and total Horovod throughput of the flat RSA and the hierarchical-RSA algorithms for 8 to 128 GPUs. The results show that hierarchical-RSA outperforms the Def-Mvapich algorithm for all the GPU counts from 8 to 128 by the maximum, minimum, and average of 17%, 7%, and 10%, respectively.

As the communication characterization results of Horovod in Chapter 3 showed, the cluster-wide imbalance-factor is significantly larger than the node-wide imbalance-factor in Deep Learning frameworks, which suggests that in Horovod, the processes on the same node arrive at the collective call with much less delay with respect to each other, than all the processes on the cluster. Consequently, DL applications with the aforementioned characteristics could benefit from the hierarchical algorithms in that with these algorithms the early arriving processes on each node will not need to wait long for the processes on their node to arrive to start the intra-node step, while with flat algorithms, the waiting time between all the processes cluster-wide to start the operation is much larger. Hence, the hierarchical algorithm delivers much better performance compared to its flat counterpart under the inherently imbalanced PAP of Horovod.



(a) Per-GPU throughput



(b) Total throughput

Figure 6.11: Horovod performance comparison between Def-Mvapich and Hierarchical-RSA algorithms for 8 to 128 GPUs evenly distributed among 2 to 32 nodes on Cedar

6.5 Summary

In this section, we showed that although the flat RSA algorithm outperforms its hierarchical counterpart for balanced PAPs, it performs poorly in the presence of imbalanced workloads. In a nutshell, the hierarchical algorithms deliver a better collective communication progression in the presence of imbalanced PAPs compared to flat algorithms for two reasons. First, with hierarchical algorithms, the processes on each node can proceed with their intra-node collective operation step without the need to wait for the late processes on the other nodes to arrive (less data dependency among processes). And second, the waiting time for the arrival of intra-node processes is significantly smaller than the delay for arrival of all the processes on the cluster (less waiting time before starting the operation). Therefore, when the PAP is imbalanced, hierarchical algorithms should be the algorithm of interest because they deliver higher performance. Exploiting the hierarchical algorithm instead of the flat algorithm improved the Horovod application results for all the studies we conducted with different GPU counts by an average of 10%. The observed improvement demonstrated that the inherently imbalanced PAP of Horovod could benefit from hierarchical algorithms' PAP-tolerant characteristics.

Chapter 7

Conclusion and Future Work

The number of processing units in HPC systems is increasing at a face pace. As the number of processes increases, the communication between them affects the performance of parallel applications more prominently. MPI is the commonly used parallel programming standard in the HPC community. The MPI standard introduces different communication semantics, such as collective operations. Collective operations facilitate one-to-many, many-to-one, and many-to-many inter-process communications in an optimized, scalable, and yet convenient way. Therefore, they have been extensively used and play a pivotal role in many MPI applications. Researchers have been improving the performance of MPI collective communication operations from different aspects for a long time. Most of these studies, however, are based on the premise that all the processes arrive at the collective call at the same time. Studies have shown that such an assumption is impractical in HPC platforms and that the process arrival pattern, even in MPI applications with perfectly balanced workloads, is sufficiently imbalanced to affect the performance negatively. In this thesis, we studied the communication characterization of a famous distributed DL framework, Horovod, including the investigation of the arrival pattern of the MPI processes. Then, we proposed two PAP-aware algorithms for intra-node MPI_Allreduce operation with small and large message sizes. Finally, we investigated a hierarchical PAP-tolerant algorithm for large message MPI_Allreduce operation capable of achieving high performance under imbalanced workloads. In the following, we will highlight the research contributions made in this study and present potential future directions.

7.1 Conclusion

In Chapter 2, we provided the background regarding the research explored in this study. In Chapter 3, we presented the communication characterization of the Horovod application. First, we observed that MPI_Allreduce is the most important collective operation used in the Horovod both in terms of the number of calls and contribution to the runtime. Next, we investigated the arrival pattern of the processes for allreduce calls in Horovod. We noted a large asynchrony between the arrival times of the involved processes at each invocation of the allreduce operations. Especially for allreduce collectives with small messages, the maximum/average case imbalance factors were huge.

In Chapter 4, we took up the challenge to design an intra-node PAP-aware allreduce algorithm for small messages. In modern HPC clusters, cutting-edge algorithms designed for allreduce collective operations with small messages utilize the memory shared between processes residing on a node for intra-node communication. Unfortunately, however, the native shared memory algorithms do not take into account the arrival pattern of the processes and hence perform poorly under imbalanced PAPs. We implemented our PAP-aware design on top of the shared memory algorithm utilized in MVAPICH. In our algorithm, at each invocation of the collective call, we dynamically choose the leader based on the arrival time of the processes. We evaluated our design against the native algorithms utilized in MVAPICH for allreduce operations with messages up to 64KB on two platforms. For

MPI_Allreduce, at MIF 20, we could achieve the highest average improvements of 20% on Beluga, and 22% on Cedar, over the native algorithms across all the messages smaller than 64KBs with 32 PPN.

It should be mentioned that our work in Chapter 4 for intra-node PAP-aware allreduce for small messages outperforms the other algorithms, especially when the intra-node process count increases and the message size is larger than 256B. Unfortunately, however, with the Horovod application, we could not go beyond four processes per node due to its single process per GPU configuration. In addition, as mentioned in Chapter 3, eight-byte messages account for the majority of the small message allreduce calls of Horovod. Therefore, at the Horovod level, our proposal only delivers comparable performance to the other algorithms, while at the microbenchmark level, we could see the full potential of our proposal in delivering significant improvements.

In Chapter 5, we proposed an algorithm capable of exploiting process imbalance to improve the performance of intra-node MPI_Reduce and MPI_Allreduce collectives with large message sizes. In our proposed algorithm, based on the arrival order of the processes at each invocation of the collective call, we allow the early arriving processes to start the reduce operation, contribute their data and leave the collective call as soon as possible without waiting for the late arrival processes. Minimizing the time each process spends in the collective site, our PAP-aware algorithm could improve the latency of the MPI_Reduce and MPI_Allreduce operations in the presence of imbalanced PAPs. We evaluated the performance of our algorithm using imbalanced microbenchmarks with different MIFs on two platforms. On Cedar and Helios, with four and sixteen processes per node, for MPI_Reduce, we could achieve the highest average improvement of 26% and 63% over the Binomial-Tree algorithm across all the messages between 64KB to 64MB at MIF 20 and 75, respectively. In addition, for the allreduce operation across all the message sizes within the range of 64KB to 64MB, we observed the highest average improvement of 20% over the best experimented algorithm at MIF 20 on Cedar with 4 PPN.

One limitation of PAP-aware algorithms including our design is that their performance is dependent on the extent of imbalance in the PAP. Therefore, in Chapter 5, although our proposal for the PAP-aware allreduce algorithm for large message sizes improves the performance for imbalanced MIFs at the microbenchmark level, it cannot show its impact on the performance at the application level (Horovod). This is due to the fact that our design starts to deliver better performance than other algorithms at slightly higher MIFs compared to the measured MIFs of Horovod for large messages.

Chapter 6 investigated a PAP-tolerant cluster-wide allreduce algorithm capable of delivering high performance under imbalanced workloads for large messages. Using microbenchmarks, we showed that although the flat algorithms deliver the best performance in perfectly balanced workloads, they suffer from performance degradation when the PAP is imbalanced. Hierarchical algorithms, on the other hand, outperform their flat counterparts under imbalanced workloads due to the less data dependency they impose on the processes. The imbalanced microbenchmark study showed that the hierarchical algorithm, compared to its flat counterpart, improves the latency of the allreduce operation by up to 29% on 16 GPUs averaged across all the message sizes within the range of 64KB to 64MB. Furthermore, we evaluated our study using the Horovod application on 8 to 128 GPUs and observed the average and maximum throughput improvements of 10% and 17%, respectively.

7.2 Future Work

Our plans to extend the research conducted in this study revolve around developing algorithms to tackle the challenges in MPI collective communications introduced by imbalanced process arrival patterns. Considering the communication characteristics of parallel applications and the cutting-edge hardware/software features exploited in new parallel computers, we will investigate the opportunities to enhance the performance of MPI collective operations in imbalanced PAP scenarios. In the following, we will review the opportunities to the algorithms and mechanisms proposed in this thesis that could be further studied.

7.2.1 Investigating Other MPI Applications

We plan to continue our work in Chapter 3 with studying the communication characteristics of other distributed DL frameworks such as ChainerMN [9] and DL applications such as CosmoFlow [41] as well as HPC applications. CosmoFlow is built on top of the TensorFlow framework and uses Deep Learning on 3D volumes to learn the physics of the universe and can be scaled up to run on more than 8,000 CPU nodes. Then, we will utilize our PAP-aware collective algorithms proposed in Chapters 4, 5, and 6 for those HPC applications that suffer from imbalanced PAPs to achieve high performance.

7.2.2 Extension of Proposed PAP-aware Algorithms to Other Collectives and Process Counts

While our focus was on MPI_Allreduce and MPI_Reduce collectives in this study, our proposals in Chapters 4 to 6 can be extended to other MPI collectives. Especially, MPI_Allgather and MPI_Bcast can benefit from our designs with minimal modifications in that they have a similar communication pattern to allreduce and reduce collectives, respectively. Designing PAP-aware intra-node and inter-node allgather and broadcast collectives for small and large message sizes and evaluating their performance under balanced and imbalanced PAPs will be investigated in the future. Furthermore, as mentioned in Section 5.4.1, we plan to modify our PAP-aware MPI_Allreduce algorithm proposed in Chapter 5 to decrease its number of steps so that it could outperform the other algorithms by a more significant margin for allreduce collectives with a large number of PPNs.

7.2.3 Minimizing the Overhead of Control Messages

In our proposed algorithms in Chapters 4 and 5, we used small control messages through shared memory as a way to exchange synchronization signals and inform every other process of the arrival pattern of the already arrived processes. The synchronization messages introduce an extra overhead to the performance of our designs. One way to minimize the overhead of signaling is to use Remote Direct Memory Access (RDMA). In RDMA communication, each process can directly access the memory of another process without their involvement and with bypassing its operating system. This permits high-throughput, low-latency communication, which can potentially minimize the overhead of control messages leading to better performance improvements for the designed algorithms.

7.2.4 Extension of Our Node-wide PAP-aware Algorithms to Cluster-wide

Our PAP-aware algorithms for the intra-node allreduce operation presented in Chapters 4 and 5 were designed using intra-node reduce followed by an intra-node broadcast operation. It should be noted that these algorithms can be extended for cluster-wide operations with the addition of an inter-node allreduce step right before the intra-node broadcast operation. In addition, the inter-node allreduce step can utilize any allreduce algorithm to perform well in different circumstances since our designs do not burden any limit on the choice of the inter-node allreduce algorithm. Therefore, we plan to use some well-known non-PAP-aware allreduce algorithms for the inter-node step and our PAP-aware algorithms for the intra-node step to design a cluster-wide PAP-aware allreduce algorithm and evaluate the performance of it using balanced and imbalanced microbenchmarks as well as HPC/DL applications.

Furthermore, we plan to extend our designs for intra-node PAP-aware algorithms to inter-node allreduce algorithms for different message sizes. This way, we could design a truly PAP-aware cluster-wide allreduce algorithm that benefits from PAP-awareness both in intra- and inter-node steps. However, one main challenge for designing inter-node PAP-aware collective operations is that no shared memory is available between the processes residing on different nodes. Therefore, the synchronization and signaling between the processes required to extract the arrival pattern cannot be performed through shared memory. In this case, RDMA can be used as the means of signaling between the processes while introducing minimum overhead to the performance of the proposed algorithms.

7.2.5 PAP-aware Collectives for NICs

One way to improve the performance of HPC applications by overlapping communication and computation is to offload some of the communication/computation tasks from the host processors to the programmable processing units embedded on the *Network Interface Cards (NICs)*. To the best of our knowledge, there have not been any studies conducted on the process arrival pattern of the collective communications performed with the assistance of processing units on NICs. Therefore, characterizing the arrival pattern of the MPI processes for different applications in such environments is necessary. With the information derived from such characterizations we would extend our PAP-aware algorithms or design new algorithms capable of delivering high-performance in imbalanced workloads by making the NICs aware of PAP and efficiently exploiting the delays between processes arrival.

7.2.6 Adaptive Algorithm Selection based on MIFs

To make all the proposals in this study in Chapter 4 to Chapter 6 useful in practice, a mechanism could be developed that detects arrival patterns and, based on different degrees of imbalanced PAPs, invokes appropriate algorithms. One way to achieve this goal is to dynamically measure the MIFs at the runtime and switch to the optimal algorithm for a certain number of collective invocations before assessing the MIFs again. However, this method might introduce some significant overhead to the application runtime due to its extra computation. The other alternative is to run the application once in advance and derive its PAP metrics, and then statistically choose the appropriate algorithms for different message sizes based on the application's MIFs.

Bibliography

[1] CIFAR-10 and CIFAR-100 datasets. https://www.cs.toronto.edu/ kriz/cifar.html. Accessed November 26, 2021.

[2] InfiniBand Trade Association. http://www.infinibandta.org. Accessed November 26, 2021.

[3] Message Passing Interface (MPI 4.0). https://www.mpi-forum.org. Accessed November 26, 2021.

[4] MPICH high-performance portable MPI. https://www.mpi-forum.org. Accessed November 26, 2021.

[5] Nvidia. NCCL Library. https://github.com/NVIDIA/nccl. Accessed November 26, 2021.

[6] PGAS Forum. http://www.pgas.org. Accessed November 26, 2021.

[7] TOP500. https://www.top500.org. Accessed November 26, 2021.

[8] A. Afsahi and Y. Qian. Remote shared memory over sun fire link interconnect. In *Fifteenth IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS)*, volume 1, pp. 381–386, 2003.

[9] T. Akiba, K. Fukuda, and S. Suzuki. ChainerMN: Scalable distributed deep learning framework. *arXiv preprint arXiv:1710.11351*, 2017.

[10] Q. Ali, S. P. Midkiff, and V. S. Pai. Efficient high performance collective communication for the Cell blade. In *Proceedings of the 23rd international conference on Supercomputing (ICS)*, pp. 193–203, 2009.

[11] A. A. Awan, K. Hamidouche, A. Venkatesh, and D. K. Panda. Efficient large message broadcast using NCCL and CUDA-aware MPI for deep learning. In *Proceedings of the 23rd European MPI Users' Group Meeting (EuroMPI)*, pp. 15–22, 2016.

[12] A. A. Awan, A. Jain, C.-H. Chu, H. Subramoni, and D. K. Panda. Communication Profiling and Characterization of Deep-Learning Workloads on Clusters With High-Performance Interconnects. *IEEE Micro*, 40(1):35–43, 2019.

[13] T. Ben-Nun and T. Hoefler. Demystifying parallel and distributed Deep Learning: An in-depth concurrency analysis. *ACM Computing Surveys (CSUR)*, 52(4):1–43, 2019.

[14] D. E. Bernholdt, S. Boehm, G. Bosilca, M. Gorentla Venkata, R. E. Grant, T. Naughton, H. P. Pritchard, M. Schulz, and G. R. Vallee. A survey of MPI usage in the U.S. exascale computing project. *Concurrency and Computation: Practice and Experience (CCPE)*, 32(3):e4851, 2020.

[15] 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 *23rd Annual IEEE Symposium on High-Performance Interconnects (HotI)*, pp. 1–9. IEEE, 2015.

[16] E. Chan, M. Heimlich, A. Purkayastha, and R. Van De Geijn. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience (CCPE)*, 19(13):1749–1783, 2007.

[17] C.-H. Chu, X. Lu, A. A. Awan, H. Subramoni, J. Hashmi, B. Elton, and D. K. Panda. Efficient and scalable multi-source streaming broadcast on GPU clusters for deep learning. In *2017 46th International Conference on Parallel Processing (ICPP)*, pp. 161–170. IEEE, 2017.

[18] S. Chunduri, S. Parker, P. Balaji, K. Harms, and K. Kumaran. Characterization of MPI usage on a production supercomputer. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 386–400. IEEE, 2018.

[19] J. Diaz, C. Munoz-Caro, and A. Nino. A survey of parallel programming models and tools in the multi and many-core era. *IEEE Transactions on parallel and distributed systems*, 23(8):1369–1386, 2012.

[20] G. E. Fagg, S. S. Vadhiyar, and J. J. Dongarra. ACCT: automatic collective communications tuning. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting (EuroPVM/MPI)*, pp. 354–361. Springer, 2000.

[21] A. Faraj, P. Patarasuk, and X. Yuan. A study of process arrival patterns for MPI collective operations. *International Journal of Parallel Programming*, 36(6):543–570, 2008.

[22] A. Faraj and X. Yuan. Automatic generation and tuning of MPI collective communication routines. In *Proceedings of the 19th annual international conference on Supercomputing (ICS)*, pp. 393–402, 2005.

[23] A. Faraj, X. Yuan, and D. Lowenthal. STAR-MPI: self tuned adaptive routines for MPI collective operations. In *Proceedings of the 20th annual international conference on Supercomputing (ICS)*, pp. 199–208, 2006.

[24] I. Faraji and A. Afsahi. GPU-aware intranode MPI_Allreduce. In *Proceedings of the 21st European MPI Users' Group Meeting (EuroMPI)*, pp. 45–50, 2014.

[25] 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)*, pp. 47–50, 2015.

[26] I. Faraji and A. Afsahi. Design considerations for GPU-aware collective communications in MPI. *Concurrency and Computation: Practice and Experience (CCPE)*, 30(17):e4667, 2018.

[27] S. M. Ghazimirsaeed, S. H. Mirsadeghi, and A. Afsahi. Communication-aware message matching in MPI. *Concurrency and Computation: Practice and Experience (CCPE)*, 32(3):e4862, 2020.

[28] R. Graham, M. G. Venkata, J. Ladd, P. Shamis, I. Rabinovitz, V. Filipov, and G. Shainer. Cheetah: A framework for scalable hierarchical collective operations. In *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pp. 73–83. IEEE, 2011.

[29] R. L. Graham and G. Shipman. MPI support for multi-core architectures: Optimized shared memory collectives. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting (EuroPVM/MPI)*, pp. 130–140. Springer, 2008.

[30] K. Hasanov and A. Lastovetsky. Hierarchical redesign of classic MPI reduction algorithms. *The Journal of Supercomputing*, 73(2):713–725, 2017.

[31] T. Hoefler, T. Schneider, and A. Lumsdaine. Accurately measuring collective operations at massive scale. In *2008 IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pp. 1–8. IEEE, 2008.

[32] G. Inozemtsev and A. Afsahi. Designing an offloaded nonblocking MPI_Allgather collective using CORE-Direct. In *2012 IEEE International Conference on Cluster Computing (Cluster)*, pp. 477–485. IEEE, 2012.

[33] K. Kandalla, H. Subramoni, G. Santhanaraman, M. Koop, and D. K. Panda. Designing multi-leader-based allgather algorithms for multi-core clusters. In *2009 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pp. 1–8. IEEE, 2009.

[34] K. Kandalla, A. Venkatesh, K. Hamidouche, S. Potluri, D. Bureddy, and D. K. Panda. Designing optimized MPI broadcast and allreduce for Many Integrated Core (MIC) InfiniBand clusters. In *21st Annual IEEE Symposium on High-Performance Interconnects (HotI)*, pp. 63–70. IEEE, 2013.

[35] 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*, pp. EPSC2016–13554, 2016.

[36] C. K. Leung, O. A. Sarumi, and C. Y. Zhang. Predictive analytics on genomic data with high-performance computing. In *2020 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pp. 2187–2194. IEEE, 2020.

[37] S. Li, T. Hoefler, C. Hu, and M. Snir. Improved MPI collectives for MPI processes in shared address spaces. *The Journal of Cluster Computing*, 17(4):1139–1155, 2014.

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

[39] P. Marendić, J. Lemeire, T. Haber, D. Vučinić, and P. Schelkens. An investigation into the performance of reduction algorithms under load imbalance. In *European Conference on Parallel Processing (Euro-Par)*, pp. 439–450. Springer, 2012.

[40] P. Marendic, J. Lemeire, D. Vucinic, and P. Schelkens. A novel MPI reduction algorithm resilient to imbalances in process arrival times. *The Journal of Supercomputing*, 72(5):1973–2013, 2016.

[41] A. Mathuriya, D. Bard, P. Mendygral, L. Meadows, J. Arnemann, L. Shao, S. He, T. Kärnä, D. Moise, and S. J. Pennycook. CosmoFlow: Using Deep Learning to learn the universe at scale. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 819–829. IEEE, 2018.

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

[43] C. Obrecht, F. Kuznik, B. Tourancheau, and J.-J. Roux. Scalable lattice Boltzmann solvers for CUDA GPU clusters. *Parallel Computing*, 39(6-7):259–270, 2013.

[44] B. S. Parsons. Accelerating MPI collective communications through hierarchical algorithms with flexible inter-node communication and imbalance awareness. PhD thesis, Purdue University, 2015.

[45] B. S. Parsons and V. S. Pai. Exploiting process imbalance to improve MPI collective operations in hierarchical systems. In *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS)*, pp. 57–66, 2015.

[46] P. Patarasuk and X. Yuan. Efficient MPI Bcast across different process arrival patterns. In *2008 IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pp. 1–11. IEEE, 2008.

[47] S. Pellegrini, T. Hoefler, and T. Fahringer. On the effects of CPU caches on MPI point-to-point communications. In *2012 IEEE International Conference on Cluster Computing (Cluster)*, pp. 495–503. IEEE, 2012.

[48] F. Petrini, S. Coll, E. Frachtenberg, and A. Hoisie. Hardware-and software-based collective communication on the Quadrics network. In *Proceedings IEEE International Symposium on Network Computing and Applications (NCA)*, pp. 24–35. IEEE, 2001.

[49] J. Pjesivac-Grbovic. Towards automatic and adaptive optimizations of MPI collective operations. PhD thesis, University of Tennessee, 2007.

[50] J. Proficz. Improving all-reduce collective operations for imbalanced process arrival patterns. *The Journal of Supercomputing*, 74(7):3071–3092, 2018.

[51] J. Proficz and K. M. Ocetkiewicz. Improving Clairvoyant: reduction algorithm resilient to imbalanced process arrival patterns. *The Journal of Supercomputing*, 77(6):6145–6177, 2021.

[52] J. Proficz, P. Sumionka, J. Skomiał, M. Semeniuk, K. Niedzielewski, and M. Walczak. Investigation into MPI all-reduce performance in a distributed cluster with consideration of imbalanced process arrival patterns. In *International Conference on Advanced Information Networking and Applications*, pp. 817–829. Springer, 2020.

[53] Y. Qian and A. Afsahi. Efficient shared memory and RDMA based collectives on multi-rail QsNet II SMP clusters. *Cluster Computing*, 11(4):341–354, 2008.

[54] Y. Qian and A. Afsahi. Process arrival pattern and shared memory aware alltoall on InfiniBand. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting (EuroPVM/MPI)*, pp. 250–260. Springer, 2009.

[55] 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.

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

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

[58] H. Ritzdorf and J. L. Traff. Collective operations in NEC's high-performance MPI libraries. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, pp. 1–10. IEEE, 2006.

[59] F. Seide and A. Agarwal. CNTK: Microsoft's open-source deep-learning toolkit. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 2135–2135, 2016.

[60] A. Sergeev and M. Del Balso. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799*, 2018.

[61] S. Sistare, R. Vandevaart, and E. Loh. Optimization of MPI collectives on clusters of large-scale SMPs. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (SC)*, pp. 23–36, 1999.

[62] S. Sur, U. K. R. Bondhugula, A. Mamidala, H.-W. Jin, and D. K. Panda. High performance RDMA based all-to-all broadcast for InfiniBand clusters. In *International Conference on High-Performance Computing (HiPC)*, pp. 148–157. Springer, 2005.

[63] Y. H. Temucin, A. Sojoodi, P. Alizadeh, and A. Afsahi. Efficient Multi-Path NVLink/PCIe-Aware UCX based Collective Communication for Deep Learning. In *28th Annual IEEE Symposium on High-Performance Interconnects (HotI)*, pp. 1–10. IEEE, 2021.

[64] 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.

[65] V. Tipparaju, J. Nieplocha, and D. K. Panda. Fast collective operations using shared and remote memory access protocols on clusters. In *Proceedings International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 1–10. IEEE, 2003.

[66] J. L. Träff, A. Ripke, C. Siebert, P. Balaji, R. Thakur, and W. Gropp. A pipelined algorithm for large, irregular all-gather problems. *The International Journal of High Performance Computing Applications*, 24(1):58–68, 2010.

[67] M. G. Venkata, P. Shamis, R. Sampath, R. L. Graham, and J. S. Ladd. Optimizing blocking and nonblocking reduction operations for multicore systems: Hierarchical design and implementation. In *2013 IEEE International Conference on Cluster Computing (Cluster)*, pp. 1–8. IEEE, 2013.

[68] S. White and L. V. Kale. Optimizing point-to-point communication between adaptive MPI endpoints in shared memory. *Concurrency and Computation: Practice and Experience (CCPE)*, 32(3):e4467, 2020.

[69] T. S. Woodall, R. L. Graham, R. H. Castain, D. J. Daniel, M. W. Sukalski, G. E. Fagg, E. Gabriel, G. Bosilca, T. Angskun, J. J. Dongarra, et al. TEG: A high-performance, scalable, multi-network point-to-point communications methodology. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting (EuroPVM/MPI)*, pp. 303–310. Springer, 2004.

[70] M.-S. Wu, R. A. Kendall, and K. Wright. Optimizing collective communications on SMP clusters. In *2005 International Conference on Parallel Processing (ICPP)*, pp. 399–407. IEEE, 2005.

[71] S. A. Zenios. High-performance computing in finance: The last 10 years and the next. *Parallel Computing*, 25(13-14):2149–2175, 1999.

[72] H. Zhu, D. Goodell, W. Gropp, and R. Thakur. Hierarchical collectives in MPICH2. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting (EuroPVM/MPI)*, pp. 325–326. Springer, 2009.